MergeCOM-3

Advanced Integrator's Tool Kit

User's Manual

P/N 009-13115-02 (13)

MergeCOM-3 Advanced Integrator's Tool Kit

User's Manual

Authors: Brian Long, Jeff Serra, Steve Wranovsky

Document Revision History

Version 6

December 5,2001
Final update for v3.0.0 toolkit.

Version 7

December 13, 2001
Revised information on encapsulated data.

Version 8

February 19, 2002
Revised page numbering.

Version 9

May 1, 2002
Fixed revision number on front page, and copyright date.

Version 10

Oct 23, 2002
Updated company name/phone/fax. Added section for Deflate.

Version 11

June 24, 2003
Updated registering compression callbacks section, standard compressor/decompressor supports JPEG_2000,JPEG_2000_LOSSLESS_ONLY, information on which Pegasus libraries are used.

Version 12

September 16, 2004
Added details on the new parts of the DICOM standard and updated references on acquiring the standard. Updated documentation of service classes and SOP classes supported including Media Storage related SOP Classes and the Directory Records supported. Documented two new VRs that are now supported. Described changes in mergecom.app service lists to support asynchronous communications and extended negotiation. Added a section describing support of asynchronous communications. Changed format of compression callbacks section and added more detail on color images. Updated registered callback description to include new callback types.

Version 13

March 30, 2005
Added details on the new parts of the DICOM standard. Updated documentation of service classes and SOP classes supported including Media Storage related SOP Classes and the Directory Records supported. Added further clarifications of the use of registered callback functions and asynchronous communications. Also updated configuration options and performance tuning information.

Table of Contents

Overview

This user's manual is targeted toward the developer of medical imaging applications using the MergeCOM-3 Advanced Integrator's Tool Kit to supply DICOM network or media functionality.

MergeCOM-3 supplies you with a powerful and simplified interface to DICOM. It lets you focus on the important details of your application and immediate needs of your end users, rather than the often complex and confusing details of the DICOM Standard.

The goal of this manual is to give you basic understanding of DICOM, and a clear understanding of the Advanced Tool Kit.

The DICOM Standard

The DICOM (Digital Imaging and Communications in Medicine) Standard was originally developed by a joint committee of the American College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA) to "facilitate the open exchange of information between digital imaging computer systems in medical environments"[1].

Since its initial completion in 1993, the standard has taken hold. More and more products are advertising DICOM conformance, and more customers are requiring it. DICOM has also been incorporated as part of a developing European standard by CEN, as a Japanese standard by JIRA, and is increasingly becoming an International Standard.

DICOM Version 3.0 is composed of several hundreds of pages over sixteen separate parts. Each part of the standard focuses on a different aspect of the DICOM protocol:

Part 1- Introduction and Overview

Part 2- Conformance

Part 3- Information Object Definitions

Part 4- Service Class Specifications

Part 5- Data Structures and Encoding

Part 6- Data Dictionary

Part 7- Message Exchange

Part 8- Network Communication Support for Message Exchange

Part 9- Point-to-Point Communication Support for Message Exchange (retired)

Part 10- Common Media Storage Functions for Data Interchange

Part 11- Media Storage Application Profiles

Part 12- Media Formats and Physical Media for Data Interchange

Part 13- Print Management Point-to-Point Communication Support (retired)

Part 14- Grayscale Standard Display Function

Part 15- Security Profiles

[1] NEMA Standards Publication No. PS 3.5-1993; DICOM Part 5 - Data Structures and Encoding, p.4.

Part 16- DICOM Content Mapping Resource

Part 17- Explanatory Information

Part 18- Web Access to DICOM Persistent Objects (WADO)

A quick walk through DICOM

Part 1 of the standard gives an overview of the standard. Since this part was approved before most of the other parts were completed, it is already somewhat outdated and can be confusing.

Part 2 describes DICOM conformance and how to write a conformance statement. A conformance statement is important because it allows a network administrator to plan or coordinate a network of DICOM applications. For an application to claim DICOM conformance, it must have an accurate conformance statement.

Parts 3 and 4 define the types of services and information that can be exchanged using DICOM.

Parts 5 and 6 describe how commands and data shall be encoded so that decoding devices can interpret them.

Part 7 describes the structure of the DICOM commands, that along with related data, make up a DICOM message. This part also describes the association negotiation process, whereby two DICOM applications mutually agree upon the services they will perform over the network.

Part 8 describes how the DICOM messages are exchanged over the network using two prominent transport layer protocols; TCP/IP and OSI. This is termed the DICOM Upper Layer Protocol (DICOM UL).

Part 9 is rarely of interest, as it describes how DICOM messages shall be exchanged using the 'old' 50-pin point-to-point connection originally specified in the predecessor to DICOM (ACR/NEMA Version 2). This part has been retired from the DICOM standard.

Part 10 describes the DICOM model for the storage of medical imaging information on removable media. It specifies the contents of a DICOM File Set, the format of a DICOM File and the policies associated with the maintenance of a DICOM Media Storage Directory (DICOMDIR) structure.

Part 11 specifies Media Storage Application Profiles that standardizes a number of choices related to a specific clinical need (modality or application). This includes the specification of a specific physical medium and media format (e.g., CD-ROM, 3.5" high-density floppy, ...), as well as the types of information (objects) that can be stored within the DICOM File Set. Part 11 also includes useful templates to provide guidance in authoring media application conformance statements.

Part 12 details the characteristics of various physical medium and media formats that are referenced by the Media Storage Application Profiles of Part 11.

While parts 11 and 12 of DICOM are expected to evolve along with the introduction of new clinical procedures and the advancement of storage media and file system technology, Part 10 should remain quite stable since it specifies file formats independent of medical application or storage technology.

Part 13 details a point to point protocol for doing print management services. This part has been retired from the DICOM standard.

Part 14 specifies a standardized display function for display of grayscale images.

Part 15 specifies Security Profiles to which implementations may claim conformance. Profiles are defined for secure network transfers and secure media.

Part 16 specifies the DICOM Content Mapping Resource (DCMR) which defines the templates and context groups used elsewhere in the standard.

Part 17 consolidates informative information previously contained in other parts of the standard. It is composed of several annexes describing the use of the standard.

Part 18 specifies a web-based service for accessing and presenting DICOM persistent objects (e.g. images, medical imaging reports).

Figure 1 maps portions of the DICOM Standard dealing with networking to the ISO Open Systems Interconnection (OSI) basic reference model. The organization and terminology of the DICOM Standard corresponds closely with that used in the OSI Standard.

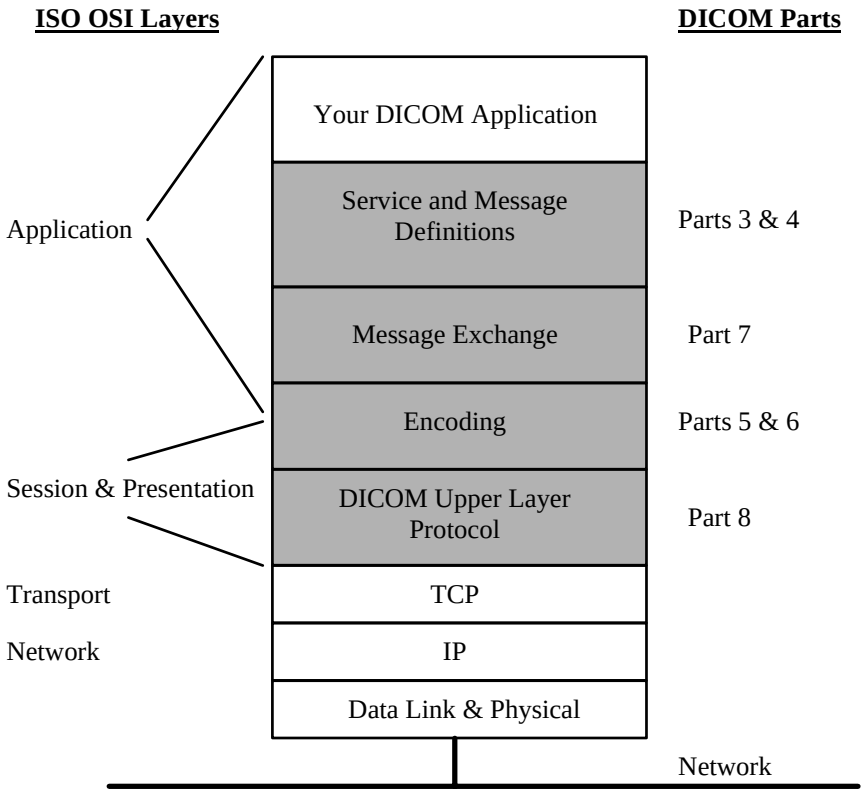

Figure 1: The DICOM Protocol Stack

Where to get the DICOM Standard

As a user of this Merge tool kit, you should have access to the DICOM Standard. MergeCOM-3 Advanced takes care of most of the details of DICOM for you. However, the standard is the final word. You will probably find Parts 2 – 6 most useful. The DICOM Standard can be ordered from:

NEMA
1300 N. 17th Street
Suite 1847
Rosslyn, VA 22209
USA
http://medical.nema.org

The DICOM Standard is typically published every other year. Each version includes approved changes since the last publishing. The most recent version was published in 2003. Besides purchasing a printed copy of the standard from NEMA,

it is also freely available in PDF format. The 2003 version of the standard can be downloaded from NEMA's public ftp site at this URL:

ftp://medical.nema.org/medical/Dicom/2003/printed/

Special Note!

Please note: The DICOM Standard is evolving so rapidly, that additions to the Standard are published as 'supplements'. For example, the media extensions have been incorporated into the DICOM Standard as a supplement that contains addenda to various parts of the standard (e.g., PS3.3, PS3.4, …). If you find that this document references a part of the Standard and you cannot find what you are looking for in that part, you probably need to get the proper supplement from NEMA. Other additions to the Standard (e.g., new image objects or documents) will also be published as supplements. NEMA also makes all supplements to the standard freely available on their ftp server. You can reference these supplements at this URL:

ftp://medical.nema.org/medical/dicom/Final/

The MergeCOM-3 Advanced Tool Kit

MergeCOM-3 Advanced provides a generalized implementation of DICOM in an ANSI-C Function Library that you can link with your application. You make simple function calls to open connections with other DICOM devices on a network, and to build and exchange DICOM messages or DICOM files.

Figure 2 presents a pictorial representation of a DICOM Application Entity; MergeCOM-3 Advanced implements for you everything in Parts 5, 6, 7, 8, and 10 of the DICOM Standard. It also makes it much easier for your application to implement according to Parts 3 and 4 by supplying many tools for the management of DICOM messages, and to Part 12 by supplying 'hooks' to your applications underlying file system.

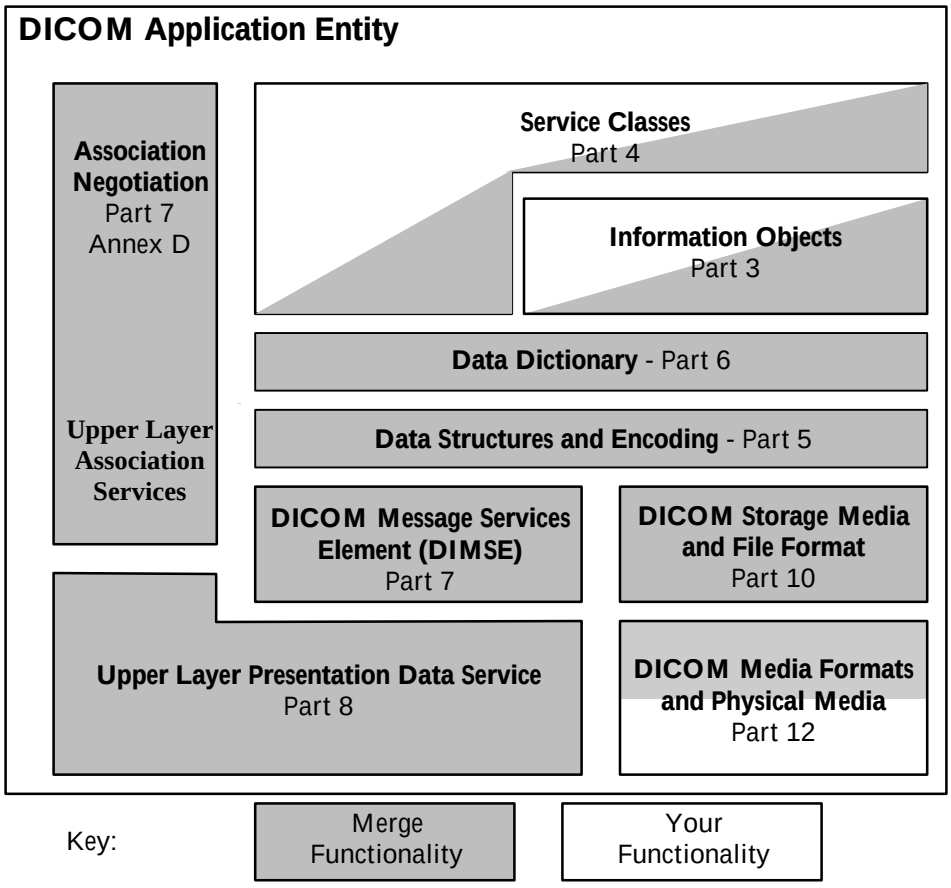

Figure 2: The DICOM Application Layer

The Advanced Tool Kit also supplies useful utility programs for testing a DICOM network connection, creating sample DICOM messages and writing them to a file, and for validating and listing the contents of DICOM messages.

Finally, sample application manuals (Image Transfer, Query/ Retrieve, Printing, Media Storage, Worklist Management, etc.) along with sample working source code give you valuable examples to work from when developing your own DICOM applications.

The DICOM Standard and Merge's Advanced Tool Kit allow applications to add private information to a DICOM message or file. For most application developers, this is more than sufficient. For applications that need to define their own non-standard private network or file services, an optional MergeCOM-3 Extended Tool Kit is available.

Development Platform Requirements

To use the MergeCOM-3 Advanced Tool Kit Library, you must run on a MergeCOM-3 supported computing platform. The advanced tool kit was designed to be portable and is available for many platforms (e.g., Sun Solaris 1.x, Sun Solaris 2.x, HP-UX, Linux, Windows 95/98/NT/2000/XP, SGI Irix, AIX, MacOS X). If it is not currently available for your target platform, please contact Merge eFilm. We may already be working on the port.

Once on a supported platform, you will need an ANSI-C compiler along with the Standard C Libraries. You will also need a Berkeley Sockets or WinSock (for

Windows 95/98/NT/2000/XP) Library for interfacing to TCP/IP and a linker to link your application with the libraries. In the case of the MacOS version of the tool kit no additional socket libraries are needed.

Your development environment (or at a minimum your target environment) should run on a machine with a network interface over which you can run the TCP/IP protocol. The advanced tool kit library supplies you with the DICOM protocol that runs on top of TCP/IP.

If your application will write DICOM files to interchangeable media, you will need a device driver for the media storage device and a programming interface between your operating system and the file system on that device.

Read your platform notes!

More specific requirements can be found in the Platform Notes pamphlet specific to a platform. This could include supported OS versions, supported compilers and linkers, and required compiler/build options.

Library Structure

Understanding the organization and components of the MergeCOM-3 Advanced Library is important to developing an efficient and capable DICOM application (see Figure 3). Following is a description of the header files that must be included within your application, and a description of the library's structure and the external components it uses at runtime to provide DICOM functionality.

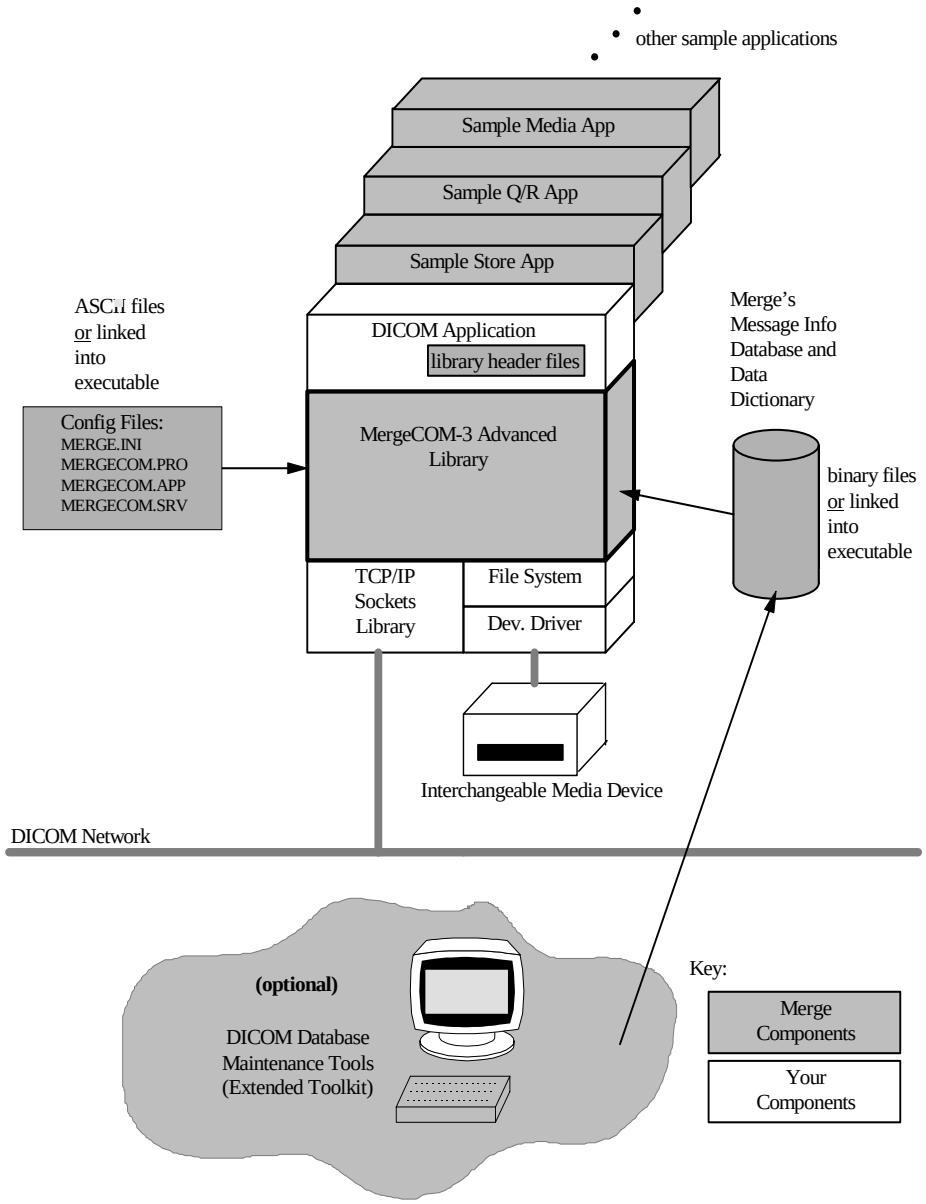

Figure 3: MergeCOM-3 Advanced Tool Kit Library Organization

Header Files

Your applications interface to the Advanced Library is described in five supplied header files:

- `mergecom.h`
- `mc3msg.h`
- `mc3media.h`
- `mcstatus.h`
- `diction.h`

The `mergecom.h` header file contains the prototypes of the functions used to register your application with the tool kit library and manage associations with

other DICOM AE's over the network. `Mc3msg.h` specifies the message object functions that allow you to populate or parse a message and supply you with powerful message validation features. `Mc3media.h` contains the functions used to create and maintain DICOM files and the DICOMDIR directory of a DICOM file set. `Mcstatus.h` specifies functions that allow your application to interpret the status codes returned from advanced tool kit calls. Finally, `diction.h` supplies useful `#defines` with descriptive names for all the DICOM Tags you might refer to in your applications.

Your application must `#include` these header files to make use of the appropriate tool kit library functionality.

MergeCOM-3 Advanced Library

The MergeCOM-3 Advanced Library (usually named `mc3adv.a` or `mc3adv.lib`) is the object code library for your computing platform that you link into your DICOM application. This library services your calls to the advanced tool kit. In the process of servicing networking calls, the advanced tool kit requires the services of a Berkeley Sockets (or WinSock) Library for your platform. If you are performing any DICOM network operations, this sockets library must also be linked with your application. Finally, if you are using the library to maintain a DICOM file set, you may need a special-purpose library to interface with your media storage device.

The MergeCOM-3 Advanced library has been carefully designed to be re-entrant and has been validated to be thread-safe on several multi-threading capable platforms. Also, shared libraries or dynamic link libraries (DLL's) are normally supplied for platforms which support them.

When a MergeCOM-3 Advanced Application is first run, it reads in its configuration files; usually named `merge.ini`, `mergecom.app`, `mergecom.pro`, and `mergecom.srv`. Tool Kit configuration is described later in this document and is detailed further in the Reference Manual. Usually, it is desirable to keep these configurable parameters in ASCII files for easy modification. When modifying your configuration files, your application must be re-run for those changes to take effect.

In cases where the tool kit configuration is unlikely to be changed or it is desirable to make these changes within the running application, the tool kit configuration can be compiled into your application. Most configurable parameters can be dynamically modified and reset within your running application.

Binary Message Information and Data Dictionary Files

A great deal of the power of MergeCOM-3 Advanced lies in its message handling and message validation capabilities. Message Objects are what is communicated between DICOM Application Entities. When your application creates a DICOM message object, the library accesses a binary message info file with information about that class of message. This info file describes to the library what attributes to expect as part of that message and each attribute's characteristics (Value Type, Conditions, and Enumerated or Defined Terms).

Another binary file containing the data dictionary is also accessed by the library. The data dictionary contains other characteristics of attributes (Name, Value Representation, and Value Multiplicity).

Performance Tuning

MergeCOM-3 Advanced gives you added flexibility, by not requiring your application to make use of the message info file. Certain API calls allow you to open messages without accessing the info files. This means that the tool kit cannot

validate your message against the DICOM standard, but this may not always be necessary once an application becomes stable. These options are discussed in greater detail in the Developing DICOM Applications section of this document.

Two specialized classes (subclasses) of message objects are also supported by the Advanced Library: items and files. Items are DICOM 'sub-messages' that can be stored in a DICOM message within a sequence of items. DICOM files are specialized DICOM messages that contain additional file meta-information and are written to or read from interchangeable media rather than transmitted or received over a network. Most MergeCOM-3 Advanced API calls dealing with message objects can also operate on items and files (these calls would be called polymorphic in object-oriented parlance). DICOM messages, items, and files will be described in much greater detail later in this document

Sample Applications

The sample applications and application guides can be a big help!

Included with the tool kit are sample applications in ANSI-C source code form and a `Makefile` that compiles and links the sample applications with the tool kit library. Sample client and server applications are supplied for Storage, Query/Retrieve, and Print services. A sample HIS/RIS application should be available soon (contact Merge eFilm for details). Also, a sample DICOM File Service application is provided.

Before writing your own applications, you should read the corresponding Sample Application Guides and look at the sample source. The guides also include a DICOM conformance statement for the example application. While these sample applications are primitive in features and user interface, they illustrate how to use the Advanced Tool Kit API to perform DICOM services over a network.

MergeCOM-3 Extended Tool Kit

Merge eFilm has a DICOM Database Management System in which the DICOM standard is maintained. This database, along with a few additional tools, is used to generate the binary message info and dictionary files accessed by the Advanced Tool Kit, or compiled into your application. As the DICOM standard is updated or extended, by simply maintaining this database, we can generate new binary files and keep the tool kit current. This also reduces the number of changes that must be made in the core Advanced Tool Kit library over time.

The Extended version of this tool kit makes some of these tools available to application developers who need to significantly extend the standard with private attribute and private service definitions. The extended version supplies you with an ASCII file database of the standard that you can extend, along with executables for your platform that translate these ASCII files to the binary message info and data dictionary files used by the tool kit at run time. In this way, you can extend the tool kit to validate your own private attributes and services.

Documentation Roadmap

The MergeCOM-3 Advanced documentation is structured as pictured in Figure 4.

Read Me FIRST!

The User's Manual is the foundation for all other documentation because it explains the concepts of DICOM and the Advanced Tool Kit. Before plunging into the Reference Manual or Sample Application Guide, you should be comfortable with the material in the User Manual.

The Reference Manual is where you go for detailed information on the Advanced Tool Kit. This includes the Application Programming Interface (API), tool kit

configuration, the runtime object database, and status logging. The Reference Manual also includes a DICOM conformance statement for the tool kit.

The DICOM Message Database Manual is an optional extension that describes the organization of the MergeCOM-3 DICOM Database and how to use it to extend standard services and define your own private services. Tools are supplied for converting the contents of the database into the binary runtime object database.

sample applications

The Sample Application Guide describes approaches to developing specific classes of DICOM applications (Image Transfer, Query/Retrieve, Print, HIS/RIS, Storage Media, etc.). It highlights pertinent information from Parts 3 or 4 of the DICOM Standard in a more readable way and in the context of the Advanced Tool Kit. The Application Guide also details the DICOM messages that can be passed between applications on the network. Also, a sample application is described and the application supplied in source form for your platform.

Platform-specific information required to use the Advanced tool kit on your target platform are specified in Platform Notes. This includes supported compilers, compiler options, link options, configuration, and run-time related issues.

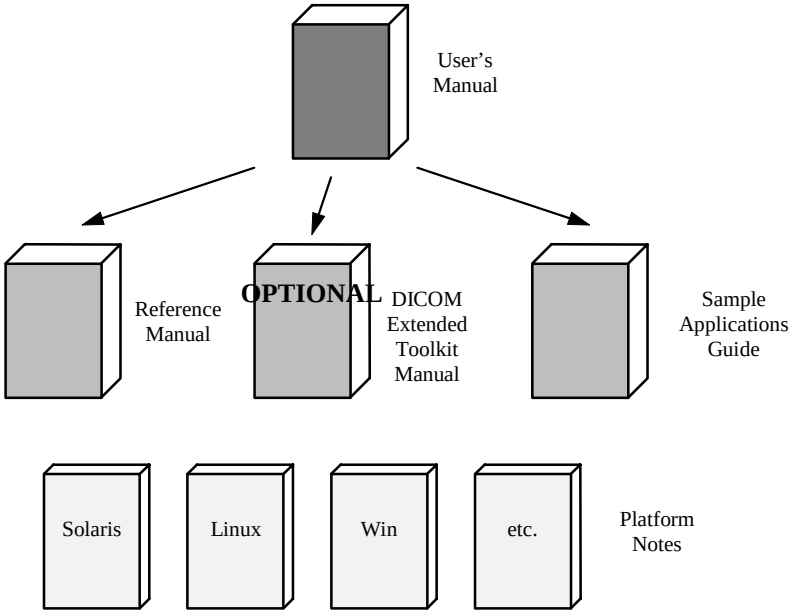

Figure 4: MergeCOM-3 Advanced Tool Kit Documentation Roadmap

Conventions

This manual follows a few formatting conventions.

Terms that are being defined are presented in **boldface**.

sample margin note

Margin notes (in the left margin) are used to highlight important points or sections of the document.

Sample commands appear in **`bold courier`** font, while sample output , source code, and function calls appear in `standard courier` font.

Hexadecimal numbers are written with a trailing H. For example 16 decimal is equivalent to 10H hexadecimal.

Performance Tuning

Portions of the document that can relate directly to the performance of your application are marked with the special margin note ***Performance Tuning***.

Understanding DICOM

The twelve separate parts of the DICOM Standard can seem overwhelming, and most would agree that they are difficult to read. Part of what makes a successful standard is precision and detail. Our goal here is to explain the key concepts without delving too far into the detail, most of which is handled automatically for you by the Advanced Tool Kit.

General Concepts

Some key concepts that must be understood to use the Advanced Tool Kit wisely are common across both DICOM networking and interchangeable media applications. These concepts are discussed first.

Application Entities

The DICOM Standard refers extensively to **Application Entities (AE's)**. An application entity is simply a DICOM application. If your application interacts with other applications on a network or with interchangeable media using the DICOM protocol, it is an application entity.

client/server

DICOM also refers to **Service Class Users (SCU's)** and **Service Class Providers (SCP's)**. An application entity is an SCU when it requests DICOM services over a network and an SCP when it provides DICOM services over a network. We will more often refer to the SCU as a **Client** and the SCP as a **Server.** A single DICOM application entity can act as both a client and a server. This client/server model is a powerful and omnipresent one in the world of distributed network computing.

Services and Meta Services

DICOM is formed around the concepts of **Services** and **Service Classes** (see Table 1). The DICOM Standard specifies a set of services that can be performed over a network. Some of the services can also be stored to interchangeable media (these are italicized in Table 1). As new services are introduced, the standard will be further expanded. The standard also groups related services into a service class. Table 1 lists the DICOM standard service classes and their component services.[2]

When a particular collection of services in a service class implies a higher level of service, this collection is combined by the standard into a **Meta Service**. Specifying that your application supports a specific meta service is a useful shorthand for explicitly listing out the collection of services that make up that meta service.

Service Class	Services	Description
Verification	Verification	Verifies application level communication between DICOM application entities (AE's).
Storage	*Computed Radiography Image Storage* *CT Image Storage*	Transfer of medical images and related standalone data between

[2] The DICOM Standard actually refers to services as Service Object Pairs (SOP's) and meta services as Meta-SOPs. We avoid this terminology to avoid unnecessary detail and confusion.

	CT Image Storage *Enhanced CT Image Storage* *MR Image Storage* *Enhanced MR Image Storage* *MR Spectroscopy Storage* *Spatial Registration Storage* *Spatial Fiducials Storage* *Hardcopy Color Image Storage* *Hardcopy Grayscale Image Storage* *Stored Print Storage* *Ultrasound Image Storage* *Ultrasound Multi-frame Image Storage* *Nuclear Medicine Image Storage* *Positron Emission Tomography Image Storage* *Positron Emission Tomography Curve Storage* *Digital X-Ray Image Storage – For Presentation* *Digital X-Ray Image Storage – For Processing* *Digital Intra-oral X-Ray Image Storage – For Presentation* *Digital Intra-oral X-Ray Image Storage – For Processing* *Digital Mammography Image Storage – For Presentation* *Digital Mammography Image Storage – For Processing* Raw Data Storage *RT Beams Treatment Record Storage* *RT Brachy Treatment Record Storage* *RT Dose Storage* *RT Plan Storage* *RT Image Storage* *RT Structure Set Storage* *RT Treatment Summary Record Storage* *VL Endoscopic Image Storage* *VL Microscopic Image Storage* *VL Photographic Image Storage* *VL Slide-Coordinates Microscopic Image Storage* *Video Endoscopic Image Storage* *Video Microscopic Image Storage* *Video Photographic Image Storage* *Xray Angio Image Storage* *Xray Angio Bi-plane Image Storage* *Xray Radio-Fluoro Storage* *Secondary Capture Image Storage* *Multi-frame Grayscale Byte Secondary Capture Image Storage* *Multi-frame Grayscale Word Secondary Capture Image storage* *Multi-frame Single Bit Secondary Capture Image Storage* *Multi-frame True Color Secondary Capture Image Storage* *Stand-alone Overlay Storage* *Stand-alone Curve Storage* *Stand-alone Modality LUT Storage* *Stand-alone VOI LUT Storage* *Basic Text Structured Reporting* *Comprehensive Structured Reporting* *Enhanced Structured Reporting* *Key Object Selection Document* *Chest CAD SR Storage* *Mammography CAD SR* *Procedure Log Storage* *Grayscale Softcopy Presentation State Storage* *12-lead ECG Waveform Storage*	related standalone data between DICOM application entities, either over a network or using interchangeable media.

	Ambulatory ECG Waveform Storage *Basic Voice Audio Waveform Storage* *Cardiac Electrophysiology Waveform Storage* *General ECG Waveform Storage* *Hemodynamic Waveform Storage* *Ophthalmic 8 bit Photography Image Storage* *Ophthalmic 16 bit Photography Image Storage* *Stereometric Relationship Storage*	
Storage Commitment	Storage Commitment Push Storage Commitment Pull	Ensures that SOP Instances stored with the storage service class will not be deleted after reception but will be stored safely and can be retrieved again at a later point.
Media Storage	DICOM Basic Directory Storage and storage of various (*italicized*) services from the other Service Classes	Exists as a member of every DICOM File Set and contains general information about the file set and a hierarchical directory of the DICOM files contained in the file set.
Query/Retrieve	Patient Root Find Patient Root Move Patient Root Get Study Root Find Study Root Move Study Root Get Patient/Study Only Find Patient/Study Only Move Patient/Study Only Get	Management of images through a query and retrieval mechanism based on a small number of key attributes.
Basic Worklist Management	Modality Worklist Find General Purpose Worklist Information Model – Find General Purpose Worklist Mgmt. Meta	Supports the exchange of any type of worklist from one AE to another.
Print Management	*Basic Film Session* *Basic Film Box* *Basic Grayscale Image Box* *Basic Color Image Box* Printer Printer Configuration Print Queue Management Pull Print Request Printer Referenced Image Box VOI LUT Box Presentation LUT Basic Annotation Box Basic Print Image Overlay Box SOP Class Print Job Image Overlay Retired **Basic Grayscale Print Mgmt. Meta** **Basic Color Print Mgmt. Meta** Pull Stored Print Mgmt. Meta **Ref. Grayscale Print Mgmt. Meta** **Ref. Color Print Mgmt. Meta**	Printing (or filming) of medical images and image related data on a hard copy medium. Also, storage of print related data to interchangeable media.
Study Content Notification	Basic Study Content Notification	Allows one DICOM AE to notify another DICOM AE of the existence, contents, and source location of the images of a study.
Application Event Logging	Procedural Event Logging	Defines an application-level class-of-service that facilitates the

Logging		network transfer of Event Log Records to be logged or recorded in a central location.
Patient Management	*Detached Patient Management* *Detached Visit Management* **Detached Patient Mgmt. Meta**	Creation and tracking of the subset of patient and patient visit information that is required to aid in the management of radiographic studies.
Study Management	*Detached Study Management* *Study Component Management* Modality Performed Procedure Step Modality Performed Procedure Step Notification Modality Performed Procedure Step Retrieve General Purpose Performed Procedure Step General Purpose Scheduled Procedure Step	Creation, scheduling, performance, and tracking of imaging studies.
Results Management	*Detached Results Management* *Detached Interpretation Management* **Detached Results Mgmt. Meta**	Creation and tracking of results and associated diagnostic interpretations.
Instance Availability Notification	Instance Availability Notification	An application-level class-of-service that allows one DICOM AE to notify another DICOM AE of the presence and availability of SOP Instances that may be retrieved.
Relevant Patient Information Query	General Relevant Patient Information Query Breast Imaging Relevant Patient Information Query Cardiac Relevant Patient Information Query	Defines an application-level class-of-service that facilitates the access to relevant patient information such as it is known at the time of query.
Hanging Protocol Storage	*Hanging Protocol Storage*	Allows one DICOM AE to send a Hanging Protocol SOP Instance to another DICOM AE.
Hanging Protocol Query/Retrieve	Hanging Protocol Information Model - FIND Hanging Protocol Information Model - MOVE	Provides query/retrieve capabilities for the Hanging Protocol Storage Service Class.
Media Creation Management	Media Creation Management	Defines a mechanism by which an SCU can instruct a device to create Interchange Media containing a set of composite SOP objects.

Table 1: DICOM Services Classes and their Component Services

Information Model

DICOM information model

The DICOM Standard includes the specification of a **DICOM Information Model**. A detailed entity-relationship diagram of this model is included in both parts 3 and 4 of the standard. This model specifies the relationship between the different types of objects (also called entities) managed in DICOM. For example, a Patient has one or more Studies, each of which are composed of one or more Series and zero or more Results, etc.

objects vs. object instances

Most of DICOM's services perform actions on or with **object instances.**[3] An **object** can be thought of as a class of data (CT Image, Film Box, etc.) while an object instance is an actual occurrence of an object (a particular CT Image, a populated Film Box, etc.).

normalized vs. composite

There are two types of objects (and hence, object instances) defined in DICOM. **Normalized objects** are objects consisting of a single entity in the DICOM information model (e.g., a Film Box). **Composite objects** are composed of several related entities (e.g., an MR Image). When possible, it is preferable to deal with normalized object instances over the network, because they contain less redundant data and can be more efficiently managed by an application.

Most services inherited from the ACR/NEMA Version 2.x Standard are **composite services** (operate on composite object instances) for reasons of backward compatibility. Newly introduced services, such as the HIS/RIS and Print Management Services, tend to be **normalized services** (operate on normalized object instances).

Networking

Certain aspects of DICOM only apply to networking when using the Advanced Tool Kit. This includes networking commands and association negotiation.

Commands

DICOM defines a set of networking **commands**.[4] Each service uses a subset of these DICOM commands to perform the service over a network. These commands usually act on object instances. The C-commands operate on composite object instances, while the N-commands operate on normalized object instances.

The DICOM commands and brief descriptions of their actions are listed in Table 2.

[3] object instances are referred to as SOP Instances or managed SOP's in the DICOM standard.

[4] commands are referred to as DIMSE Services in the DICOM Standard.

Where your application takes over...

DICOM Commands	Description
C-STORE	Transfer an object instance to a remote AE.
C-GET	Retrieve object instance(s) from a remote AE whose attributes match a specified set of attributes.
C-MOVE	Move object instance(s) from a remote AE whose attributes match a specified set of attributes to yet another remote AE (or possibly your own AE - which would be another form of retrieval).
C-FIND	Match a set of attributes to the attributes of a set of object instances on a remote AE.
C-ECHO	Verify end-to-end communications with a remote AE.
N-EVENT-REPORT	Report an event to a remote AE.
N-GET	Retrieve attribute values from a remote AE.
N-SET	Request modification of attribute on a remote AE.
N-ACTION	Request an action by a remote AE.
N-CREATE	Request that a remote AE create a new object instance.
N-DELETE	Request that a remote AE delete an existing object instance.

Table 2: DICOM Commands

These DICOM commands can be thought of as primitives that every networking service is built from. In the context of a particular Service, these primitive actions translate to explicit real-world activities on the part of an Application Entity. Hence, DICOM places requirements on an application implementing a DICOM service. DICOM is careful to only express high-level operational requirements, and leaves the creative detail and look and feel of the application entity to the developer.

request vs. response

For every command, there is both a **request** and a **response**. A command request indicates that a command should be performed and is usually sent to an SCP. A command response indicates whether a command completed or its state of completion and is usually returned to an SCU. Example request commands are C-STORE-RQ, N-GET-RQ, and N-SET-RQ. Example response commands are C-STORE-RSP, N-GET-RSP, and N-SET-RSP.

IMPORTANT!

It is important to note that this service definition level is where the MergeCOM-3 Advanced Tool Kit Library leaves off, and your Application begins. While MergeCOM-3 supplies sample application guides and running sample application source code for your platform, they are only supplied as an example. They clearly explain the requirements that implementing certain DICOM services places on your application and provide worthwhile but primitive examples of how to approach your application with the tool kit. While you will see that the tool kit saves you a great deal of 'DICOM work', it does not implement your end application for you.

Association Negotiation

One of the two areas where MergeCOM-3 Advanced does a great deal of the 'DICOM work' for you is in opening an association (session) with another DICOM AE over the network. DICOM application entities need to agree on certain things before they operate with one another (open an association); these include:

- the services that can be performed between the two devices, which also impacts the commands and object instances that can be exchanged.
- the **transfer syntax** that shall be used in the network communication. The transfer syntax defines how the commands and object instances are encoded 'on the wire'.

The exchange of DICOM commands and object instances can only occur over an open association.

DICOM defines an association negotiation protocol (see Figure 5). In the most common DICOM services, a client application entity (SCU) proposes an association with a server AE (SCP). However, some services define a mechanism where the client can be the SCP which opens an association with the SCU. This is used when an SCP sends asynchronous event reports to an SCU through the N-EVENT-REPORT command. This is done through DICOM role negotiation, which is used during standard association negotiation. For the sake of simplicity, the remainder of this manual refers to the client as the SCU and the server as the SCP.

The association request proposal contains the set of services the client would like to perform and the transfer syntaxes it understands. The server then responds to the client with a subset of the services and transfer syntaxes proposed by the client. If this subset is empty, the server has rejected the association. If the subset is not empty, the server has accepted the association and the agreed upon services may be performed.

The client is responsible for releasing the association when it is finished performing its network operations. Either the client or the server can abort the association in the case of some catastrophic failure (e.g., disk full, out of memory).

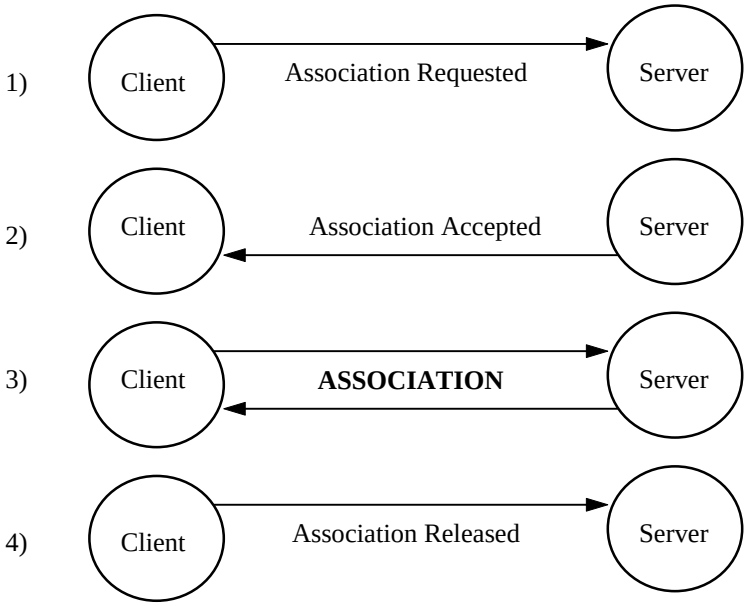

Figure 5: A Successful DICOM Association

Messages

service-command pair

Once an association is established, services are performed by AE's through the exchange of DICOM **Messages**. A message is the combination of a DICOM command request or response and its associated object instance (see Figure 6). Messages containing command requests will be referred to as **request messages**, while messages containing command responses will be referred to as **response messages**.

When a DICOM service is stored to interchangeable media in a DICOM File, the structure of a DICOM File is a slightly specialized class of DICOM message. Media interchange is discussed in detail later; the only important thing to realize for now is that much of what is discussed relating to DICOM Messages also applies to DICOM Files.

DICOM specifies the required message structure for each **service-command pair**. For example, the Patient Root Find - C-FIND-RQ service-command pair has a specific message structure. The command portion of a message is specified in Part 7 of the standard, while the object instance portion is specified in Parts 3 and 4.

attributes, values, and tags

A message is constructed of **attributes** having **values**, with each **attribute** identified by a **tag**. An attribute is a unit of data (e.g., Patient's Name, Scheduled Discharge Date, ...). A tag is a 4 byte number identifying an attribute (e.g., 00100010H for Patient's Name, 0038001CH for Scheduled Discharge Date, ...).

groups and elements

A tag is usually written as an ordered pair of two byte numbers. The first two bytes are sometimes called a **group** number, with the last two bytes being called an **element** number (e.g., (0010, 0010), (0038, 001C), ...). This terminology is partly a remnant of the ACR-NEMA Standard where elements within a group were related in some manner. This can no longer be depended on in DICOM, but the ordered pair notation is still useful and often easier to read.

Also, the ordered pair notation is important when defining a Tag for a private attribute. We will see later that all private attributes must have an odd group number.

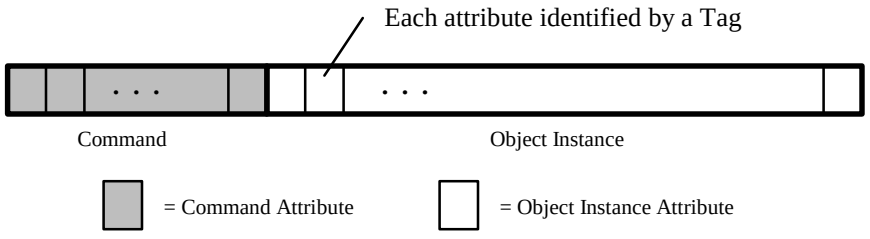

Figure 6: A DICOM Message

DICOM Data Dictionary

Attributes have certain characteristics that apply to them no matter what message they are used in. These characteristics are specified in the DICOM Data Dictionary (Part 6 of DICOM) and are **Value Representation (VR)** and **Value Multiplicity (VM)**.

Know your VR's!

Value Representation can be thought of as the 'type specifier' for the values that can be assigned to an attribute. This includes the data type, as well as its format.

The VR's defined by DICOM are listed in Table 3. You should refer to Part 5 of the standard for a detailed description of their legal values and formats.

VR	Name	VR	Name
AE	Application Entity	OF	Other Float String
AS	Age String	PN	Person Name
AT	Attribute Tag	SH	Short String
CS	Code String	SL	Signed Long
DA	Date	SQ	Sequence of Items
DS	Decimal String	SS	Signed Short
DT	Date Time	ST	Short Text
FL	Floating Point Single	TM	Time
FD	Floating Point Double	UI	Unique Identifier
IS	Integer String	UL	Unsigned Long
LO	Long String	UN	Unknown
LT	Long Text	US	Unsigned Short
OB	Other Byte String	UT	Unlimited Text
OW	Other Word String		

Table 3: DICOM Value Representations (VR's)

A single attribute can have multiple values. Value Multiplicity defines the number of values an attribute can have. VM can be specified as 1, *k* , 1-*k*, or 1-n; where *k* is some integer value and n represents 'many'. For example, Part 6 specifies the VM of Scheduled Discharge Time (0038, 001D) as 1, while the VM of Referenced Overlay Plane Groups (2040, 0011) is 1-99.

Message Handling

Given the number of services and commands specified in Tables 1 and 2, it is clear that there are a great deal of messages to manage in DICOM. Remember, each service-command pair implies a different message. Fortunately, you will see later that MergeCOM-3 Advanced saves the application developer a great deal of work in the message handling arena.

DICOM specifies the required contents of each message in Parts 3, 4, and 7 of the standard. For each attribute included in a message, additional characteristics of the attribute are defined that only apply within the context of a service. These characteristics are **Enumerated Values**, **Defined Terms,** and **Value Type**.

enumerated values vs. defined terms

DICOM specifies that some attributes should have values from a specified set of values. If the attribute is an enumerated value, it shall have a value taken from the specified set of values. A good example of enumerated values are (M, F, O) for Patient's Sex (0010, 0040) in Storage services. If the attribute is a defined term, it may take its value from the specified set, or the set may be extended with

additional values. An example of defined terms are (CREATED, RECORDED, TRANSCRIBED, APPROVED) for Interpretation Status ID (4008, 0212) in Results Management services. If this set is extended by an application with another term, such as IN PROCESS, it should be documented in that application's conformance statement.

value type (VT)

The most important characteristic of an attribute that is specified on a message by message basis, is the Value Type (VT). The VT of an attribute specifies whether or not that attribute needs to be included in a message and if it needs to have a value. Attributes can be required, optional, or only required under certain conditions (conditional attributes). Conditional attributes are always specified along with a condition. The value types defined by DICOM are listed in Table 4. Note that a null valued attribute has a value, that value being null (zero length).

Value Type (VT)	Description
1	The attribute must have a value and be included in the message. The value cannot be null (empty).
1C	The attribute must have a value and be included in the message only under a specified condition. The value cannot be null. If that condition is not met, the attribute shall not be included in the message.
2	The attribute must have a value and be included in the message. If the value for the attribute is unknown and cannot be specified, its value shall be null.
2C	The attribute must have a value and be included in the message only under a specified condition. If the value for the attribute is unknown and cannot be specified, its value shall be null. If that condition is not met, the attribute shall not be included in the message
3	The attribute is optional. It may or may not be included in the message. If included, the attribute may or may not have a null value.

Table 4: DICOM Value Types (VT's)

Private Attributes

Odd groups are private

The DICOM Standard allows application developers to add their own private attributes to a message as long as they are careful to follow certain rules. A **private attribute** is identified differently than are standard attributes. Its tag is composed of an odd group number, a private identification code string, and a single byte element number.

For example, ACME Imaging Inc. might define a private attribute to hold the name of the field engineer that last serviced their equipment. They could assign this attribute to private attribute tag (1455, 'ACME_IMG_INC', 00). This attribute has group number 1455, a private identification code string of 'ACME_IMG_INC', and a single byte element number of 00.

ACME could assign up 255 other private attributes to private group 1455 by using the other element numbers (01-FF). Part 5 of DICOM explains how these private tags are translated to standard group and element numbers and encoded into a message, while avoiding collisions. MergeCOM-3 Advanced handles these details for you.

DICOM makes a couple of rules that must be followed when using private attributes:

- Private attributes shall not be used in place of required (Value Type 1, 1C, 2, or 2C) attributes.
- The possible value representations (VRs) used for private attributes shall be only those specified by the standard (see Table 4).

The way you use private attributes in your application can also greatly affect your conformance statement. DICOM conformance is discussed in greater detail later.

Media Interchange

The DICOM Standard specifies a DICOM file format for the interchange of medical information on removable media. This file format is a logical extension of the networking portion of the standard. When an object instance that was communicated over a network would also be of value when communicated via removable media, DICOM specifies the encapsulation of these object instances in a DICOM file.

DICOM Files

DICOM File Structure

A **DICOM File** is the encapsulation of a DICOM object instance, along with **File Meta Information**. File meta information is stored in the header of every DICOM file and includes important identifying information about the encapsulated object instance and its encoding within the file (Figure 7).

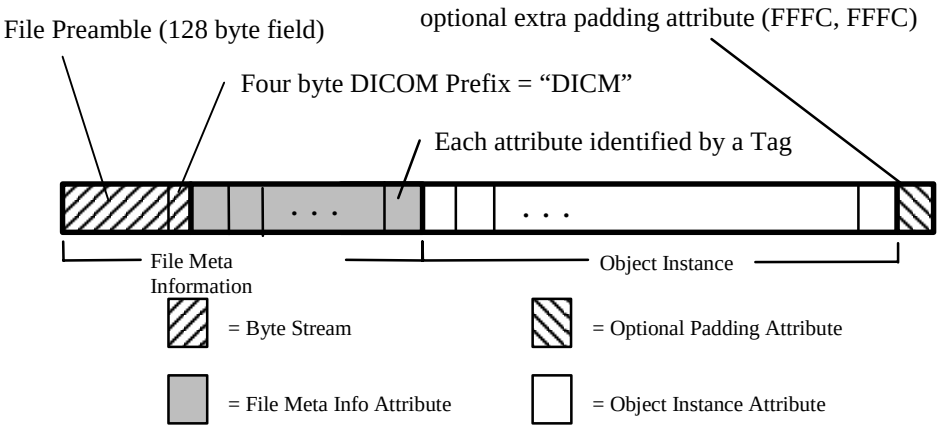

Figure 7: A DICOM File

The file meta information begins with a 128 byte buffer available for application profile or implementation specific use. **Application Profiles** standardize a number of choices related to a specific clinical need (modality or application) and are specified in Part 11 of the DICOM Standard. The next four bytes of the meta information contain the DICOM prefix, which is always "DICM" in a DICOM file and can be used as an identifying characteristic for all DICOM files. The remainder of the file (preamble and object instance) is encoded using tagged attributes (as in a DICOM Message).

The object instances that can be stored within the DICOM file are equivalent to a subset of the object instances that can be transmitted in network messages. The services that can be performed to interchangeable media are *italicized* in Table 1. The Media Storage Service Class (in Part 4 of the DICOM standard) specifies which service-command pairs can be performed to media. Remember it is the

service-command pair that identifies the object instance portion of the message, and it is only the object instance portion of the message that is stored in a DICOM file. The command attributes associated with a network message are never stored in a DICOM File.

DICOM objects that can be written to media

The service command pairs whose corresponding object instances can be stored to media are summarized in Table 5. Note that the Media Storage Directory Service is not performed over a network and the single object specified in the Basic Directory Information Object Definition (Part 3) is used.

Service	Command
Basic Text Structured Reporting	C-STORE
Comprehensive Structured Reporting	C-STORE
Computed Radiography Image Storage	C-STORE
CT Image Storage	C-STORE
Chest CAD SR Storage	C-STORE
Digital X-Ray Image Storage - For Presentation	C-STORE
Digital X-Ray Image Storage - For Processing	C-STORE
Digital Intra-oral X-Ray Image Storage - For Presentation	C-STORE
Digital Intra-oral X-Ray Image Storage - For Processing	C-STORE
Digital Mammography Image Storage - For Presentation	C-STORE
Digital Mammography Image Storage - For Processing	C-STORE
Enhanced CT Image Storage	C-STORE
Enhanced MR Image Storage	C-STORE
Enhanced Structured Reporting	C-STORE
Hardcopy Color Image Storage	C-STORE
Hardcopy Grayscale Image Storage	C-STORE
Key Object Selection Document	C-STORE
Mammography CAD SR	C-STORE
MR Image Storage	C-STORE
MR Spectroscopy Storage	C-STORE
Multi-frame Grayscale Byte Secondary Capture Image Storage	C-STORE
Multi-frame Grayscale Word Secondary Capture Image Storage	C-STORE
Multi-frame Single Bit Secondary Capture Image Storage	C-STORE

Multi-frame True Color Secondary Capture Image Storage	C-STORE
Nuclear Medicine Image Storage	C-STORE
Positron Emission Tomography Image Storage	C-STORE
Positron Emission Tomography Curve Storage	C-STORE
Procedural Log Storage	C-STORE
Raw Data Storage	C-STORE
RT Beams Treatment Record Storage	C-STORE
RT Brachy Treatment Record Storage	C-STORE
RT Dose Storage	C-STORE
RT Plan Storage	C-STORE
RT Image Storage	C-STORE
RT Structure Set Storage	C-STORE
RT Treatment Summary Record Storage	C-STORE
Secondary Capture Image Storage	C-STORE
Spatial Registration Storage	C-STORE
Spatial Fiducials Storage	C-STORE
Standalone Overlay Storage	C-STORE
Standalone Curve Storage	C-STORE
Standalone Modality LUT Storage	C-STORE
Standalone VOI LUT Storage	C-STORE
Stored Print Storage	C-STORE
Ultrasound Image Storage	C-STORE
Ultrasound Multi-frame Image Storage	C-STORE
Video Endoscopic Image Storage	C-STORE
Video Microscopic Image Storage	C-STORE
Video Photographic Image Storage	C-STORE
VL Endoscopic Image Storage	C-STORE
VL Microscopic Image Storage	C-STORE
VL Photographic Image Storage	C-STORE

VL Slide-Coordinates Microscopic Image Storage	C-STORE
X-Ray Angiographic Image Storage	C-STORE
X-Ray Angiographic Bi-plane Image Storage	C-STORE
X-Ray RadioFluoroscopic Image Storage	C-STORE
12-lead ECG Waveform Storage	C-STORE
Ambulatory ECG Waveform Storage	C-STORE
Basic Voice Audio Waveform Storage	C-STORE
Cardiac Electrophysiology Waveform Storage	C-STORE
General ECG Waveform Storage	C-STORE
Hemodynamic Waveform Storage	C-STORE
Hanging Protocol Storage	C-STORE
Ophthalmic 8 bit Photography Image Storage	C-STORE
Ophthalmic 16 bit Photography Image Storage	C-STORE
Stereometric Relationship Storage	C-STORE
Detached Patient Management	N-GET
Detached Visit Management	N-GET
Detached Study Management	N-GET
Detached Study Component Management	N-GET
Detached Results Management	N-GET
Detached Interpretation Management	N-GET
Basic Film Session	N-CREATE
Basic Film Box	N-CREATE
Basic Grayscale Image Box	N-SET
Basic Color Image Box	N-SET
Media Storage Directory Storage	C-STORE*

Table 5: Service-Command Pairs Specifying Object Instances that can be Stored in a DICOM File
** MergeCOM-3 Advanced defines a C-STORE command for the DICOMDIR service even though it does not formally exist in the DICOM Standard.*

Finally, the DICOM file can be padded at the end with the Data Set Trailing Padding attribute (FFFC, FFFC) whose value is specified by the standard to have no significance.

File Sets

DICOM Files must be stored on removable media in a **DICOM File Set.** A DICOM file set is defined as a collection of DICOM files sharing a common naming space within which file ID's are unique (e.g., a file system partition). A **DICOM File Set ID** is a string of up to 16 characters that provides a name for the file set.

A **File ID** is a name given to a DICOM file that is mapped to each media format specification (in Part 12 of DICOM). A file ID consists of an ordered sequence of one to eight components, where each component is a string of one to eight characters. One can certainly imagine mapping such a file ID to a hierarchical file system, and this is done for several media formats in Part 12. It is important to note that DICOM states that no semantic relationship between DICOM files shall be conveyed by the contents or structure of file ID's (e.g., the hierarchy). This helps insure that DICOM files can be stored in a media format and file system independent manner.

Naming DICOM File Sets and File ID's

The allowed characters in both a file ID's and file set ID's are a subset of the ASCII character set consisting of the uppercase characters (A-Z), the numerals (0-9), and the underscore (_).

The DICOMDIR

The **DICOM Directory File** or DICOMDIR is a special type of a DICOM File. A single DICOMDIR must exist within each DICOM file set, and is always given the file ID "DICOMDIR". It is the DICOMDIR file that contains identifying information about the entire file set, and usually (dependent on the Application Profile) a directory of the file set's contents.

Figure 8 shows a graphical representation of a DICOMDIR file and its central role within a DICOM File Set.

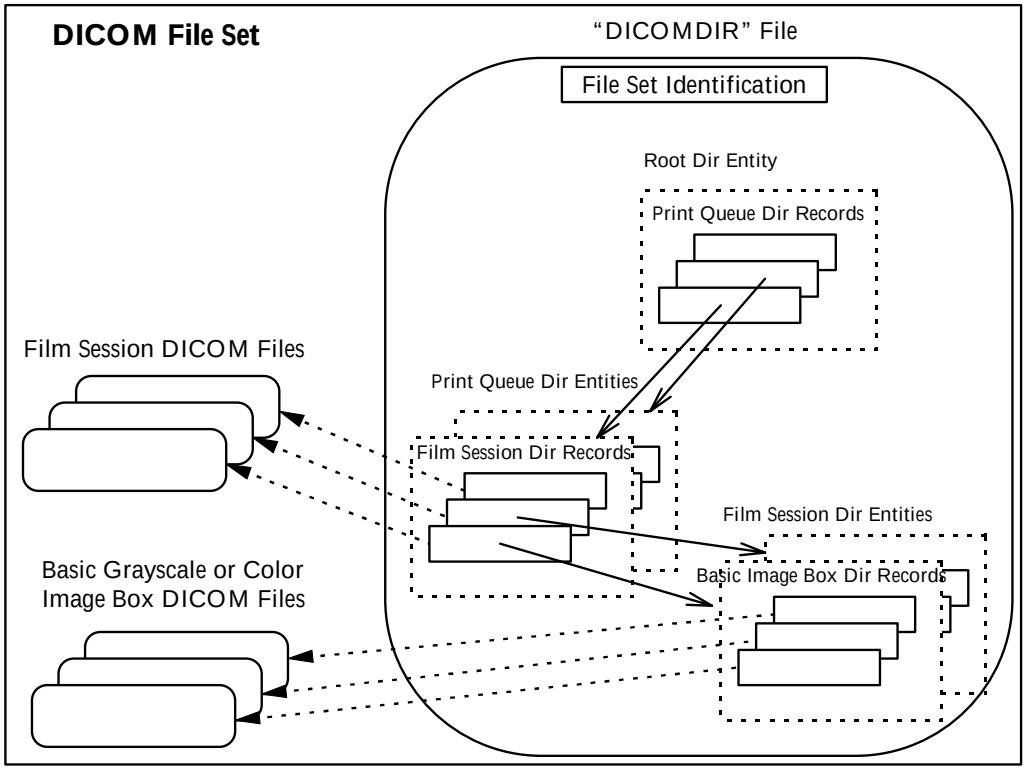

Figure 8: A DICOM Directory File (DICOMDIR) within a DICOM File Set

The DICOMDIR hierarchy

If the DICOMDIR file contains directory information, it is composed of a hierarchy of directory entities, with the top-most directory entity being the root directory entity. A **Directory Entity** is a grouping of semantically related directory records. A **Directory Record** identifies a DICOM File by summarizing key attributes and their values in the file and specifying the file ID of the corresponding file. The file ID can then be used, in the context of the native file system, to access the corresponding DICOM file. Each directory record can in turn point down the hierarchy to a semantically related directory entity.

Part 3 of the DICOM Standard specifies the allowed relationships between directory records in the section defining the Basic Directory IOD. We reproduce this table here (see Table 6) for pedagogical reasons; but, you should refer to the DICOM Standard for the most up-to-date and accurate specification.

Directory Record Type	Record Types which may be included in the next lower-level Directory Entity
(Root Directory Entity) *	PATIENT, TOPIC, PRINT QUEUE, HANGING PROTOCOL, PRIVATE
PATIENT	STUDY, PRIVATE
STUDY	SERIES, VISIT, RESULTS, STUDY COMPONENT, FILM SESSION, PRIVATE
SERIES	IMAGE, STORED PRINT, RT DOSE, RT STRUCTURE SET, RT PLAN, RT TREAT RECORD, OVERLAY, MODALITY LUT, VOI LUT, CURVE, SR DOCUMENT, PRESENTATION, KEY OBJECT DOC, SPECTROSCOPY, RAW DATA, WAVEFORM, REGISTRATION, FIDUCIAL, PRIVATE
HANGING PROTOCOL	PRIVATE
IMAGE	PRIVATE
STORED PRINT	PRIVATE
RT DOSE	PRIVATE
RT STRUCTURE SET	PRIVATE
RT PLAN	PRIVATE
RT TREAT RECORD	PRIVATE
OVERLAY	PRIVATE
MODALITY LUT	PRIVATE
VOI LUT	PRIVATE
CURVE	PRIVATE
SR DOCUMENT	PRIVATE
PRESENTATION	PRIVATE
KEY OBJECT DOC	PRIVATE

SPECTROSCOPY	PRIVATE
RAW DATA	PRIVATE
WAVEFORM	PRIVATE
REGISTRATION	PRIVATE
FIDUCIAL	PRIVATE
TOPIC	STUDY, SERIES, IMAGE, OVERLAY, MODALITY LUT, VOI LUT, CURVE, FILM SESSION, PRIVATE
VISIT	PRIVATE
RESULTS	INTERPRETATION, PRIVATE
INTERPRETATION	PRIVATE
STUDY COMPONENT	PRIVATE
PRINT QUEUE	FILM SESSION, PRIVATE
FILM SESSION	FILM BOX, PRIVATE
FILM BOX	BASIC IMAGE BOX, PRIVATE
BASIC IMAGE BOX	PRIVATE
PRIVATE	PRIVATE
MRDR	(Not applicable)

Table 6: Allowed Directory Entity

***** The first row of this table specifies the directory records that can be contained within the Root Directory Entity.

File Management Roles and Services

File Management Services

Part 10 of the DICOM Standard specifies a set of file management roles and services. There are five **DICOM File Services**, that describe the entire set of DICOM file operation primitives:

DICOM File Services	Description
M-WRITE	Create new files in a file set and assign them a file ID.
M-READ	Read existing files based on their file ID.
M-DELETE	Delete existing files based on their file ID.
M-INQUIRE FILE-SET	Inquire free space available for creating new files within a file set.

M-INQUIRE FILE	Inquire date and time of file creation (or last update if applicable) for any file within a file set.

Table 7: DICOM File Services

The MergeCOM-3 Advanced Tool Kit supplies families of functions that perform the first two (underlined) file services. The Tool Kit also implements enhanced read and write functionality for the creation and maintenance of DICOMDIR files and its hierarchy of directory entities and directory records. The remaining three file services are best implemented by the application entity through file system calls because they are file system dependent operations.

File Management Roles

DICOM AE's that perform file interchange functionality are in turn classified into three roles:

File Set Creator (FSC) — uses M-WRITE operations to create a DICOMDIR file and one or more DICOM files.

File Set Reader (FSR) — uses M-READ operations to access one or more files in a DICOM file set. An FSR shall not modify any files of the file set (including the DICOMDIR file).

File Set Updater (FSU) — performs M-READ, M-WRITE, and M-DELETE operations. It reads, but shall not modify the content of any DICOM files other than the DICOMDIR file. It may create additional files by means of an M-WRITE or delete existing files by means of an M-DELETE.

The concept of these roles is used within the DICOM conformance statement of an application entity that supports media interchange to more precisely express the capabilities of the implementation. Conforming applications shall support one of the capability sets specified in Table 8. DICOM conformance is described in greater detail in the next section.

	Media Roles				
Media Roles	**M-WRITE**	**M-READ**	**M-DELETE**	**M-INQUIRE FILE-SET**	**M-INQUIRE FILE**
FSC	Mandatory	*not required*	*not required*	Mandatory	Mandatory
FSR	*not required*	Mandatory	*not required*	*not required*	Mandatory
FSC+FSR	Mandatory	Mandatory	*not required*	Mandatory	Mandatory
FSU	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory
FSU+FSC	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory
FSU+FSR	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory
FSU+FSC+FSR	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory

Table 8: Media Application Operations and Roles

Conformance

Part 2 of DICOM discusses conformance and is important to any AE developer. For an application to be DICOM conformant it must:

- meet the minimum general conformance requirements specified in Part 2 and service specific conformance requirements specified in Part 4 (Network Services), and/or Parts 10 and 11 (Media Services).
- have a published DICOM conformance statement detailing the above conformance and any optional extensions.

Conformance also applies to aspects of the communications protocol that are managed by the Advanced Tool Kit. Most parameters are configurable by your application. The conformance statement for the MergeCOM-3 Advanced Tool Kit in the Reference Manual lists all these protocol parameters and how they can be configured.

Conformance Statement templates in each of the Sample Application Guides also provide guidance in preparing your conformance statement for your application.

Part 2 also deals with private extensions to the DICOM Standard by defining **Standard Extended Services**. Standard Extended Services give your application a little more flexibility, by allowing you to add private attributes as long as they are of value type 3 (optional) and are documented in the conformance statement.

DICOM also allows you to define your own **Specialized** and **Private Services**. These should be avoided by most applications since they are non-standard, add complexity to your application, and limit interoperability.

If you are significantly extending services or creating your own private services, you may need the MergeCOM-3 Advanced Extended Toolkit to assist in defining these services so that they can be supported by the tool kit.

Using MergeCOM-3 Advanced

You can use the MergeCOM-3 Advanced tool kit 'out of the box' by using its supplied utility programs and sample applications. In this section we discuss how to configure the tool kit and to use the utility programs. Use of the sample applications are described in the sample application guides. Later, we discuss how to develop your own DICOM applications using the MergeCOM-3 Advanced library.

Configuration

MergeCOM-3 Advanced is highly configurable, and understanding its configuration files is critical to using the library effectively.

Related parameters are grouped into sections in a configuration file as follows:

```
[SECTION_1]
   PARAMETER_1 = value1
   PARAMETER_2 = value2

[SECTION_2]
   PARAMETER_3 = value3

         .
         .
         .
```

Related sections are grouped into one of four configuration files: an initialization file, an application profile, a system profile, and a service profile. Each of these configuration files are discussed separately below. Only the key configurable parameters are summarized in this document. See the Reference Manual for detailed descriptions of all configuration files and their parameters**.**

Initialization File

The MergeCOM-3 Initialization File (usually called `merge.ini`) provides the advanced tool kit with its top-level configuration. It specifies the location of the other three configuration files, along with message and error logging characteristics.

MERGE_INI environmental variable

There are two options to access the merge.ini file for your runtime environment. The function `MC_Set_MergeINI( )` can be used to assign the path where the merge.ini file is located. You can also set the `MERGE_INI` environmental variable to point to the Merge Initialization File. This variable can be set within a command shell; for example:

In Unix C-shell:
```
setenv MERGE_INI /users/mc3adv/merge.ini
```

In Unix Bourne, Korn, or Bash shell:
```
MERGE_INI=/users/mc3adv/merge.ini; export MERGE_INI
```

In DOS command shell:
```
set MERGE_INI=\mc3adv\merge.ini
```

See the Platform notes for your platform if none of these methods apply.

The initialization file contains one `[MergeCOM3]` section that points to the location of the other three MergeCOM-3 initialization files, specifies characteristics of the message/error log kept by the advanced library, turns particular types of logging on and off, and specifies where the messages are logged (file, screen, both, or neither). In most cases the INFO, WARNING, and ERROR messages will be sufficient. The T*n*_MESSAGE settings (where *n* is an integer between 1 and 9) turns on lower-level protocol tracing capabilities. These capabilities can prove useful when running into difficulties communicating with other implementations of DICOM over a network and can be used by Merge service engineers in diagnosing lower-level network problems.

Application Profile

The MergeCOM-3 Application Profile (usually called `mergecom.app`) specifies the characteristics of your own application entity and the AE's your application will connect with over a network. The name and location of this file is specified in the `[MergeCOM3]` section of the MergeCOM-3 initialization file.

DICOM AE Title

When your application acts as a client (SCU), you must specify in the application profile the network address of the server (SCP) Application Entities you wish to connect (open an association) with. Your client refers to the application entity by a **DICOM Application Entity Title** and this is the same way it is referred to in the application profile. The AE title consists of a string of characters containing no spaces and having a length of 16 characters or less. A section of the profile exists for each Server AE you wish to connect with.

For example, if your application is an image source and also performs query and retrieval of images from two separate DICOM AE's, it might contain sections like the following:

```
[Acme_Store_SCP]
   PORT_NUMBER     = 104
   HOST_NAME       = acme_sun1
   SERVICE_LIST    = Storage_Service_List

[Acme_QR_SCP]
   PORT_NUMBER     = 104
   HOST_NAME       = acme_hp2
   SERVICE_LIST    = Query_Service_List
```

Acme_Store_SCP and Acme_QR_SCP are the AE titles for the applications you wish to connect with. The storage server runs on a Sun computer having the host name acme_sun1, while the query/retrieve server runs on an HP workstation with the host name acme_hp2. Both servers listen on port 104 (the standard DICOM listen port). The host name and port combined make up the TCP/IP network address for a listening server application. See Figure 9.

Service List

The SERVICE_LIST is set to the name of another section in the application profile that lists the DICOM services that will be negotiated with that application entity. For example, in this case these sections might look like:

```
[Storage_Service_List]
   SERVICES_SUPPORTED = 11 # Number of Services in list
   SERVICE_1               = STANDARD_MR
   SERVICE_2               = STANDARD_CR
   SERVICE_3               = STANDARD_CT
   SERVICE_4               = STANDARD_CURVE
   SERVICE_5               = STANDARD_MODALITY_LUT
   SERVICE_6               = STANDARD_OVERLAY
   SERVICE_7               = STANDARD_SEC_CAPTURE
   SERVICE_8               = STANDARD_US
   SERVICE_9               = STANDARD_US_MF
   SERVICE_10              = STANDARD_VOI_LUT
   SERVICE_11              = STANDARD_NM

[Query_Service_List]
   SERVICES_SUPPORTED = 2  # Number of Services in list
   SERVICE_1               = STUDY_ROOT_FIND
   SERVICE_2               = STUDY_ROOT_MOVE
```

[Storage_Service_List] lists the storage services that will be requested, while [Query_Service_List] lists the type of query/retrieve that will be requested. These service names are the strings used in MergeCOM-3 Advanced to identify standard DICOM services and are discussed further in specific application guides. Any services listed must be defined in the service profile, discussed below.

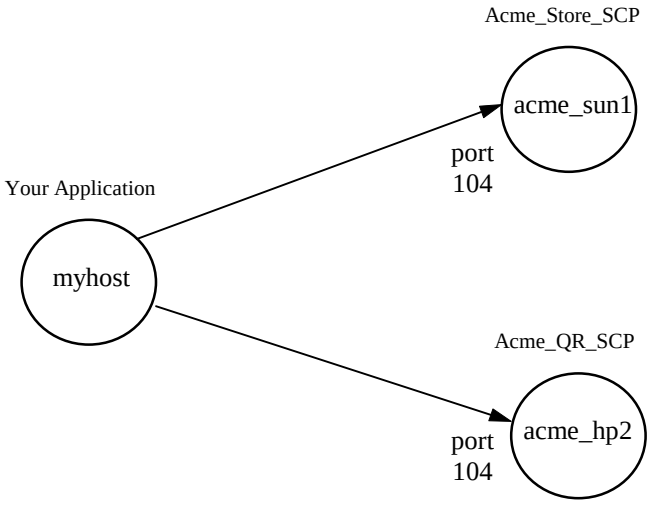

Figure 9: An example configuration of DICOM applications

Don't forget!

A service list also needs to be defined for each of your own server AE's. Even though you do not need a section for your server AE Title (since it is running on your local machine), you do need to specify a service list that your application supports as an SCP. If your application also acts as a storage server, for example, it could use [Storage_Service_List]. You also need to specify a listen port for your server AE in the System Profile, which is discussed below.

Transfer Syntax List

For advanced users, MergeCOM-3 allows for the defining of the transfer syntaxes supported for each service in a service list. This functionality is implemented through the use of transfer syntax lists. The basic service lists discussed above can be modified to include these transfer syntax lists. The following is an example service list that has transfer syntaxes specified for each service:

```
[Storage_Service_List]
    SERVICES_SUPPORTED   = 3  # Number of Services
    SERVICE_1            = STANDARD_MR
    SYNTAX_LIST_1        = MR_Syntax_List
    SERVICE_2            = STANDARD_US
    SYNTAX_LIST_2        = US_Syntax_List
    SERVICE_3            = STANDARD_CT
    SYNTAX_LIST_3        = CT_Syntax_List

[MR_Syntax_List]
    SYNTAXES_SUPPORTED   = 4  # Number of Syntaxes
    SYNTAX_1             = JPEG_BASELINE
    SYNTAX_2             = EXPLICIT_BIG_ENDIAN
    SYNTAX_3             = EXPLICIT_LITTLE_ENDIAN
    SYNTAX_4             = IMPLICIT_LITTLE_ENDIAN

[US_Syntax_List]
    SYNTAXES_SUPPORTED   = 2  # Number of Syntaxes
    SYNTAX_1             = RLE
    SYNTAX_2             = IMPLICIT_LITTLE_ENDIAN

[CT_Syntax_List]
    SYNTAXES_SUPPORTED   = 2  # Number of Syntaxes
    SYNTAX_1             = EXPLICIT_LITTLE_ENDIAN
    SYNTAX_2             = IMPLICIT_LITTLE_ENDIAN
```

[Storage_Service_List] lists some standard storage service class services used by MergeCOM-3. The SYNTAX_LIST_N parameter has been added to this example to specify a transfer syntax list for each service. This optional parameter is set to the name of another section in the application profile which lists a group of DICOM transfer syntaxes to be negotiated. When this parameter is not set, the default non-compressed transfers syntaxes (implicit VR little endian, explicit VR little endian, and explicit VR big endian) are negotiated.

The [MR_Syntax_List], [US_Syntax_List], and [CT_Syntax_List] sections each define a separate transfer syntax list for the MR, US, and CT services respectively. MergeCOM-3 Advanced currently supports all transfer syntaxes specified in the DICOM standard. The names used for these transfer syntaxes are defined in Appendix B of the *MergeCOM-3 Advanced Integrator's Tool Kit Reference Manual*.

Transfer syntax priority during association negotiation

For server (SCP) applications, the order in which transfer syntaxes are specified in a transfer syntax list dictates the priority MergeCOM-3 places on them during association negotiation. For example, in the [US_Syntax_List] specified above, if a client (SCU) proposed the Ultrasound storage service with the RLE compressed transfer syntax and the implicit VR little endian transfer syntax, MergeCOM-3 would select the RLE transfer syntax because it was listed first in the transfer syntax list.

When a transfer syntax list is not specified in a service list the priority MergeCOM-3 Advanced places on transfer syntaxes during association negotiation is dependent on the hardware platform. On little endian machines (Intel and DEC Alpha based systems) the priority order is: Explicit VR Little Endian, Implicit VR Little Endian, and Explicit VR Big Endian. On big endian machines the priority order is: Explicit VR Big Endian, Explicit VR Little Endian, and Implicit VR Little Endian.

Role negotiation

MergeCOM-3 also supports DICOM role negotiation through its service lists. Whereas in previous examples, the same service list could be used for both client (SCU) and server (SCP), these service lists are specific to the role to be negotiated for each service.

```
[SCU_Service_List]
    SERVICES_SUPPORTED   = 1  # Number of Services
    SERVICE_1            = STORAGE_COMMITMENT_PUSH
    ROLE_1               = SCU

[SCP_Service_List]
    SERVICES_SUPPORTED   = 1  # Number of Services
    SERVICE_1            = STORAGE_COMMITMENT_PUSH
    ROLE_1               = SCP
```

In this case, the [SCU_Service_List] supports the Storage Commitment Push SOP class as an SCU and the [SCP_Service_List] supports the Storage Commitment Push SOP class as an SCP. MergeCOM-3 will negotiate the association based on the settings for these roles.

The role for a service can be defined as SCU, SCP, BOTH, or be undefined. Table 9 contains a complete listing of configurable roles for both requestors and acceptors along with the resultant negotiated roles. Note that in some cases a service will be rejected because the roles being negotiated do not match.

Requestor's Configured Role	Acceptor's Configured Role	Requestor's Negotiated Role	Acceptor's Negotiated Role

SCU	SCP	SCU	SCP
	SCU	Rejected	Rejected
	BOTH	SCU	SCP
	NOT DEFINED	SCU	SCP
SCP	SCP	Rejected	Rejected
	SCU	SCP	SCU
	BOTH	SCP	SCU
	NOT DEFINED	Rejected	Rejected
BOTH	SCP	SCU	SCP
	SCU	SCP	SCU
	BOTH	BOTH	BOTH
	NOT DEFINED	SCP	SCP
NOT DEFINED	SCP	SCU	SCP
	SCU	Rejected	Rejected
	BOTH	SCU	SCP
	NOT DEFINED	SCU	SCP

Table 9: Negotiated Roles

Configuring Asynchronous Communications Support

MergeCOM-3 also optionally supports DICOM Asynchronous Operations Window Negotiation through service lists. The same service list can be used in this case for both the client (SCU) and server (SCP). The following is an example service list that configures DICOM asynchronous communication negotiation:

```
[SCU_Or_SCP_Service_List]
    SERVICES_SUPPORTED   = 1  # Number of Services
    MAX_OPERATIONS_INVOKED = 10
    MAX_OPERATIONS_PERFORMED = 10
    SERVICE_1            = STANDARD_MR
```

In this case, the `[SCU_Or_SCP_Service_List]` supports the Standard MR SOP Class. For all services, it supports 10 maximum operations invoked and 10 maximum operations performed. When `MAX_OPERATIONS_INVOKED` and `MAX_OPERATIONS_PERFORMED` are not included in the service list, asynchronous communications are not negotiated. See a subsequent section for details on implementing DICOM asynchronous communications with MergeCOM-3.

Configuring Extended Negotiation for Clients

MergeCOM-3 optionally supports configuration of DICOM Extended Negotiation information in service lists. Currently, the DICOM standard allows extended negotiation information for the Storage and Query/Retrieve Service classes as defined in PS3.4 of the standard. The extended negotiation information can be set for only the client (SCU). Server applications utilizing extended negotiation must set this information with a call to `MC_Set_Negotiation_Info_For_Association`.

```
[SCU_Service_List]
    SERVICES_SUPPORTED   = 2  # Number of Services
    SERVICE_1            = STUDY_ROOT_QR_FIND
    EXT_NEG_INFO_1       = 0x01
    SERVICE_2            = STUDY_ROOT_QR_MOVE
    EXT_NEG_INFO_2       = 0x01
```

In this case, the [SCU_Service_List] supports the Study Root Q/R Find and Move services. Both services have set a single byte of extended negotiation information set to hexadecimal 0x01. (In this case, this implies the Client supports relational Queries and Moves.) Multiple hexadecimal bytes can be set in the service list by listing each byte in the format "0x00 0x01 0x02".

System Profile

The MergeCOM-3 System Profile (usually called mergecom.pro) contains configuration parameters for the Advanced Library itself. The name and location of this file are specified in the [MergeCOM3] section of the MergeCOM-3 initialization file.

Many of these parameters should never need to be modified by the user, including low-level protocol settings such as time-outs. Only the parameters that should be understood by every user of the tool kit are discussed here; see the Reference Manual for a discussion of all parameters.

Nothing works without the license number

Most importantly, you must place the license number you received when you purchased the tool kit in the [ASSOC_PARMS]section of the system profile. If the license you received with your tool kit was DA53-31D34 you would need to set it in the [ASSOC_PARMS]section as follows:

```
[ASSOC_PARMS]
   LICENSE  = DA53-31D34
   IMPLEMENTATION_CLASS_UID = 2.16.840.1.113669.2.1.2
   IMPLEMENTATION_VERSION   = MergeCOM3_340
   ACCEPT_MULTIPLE_PRES_CONTEXTS  = Yes
```

The tool kit sample applications, and your own applications that use the Advanced Library will not work without a valid license number.

The above example of the [ASSOC_PARMS] section of the system profile also contains example implementation class UID and implementation version configuration values. The implementation class UID is intended by the DICOM standard to be unique for major revisions of an application entity. The implementation version is intended to be unique for the minor revisions of an application entity. These configuration values are used during association negotiation by MergeCOM-3 and are intended to aid in tracking versions of applications in the field.

The ACCEPT_MULTIPLE_PRES_CONTEXTS configuration value is used by server (SCP) applications. This value determines if multiple presentation contexts can be negotiated for a single DICOM Service. This option is discussed below.

As mentioned earlier, a listen port must be identified for your server AE. Port 104 is the standard DICOM listen port. This, along with the number of simultaneous connections you wish your server to support, is specified in the [TRANSPORT_PARMS] section. MAX_PENDING_CONNECTIONS specifies the number of connections that can be queued before they are serviced.

```
[TRANSPORT_PARMS]
      TCPIP_LISTEN_PORT                    = 104
```

```
# Max number of open listen channels
MAX_PENDING_CONNECTIONS                = 5
```

An important section of the System Profile is the [MESSAGE_PARMS]section:

```
[MESSAGE_PARMS]
   LARGE_DATA_STORE        = FILE #  | MEM Default = FILE
   LARGE_DATA_SIZE         = 200
   OBOW_BUFFER_SIZE        = 4096
   DICTIONARY_ACCESS       = MEM #  | FILE Default = FILE
   DICTIONARY_FILE         = /users/mc3adv/mrgcom3.dct
   TEMP_FILE_DIRECTORY     = /users/mc3adv/tmp_files/
   MSG_INFO_FILE           = /users/mc3adv/mrgcom3.msg
```

Dealing with large data

The LARGE_DATA_STORE parameter informs the tool kit where it should store large data: either in memory, or in temporary files on disk. Large data is defined as a value for an attribute larger than LARGE_DATA_SIZE bytes. Pixel data associated with a medical image would most certainly be considered large data.

If you are running your process on a resource rich system that supplies plenty of physical and virtual memory, you should select LARGE_DATA_STORE = MEM to improve your performance. If your process is not so fortunate or you are dealing with messages with very large data values, you will want to use LARGE_DATA_STORE = FILE. In this case, the Advanced Tool Kit will manage the large data in temporary files located in the TEMP_FILE_DIRECTORY you specify.

callbacks

Large data that is of value representation OB (Other Byte) , OW (Other Word), or OF (Other Float) is treated specially by the tool kit. Pixel Data, Curves, and Overlays are composed of this type of data. You can let the tool kit manage OB/OW data for you like any other large data, or register your own Callback Function in your applications to deal with such data as it is being received or transmitted over the network. The use of Callbacks will be covered later when we discuss developing DICOM applications with the tool kit.

Performance Tuning

The OBOW_BUFFER_SIZE is used to tell the tool kit what size 'chunks' in bytes of OB/OW data it should read in before either writing the data to a temporary file or passing it to your Callback Function. Choosing a large number for OBOW_BUFFER_SIZE means less time spent by your application process writing to temporary files or making callbacks, but results in a larger process size. If you need to use temporary files or callbacks, you should tune this parameter to maximize performance within the constraints of your runtime environment.

dictionary file

Performance Tuning

The DICTIONARY_ACCESS parameter specifies whether the DICOM Data Dictionary should be loaded into memory the first time it is accessed by an application and kept there, or be accessed from a file every time it is referenced. The **dictionary file** is a binary file supplied with your tool kit with the default name of mrgcom3.dct. This file is accessed very frequently by the tool kit and you should set this parameter to MEM in all but the most extreme cases of memory limitation. You also specify the location and name of the data dictionary file using the DICTIONARY_FILE parameter.

message info files

Another binary file supplied with the tool kit is the **message info file**. This file contains binary encoded message objects and is accessed when an application opens a message. Once open, these objects reside in memory, are 'filled in' by your application, and become a message object instance that can be exchanged over the network. The message info file, along with the data dictionary file, also make possible the powerful message validation capabilities of the advanced tool kit. The message info file is a binary file supplied with your tool kit with the default name

of `mrgcom3.msg`. You also specify the location and name of the message info file using the `MSG_INFO_FILE` parameter.

capturing network data

It is often useful to capture the raw data that is transmitted across the network to help determine exactly what each side of an association is sending. Network "sniffer" programs are often used to capture this data, but they are often not useful when the data is being transmitted over a secure network connection, as the data is often encrypted. MergeCOM-3 provides a network capture facility that will capture non-encrypted outbound network data before sending it a secure socket handler, and capture decrypted inbound data that it receives from a secure socket handler. The data that is captured is formatted such that it can be analyzed using the MergeDPM© utility.

Refer to the MergeCOM-3 Reference Manual, Appendix B, for a discussion of the following configuration parameters that are used to configure the network capture facility:

```
NETWORK_CAPTURE
CAPTURE_FILE
CAPTURE_FILE_SIZE
NUMBER_OF_CAP_FILES
REWRITE_CAPTURE_FILES
```

The library provides the `MC_Register_Network_Capture_Callbacks( )` function that allows you to, in effect, replace the default MergeCOM-3 network capture handler with one of your own.

Service Profile

items

The Service Profile (usually called `mergecom.srv`) informs the tool kit what types of services and commands it supports, and what the corresponding message info files are. This file also lists the meta-services and **items** supported by the tool kit. Items are the nested 'sub-messages' contained within attributes of a message having the VR Sequence of Item (SQ) and will be discussed in greater detail later. The name and location of the service profile are specified in the `[MergeCOM3]` section of the MergeCOM-3 initialization file.

The service profile, along with the data dictionary and message info files, is generated from the MergeCOM-3 DICOM Database and should be modified by other means only by very experienced or specialized users.

Message Logging

MergeCOM-3 Advanced supplies a message logging facility whereby three primary classes of messages can be logged to a specified file and/or standard output:

- Errors
- Warnings
- Status

Error messages include unrecoverable errors, such as "association aborted", or "failure to connect to remote application". Other error messages may be catastrophic but it is left to the application to determine whether or not to abort an association, such as an "invalid attribute value" or "missing attribute value" in a DICOM message.

Warnings are meant to alert tool kit users to unusual conditions, such as missing parameters that are defaulted or attributes having values that are not one of the defined terms in the standard.

Status messages give high-level messages describing the opening of associations and exchanging of messages over open associations.

As discussed earlier, other more detailed logging can be obtained by using the `T1_MESSAGE` through `T9_MESSAGE` logging levels. For example, the `T5_MESSAGE` logging level can be used to log the results of an `MC_Validate_Message( )` call (see the Reference Manual).

An excerpt from a MergeCOM-3 Advanced message log file is included below that contains all three classes of messages: errors, warnings, and informational.

Message Log Example:

```
                                .
                                .
                                .
03-29 21:14:54.77 MC3 W: (0010,1010): Value from stream had
problem:
03-29 21:14:54.78 MC3 W: |   Invalid value for this tag's VR
03-29 21:14:56.41 MC3(Read_PDU_Head) E: Error on
Read_Transport call
03-29 21:14:56.41 MC3(MCI_nextPDUtype) E: Error on
Read_PDU_Head call
03-29 21:14:56.41 MC3(Transport_Conn_Closed_Event) E:
Transport unexpectedly closed
03-29 21:14:56.41 MC3(MCI_ReadNextPDV) I: DUL_read_pdvs
error: UL Provider aborted the association
03-29 21:14:56.41 MC3 E: (0000,0000): Error during
MC_Stream_To_Message:
03-29 21:14:56.41 MC3 E: |   Callback cannot comply
03-29 21:14:56.41 MC3(MC_Read_Message) E: Network connection
unexpectedly shut down
                                .
                                .
                                .
```

On more advanced computing platforms, additional information is logged, such as process and thread id numbers identifying where the message was generated.

Utility Programs

The MergeCOM-3 Advanced Tool Kit supplies several useful utility programs. These utilities can be used to help you validate your own implementations and better understand the standard.

All these utilities use the MergeCOM-3 Advanced Library and require that you set your `MERGE_INI` environmental variable to point to the proper configuration files (as described earlier).

mc3comp

Do a DICOM 'diff"

The `mc3comp` utility can be used to compare the differences between two DICOM objects. The objects can be encoded in either the DICOM file or "stream" format and do not have to be encoded in the same format. The utility will output differences in tags between the messages taking into account differences in byte ordering and encoding. The syntax for the utility is the following:

```
mc3comp [-t1 <syntax> -t2 <syntax>] [-e file]
        [-o -m1 -m2] file1 file2
```

```
-t1 <syntax>  Optional specify transfer syntax of 'file1'
              message, where <syntax> = 'il' for implicit
              little endian (default), 'el' for explicit
              little endian, 'eb' for explicit big endian
-t2 <syntax>  Optional specify transfer syntax of 'file2'
              message, where <syntax> = 'il' for implicit
              little endian (default), 'el' for explicit
              little endian, 'eb' for explicit big endian
-e <file>     Optional exception file of all tags to ignore
              in comparison
-o            Compare OB/OW (e.g., binary pixel) data
-m1           Compare 'file1' in DICOM-3 file format.
-m2           Compare 'file2' in DICOM-3 file format.
-h            Show these options.
file1         DICOM SOP Instance (message) file
file2         Another DICOM SOP Instance (message) file

Example:   mc3comp -t1 il -m2 -o 1.img 1.dcm
```

mc3conv

Convert image formats

The mc3conv utility can be used to convert a DICOM object between various transfer syntaxes and formats. The utility will read an input file and then write the output file in the transfer syntax specified in the command line. The utility can also convert between DICOM "stream" format and the DICOM file format. The syntax for the mc3conv utility is the following:

```
mc3conv input_file output_file [-t <syntax>] [-m]
        [-tag <tag> <"new value">]

input_file    DICOM SOP Instance (message) file
output_file   Output DICOM SOP Instance (message) file
-t            Specify transfer syntax for 'output_file',
              where <syntax> = 'il' for implicit little
              endian (default), 'el' for explicit little
              endian, 'eb' for explicit big endian
-m            Specify format of 'output_file' to be DICOM-3
              media (Part 10) format.
-tag          Change value for this tag in 'output_file',
              where <tag> = the tag that is to be changed in
              hex 0x... <new value> = the value for the tag
              in quotes, multi values separated as
              "val1\val2"
-h            Show these options.

Example:  mc3conv in.img out.dcm -t el -m
```

mc3echo

Do a DICOM 'ping'

The mc3echo utility validates application level communication between two DICOM AE's. An echo test is the equivalent of a network 'ping' operation, but at the DICOM application level rather than the TCP/IP transport level.

All server (SCP) applications built with the advanced tool kit also have built-in support of the Verification Service Class and the C-ECHO command.

The command syntax follows:

```
mc3echo [-c count] [-r remote_host]
        [-l local_app_title] [-p remote_port]
        remote_app_title

-c count      Integer number specifying the number of echoes
              to send to the remote host.  If -c is not
              specified, one echo will be performed
-r remote_host      Host name of the remote computer  If -r
              is not specified, the default value for
```

```
                remote_host is configured in the Application
                Profile.
-l local_app_title  Application title of this program.  If -
                l is not specified, the default value for
                local_app_title is MERGE_ECHO_SCU
-p remote_port      Port number the remote computer is
                listening on.  If -p is not specified, the
                default value for remote_host is configured in
                the Application Profile.
```

mc3list

Display message contents

`mc3list` displays the contents of binary DICOM message files in an easy to read manner. The message files could have been generated by `mc3file` (see below) or written out by your application.

`mc3list` is a useful educational tool as well as a tool that can be used for off-line display of the DICOM messages your application generates or receives.

The command syntax follows:

```
mc3list <filename> [-t <syntax>] [-m]

filename      Filename containing message to display
-t            Specify transfer syntax of message, where
              syntax = "il" (implicit little endian), "el"
              (explicit little endian), or "eb" (explicit
              big endian)
-m            Optional display a DICOM file object
```

If the DICOM service and/or command cannot be found in the message file, a warning will be displayed, but the message will still be listed.

The default transfer syntax is implicit little endian (the DICOM default transfer syntax). If the transfer syntax is incorrectly specified, the message will not be displayed correctly.

mc3valid

DICOM message validation tool

The `mc3valid` utility validates binary message files according to the DICOM standard and notifies you of missing attributes, improper data types, illegal values, and other problems with a message. `mc3valid` is a powerful educational and validation tool that can be used for the off-line validation of the DICOM messages your application generates or receives.

The command syntax follows:

```
mc3valid <filename> [-e | -w | -i] [-s <serv> -c <cmd>] [-l]
                    [-m] [-q] [-t <syntax>]
<filename>    Filename containing message to validate
-e            Display error messages only
-w            Display error and warning messages (default)
-i            Display informational, error, and warning
              messages
-s <serv>     Optional force the message to be validated
              against service name "serv", used along with
              '-c'
-c <cmd>      Optional force the message to be validated
              against command name "cmd", used along with '-
              s'
-l            Optional list and select possible service-
              command pairs
-m            Optional specify the input file as being a
              DICOM file object
-q            Optional disable prompting for correct
              service-command pairs
```

```
-t          Specify transfer syntax of message, where
            syntax = "il" (implicit little endian), "el"
            (explicit little endian), or "eb" (explicit
            big endian)
```

This command validates the specified message file; printing errors, warnings, and information generated to standard output. The user can force the message to be validated against a specified DICOM service-command pair if the message does not already contain this information.

If the service-command pair is not contained in the message, the program will list the possible service-command pairs and the user can select one of them. When using this program with a batch file, this option can be shut off with the -q flag.

The default transfer syntax is implicit little endian (the DICOM default transfer syntax). If the transfer syntax is incorrectly specified, the message cannot be validated.

limitations

While `mc3valid`'s message validation is quite comprehensive, it does have limitations. These limitations are discussed in detail in the description of the `MC_Validate_Message( )` function in the Reference Manual. The DICOM Standard should always be considered the final authority.

mc3file

DICOM message generation tool

Sample DICOM messages can be generated with the `mc3file` utility. You specify the service, command, and transfer syntax and `mc3file` generates a 'reasonable' sample message that is written to a binary file. The contents of this file are generated in DICOM file format or in exactly the format as the message would be streamed over the network.

The program fills in default values for all the required attributes within the message. You can also use this utility to generate its own configuration file, which you can then modify to specify your own values for attributes in generated messages.

These generated messages are purely meant as 'useful' examples that can be used to test message exchange or give the application developer a feel for the structure of DICOM messages. They are not intended to represent real world medical data.

The messages generated can be validated or listed with the `mc3list` and `mc3valid` utilities. The command syntax for `mc3file` is the following:

```
mc3file <serv> <cmd> <num> [-g <file>] [-c <file>] [-l] [-m] [-q]
        [-t <syntax>] [-f <file>]

       <serv> <cmd>  These two options are always used together.
                     They specify the service name and command for
                     the message to be generated.  These names can
                     be either upper or lower case.  If the exact
                     names for a service command pair are not
                     known, the -l option can be used instead to
                     specify the service name and command.  If the
                     service name and command are improperly
                     specified, mc3file will act as if the -l
                     option was used and ask the user to input the
                     correct service name and command.
       <num>         This option specifies the number of message
                     files to be generated by mc3file.  If the -g
                     option is used, this option is not needed on
                     the command line.  If the -c option is used,
                     mc3file assumes the number is 1, although a
                     higher number can be specified on the command
```

line. mc3file will vary any fields that have a value representation of time when multiple files are generated, although when the -c option is used, the utility will use the time fields as specified in the configuration file. Thus multiple message files generated with the -c option are identical.

-g <filename> This option causes mc3file to generate an ASCII configuration file. The file contains a listing of all the valid attributes for the specified message. The utility also adds sequences contained in the message along with their attributes. Each attribute in the file contains the tag, value representation, and the default value MC3File uses for the attribute. If a given attribute has more than one value, the character "\" is used to delimit the values. A default value listed as "NULL" means the attribute is set to NULL. If the filename specified already exists, it will be written over my MC3File. The configuration file can be modified and reloaded into MC3File with the -c option to generate a DICOM message.

-c <filename> This option reads in a configuration file previously generated by mc3file. The service name and command for the message need not be specified on the command line because they are contained in <filename>. Because multiple files generated with this option are identical, mc3file assume only one file should be generated. This assumption can be overridden by specifying a number on the command line.

-l This option lists all the service command pairs supported by mc3file. When generating a message, this option can be used instead of explicitly specifying the service name and command on the command line. When specified alone in the command line, the complete list of pairs is printed out without pausing.

-m This option allows the user to generate a DICOM file. When generating the file object, mc3file encodes the File Meta Information.

-q This option prevents mc3file from prompting the user for correct service command pairs. It is a useful option when running the program from a batch file.

-t <syntax> This option specifies the transfer syntax the DICOM message generated is stored in. The default transfer syntax is implicit little endian. The possible values for <syntax> are "il" for implicit little endian, "el" for explicit little endian, and "eb" for explicit big endian.

-f <file> This option allows the user to specify the first eight characters of the names of the DICOM message files being generated. mc3file will then append a unique count to the end of the filename for each message being generated. The default value is "file" when creating a DICOM file and "message" when creating the format that DICOM messages send over a network.

MC3File retrieves default values for attributes from the text file "default.pfl". Unlike the "info.pfl" and "diction.pfl" files which are converted into binary files, "default.pfl" is used as a text file. It will first be searched for in the current directory and then in the message information directory. This file contains default

values for all messages and for specific service-command pairs. This file can be modified to contain defaults specific for the user, although it is recommended that a backup of the original be kept. If this file is modified, there are no guarantees that the messages generated will validate properly.

Developing DICOM Applications

The MergeCOM-3 Advanced Application Programming Interface (API) provides simple yet powerful DICOM functionality. Function calls are provided that open associations with remote servers, wait for associations from remote clients, and deal with DICOM message exchange over an open association. Functions are also provided for the creation and reading of DICOM files and the creation, maintenance, and navigation of DICOMDIR's. Advanced features include message validation against the DICOM Standard, support of sequences of items, Callback Functions for flexible handling of Pixel Data, and support of Private Attributes.

This section of the User's Manual attempts to present the highlights of the MergeCOM-3 Advanced API in a logical manner as it might be used in real DICOM applications. The function calls are presented in the context of example ANSI-C source code snippets, and alternative approaches are presented that tradeoff certain features for the benefits of increased performance.

Most of the discussions that follow pertain both to networking and media interchange applications; only the *Association Management, Negotiated Transfer Syntaxes,* and *Message Exchange* sections are networking specific. The last two sections; *DICOM Files* and *DICOMDIR* are media interchange specific.

Library Initialization

Your first call to the MergeCOM-3 Advanced Library must always be the `MC_Library_Initialization( )` function. This function specifies how and when you wish to initialize the library with the contents of its configuration files, data dictionary, and message info files.

Almost all typical applications will initialize themselves from configuration files, and make use of the binary dictionary and message info files in building message objects. So in most cases, the initialize call will look like the following:

```
MC_Status = MC_Library_Initialization (NULL, NULL, NULL);
```

Configurable parameters can be modified by your application after library initialization, at runtime, by using the `MC_Set_..._Config_Value( )` functions. These functions are detailed in the Reference Manual.

If you change the configuration of the library at runtime using the `MC_Set_..._Config_Values( )` functions and wish to reset it to its initial state, you should use the `MC_Library_Reset( )` function, which has no parameters. It resets the library to its initial state using the same options as originally specified in the `MC_Library_Initialization( )` call.

check the return codes

You should always check the return code from any function call to MergeCOM-3 to see if an error occurred. Any value other than `MC_NORMAL_COMPLETION` implies an error. A possible error code for this call is `MC_INVALID_LICENSE` when the tool kit is not configured with a valid license number. All error codes that can be returned for each API function call are specified in the Reference Manual.

Statically Linked Configuration

In specialized applications (such as embedded systems or where performance at initialization time is critical) one can specify different parameter values to `MC_Library_Initialization( )` to call initialization functions rather than accessing files. These initialization functions are generated by two tools supplied with MergeCOM-3 Advanced: `genconf` and `gendict`.

`genconf` generates a ANSI-C module containing the `MC_Config_Values( )` function providing the contents of the configuration files. This module can be compiled and linked into your application, along with the tool kit library, to supply an initial configuration without accessing the file system. This is done by specifying `MC_Config_Values` as the first parameter to `MC_Library_Initialization( )`.

`gendict` is similar to `genconf` in that it generates an ASCII C module, but its function is named `MC_Dictionary_Values( )` and can be used to initialize the library with the contents of the binary data dictionary file. By compiling and linking this module into your application, and by specifying `MC_Dictionary_Values` as the second parameter to the `MC_Library_Initialization( ),` your application can access the data dictionary without accessing the file system.

The third parameter of `MC_Library_Initialization( )` is reserved for future use. See the Reference Manual for further details on these two tools and `MC_Library_Initialization( )`.

Registering Your Application

Before performing any network or media activity, your application must register its **DICOM Application Title** with the MergeCOM-3 Advanced Tool Kit. The tool kit returns to you an **Application ID,** a handle that is used to refer to this particular AE in subsequent function calls.

This DICOM Application Title is equivalent to the DICOM Application Entity Title defined earlier. If your application is a server, this application title must be made known to any client application that wishes to connect to you. If your application is a client, your application title may need to be made known to any server you wish to connect to, depending on whether the server is configured to act as an server (SCP) only to particular clients for security reasons.

For example, if your application title is "ACME_Query_SCP", you would register with the tool kit as follows:

```
MC_Status = MC_Register_Application( &MyAppID,
       "ACME_Query_SCP");
```

If you wish to disable your application and free up its resources to the system you should release it as follows:

```
MC_Status = MC_Release_Application(&MyAppID);
```

Even if your application is a media interchange only application, you still need to register it so that the Advanced Library has some way to refer to this application in other calls; such as `MC_Register_Callback_Function( ).`

Current and potentially future DICOM service classes assume that Application Entity Titles on a DICOM network are unique. For instance, the retrieve portion of the Query/Retrieve service class specifies that an image be moved to a specific Application Entity Title (and not to a specific hostname and listen port). If two

identical Application Entity Titles existed on a network, a server application can only be configured to move images to one of these applications. For this reason, the DICOM Application Entity Title for your applications should be configurable.

Association Management (Network Only)

Once you have registered one or more networking applications, you will probably want to initiate an association if you are a client, or wait for an association if you are a server.

Opening and closing an association

To initiate an association as a client, you make an `MC_Open_Association( )` call. You specify your Application ID and the Remote Application Title of the server you wish to connect to. If the association is accepted, the function returns normally, with an Association ID that is used to refer to this association in future calls. When you are done making DICOM service requests (sending and receiving messages) over the association, you should release the association with an `MC_Close_Association( )` call.

You also have the option of using the `MC_Open_Secure_Association( )` call. Use that function if you will be supplying additional code that will maintain a secure connection using a protocol such as Secure Socket Layer (SSL) or Transport Layer Security (TLS). Refer to the MergeCOM-3 Reference manual for more information about the `MC_Open_Secure_Association( )` call.

Client Side Example:

```
status = MC_Open_Association( MyStoreClientId,
       &associationID, RemoteAppTitle, NULL, NULL, NULL);
if (status != MC_NORMAL_COMPLETION)
{
       printf("Unable to open association with
              \"%s\":\n", RemoteAppTitle);
       printf("\t%s\n", MC_Error_Message(status));
       return 1;
}
else
       printf("Connected to remote system [%s]\n",
              RemoteAppTitle);

/* Do your message exchange here */

status = MC_Close_Association(&associationID);
if (status != MC_NORMAL_COMPLETION)
{
       printf("Close association failed\n");
       printf("\t%s\n", MC_Error_Message(status));
       return 1;
}
/*
 * Now you can exit normally.
 */
```

Waiting for an association

To wait for an association request as a server application, you make the `MC_Wait_For_Association( )` call. You can specify a timeout in this call to indicate how long you wish to wait for a valid association request; if you specify a timeout of -1 you wait forever. If this call returns with a status of `MC_NORMAL_COMPLETION`, an Application ID is returned that indicates the AE that has received the valid association request along with an Association ID for the new association. This Association ID is used to refer to this particular association in future calls. The server application must either `MC_Accept_Association( )`

or MC_Reject_Association( ) before DICOM messages can be exchanged over the association.

The server application detects when the client has released the association in the last message received from the client. This will be discussed further in later sections dealing with DICOM message exchange.

You also have the option of using the MC_Wait_For_Secure_Association( ) call. Use that function if you will be supplying additional code that will maintain a secure connection using a protocol such as Secure Socket Layer (SSL) or Transport Layer Security (TLS). Refer to the MergeCOM-3 Reference manual for more information about the MC_Wait_For_Secure_Association( ) call.

Server Side Example:

```
status = MC_Wait_For_Association("Service_List_1",
            -1, &calledApplicationID, &associationID);
if (status != MC_NORMAL_COMPLETION)
{
      printf("\tError on MC_Wait_For_Association:\n");
      printf("\t\t%s\n", MC_Error_Message(status));
      printf("\t\tProgram aborted.\n");
      abort();
}
if (calledApplicationID != MyApplicationID)
{
      printf("\tUnexpected application identifier on \
            MC_Wait_For_Association.\n");
      printf("\t\tProgram aborted.\n");
      abort();
}
status = MC_Accept_Association(*associationID);
if (status != MC_NORMAL_COMPLETION)
{
      printf("\tError on MC_Accept_Association:\n");
      printf("\t\t%s\n",MC_Error_Message(status));
      return;
}
/*
 * Handle message exchange here.  It is during message
 * exchange where you detect that the association has been
 * closed and act accordingly
 */
```

Querying the associations characteristics

Three additional functions; MC_Get_Association_Info( ), MC_Get_First_Acceptable_Service( ), and MC_Get_Next_Acceptable_Service( ) allow the client or server to query the characteristics of an association. This is useful to a client, so that it knows what subset of services, transfer syntaxes and DICOM roles that it proposed have been accepted and can now perform with the server. Similarly, a server can look at characteristics of the association request (such as the network node name of the client) and either accept or reject the association. See the Reference Manual for a detailed description of these functions.

Using select( ) to handle asynchronous events

In specialized cases where the server application is waiting on several asynchronous events, not just the association event, the MC_Get_Listen_Socket( ) call can be made to request the file descriptor for the DICOM listen socket. In this way the server application can do a select( ) system call on this and other file descriptors. When the select returns with an event on the DICOM listen socket descriptor, the application can call MC_Wait_For_Association( ) and get an immediate response.

Similarly, once the association is established, both the client and server applications can use the `MC_Get_Association_Info( )` call to get the file descriptor for the socket over which message exchange will occur. Again, `select( )` can be used to wait asynchronously for a DICOM request or response message. This is discussed further later, under *Message Exchange*.

Negotiated Transfer Syntaxes (Network Only)

MergeCOM-3 Advanced supports all currently approved standard and encapsulated DICOM transfer syntaxes. Encapsulated transfer syntaxes require compression of the pixel data contained in the message. These messages can be sent and received by the tool kit, although the tool kit will not do the actual compression and decompression. Encoding of this pixel data is discussed below.

For advanced users, the tool kit allows for the negotiation of more than one transfer syntax for a given DICOM service. This functionality is of most use for applications supporting encapsulated transfer syntaxes. This functionality may be disabled by use of the `ACCEPT_MULTIPLE_PRES_CONTEXTS` configuration value. In order to understand how it is implemented, a more in depth description of DICOM association negotiation is required.

During association negotiation a client (SCU) application will propose a set of presentation contexts over which DICOM communication can take place. Each presentation context consists of an abstract syntax (DICOM service) and a set of transfer syntaxes that the client (SCU) understands. The server (SCP) will typically accept a presentation context if it supports the abstract syntax and one of the proposed transfer syntaxes.

As previously discussed, the abstract and transfer syntaxes supported by a server (SCP) are defined through a service list contained in the MergeCOM-3 Application Profile. When support within a server (SCP) is limited to the three non-encapsulated DICOM transfer syntaxes, the tool kit will transparently handle the use of multiple presentation contexts for a DICOM service. However, when encapsulated DICOM transfer syntaxes are used, the server (SCP) must be able to determine the transfer syntax of messages it receives so that it can properly parse the pixel data contained in them. When a single presentation context is negotiated for a DICOM service, the `MC_Get_First_Acceptable_Service( )` and `MC_Get_Next_Acceptable_Service( )` functions can be used to determine the transfer syntax for a service. When more than one presentation context is negotiated for a service, the `MC_Get_Message_Transfer_Syntax( )` function must be used to retrieve this transfer syntax. The following is a typical call to this function:

```
MC_Status  =  MC_Get_Message_Transfer_Syntax(ImageMsgID,
                    &transferSyntax);
```

Exchange of messages over the network is discussed further below.

Transfer Syntax Lists for SCUs

The presentation contexts supported for client (SCU) applications using MergeCOM-3 are also defined through the MergeCOM-3 Application Profile. The following is a typical client's (SCU) configuration:

```
[Acme_Store_SCP]
    PORT_NUMBER     = 104
    HOST_NAME       = acme_sun1
    SERVICE_LIST    = Storage_Service_List

[Storage_Service_List]
    SERVICES_SUPPORTED   = 1  # Number of Services
    SERVICE_1            = STANDARD_CT
```

In this case, the client (SCU) would propose the CT Image Storage service in a single presentation contexts. The transfer syntaxes for each service are the three standard (non-encapsulated) DICOM transfer syntaxes.

The following example is the configuration for a client (SCU) that supports more than one presentation context for a service:

```
[Acme_Store_SCP]
    PORT_NUMBER     = 104
    HOST_NAME       = acme_sun1
    SERVICE_LIST    = Storage_Service_List

[Storage_Service_List]
    SERVICES_SUPPORTED   = 2  # Number of Services
    SERVICE_1            = STANDARD_CT
    SYNTAX_LIST_1        = CT_Syntax_List_1
    SERVICE_2            = STANDARD_CT
    SYNTAX_LIST_2        = CT_Syntax_List_2

[CT_Syntax_List_1]
    SYNTAXES_SUPPORTED   = 1  # Number of Syntaxes
    SYNTAX_1             = JPEG_BASELINE

[CT_Syntax_List_2]
    SYNTAXES_SUPPORTED   = 1  # Number of Syntaxes
    SYNTAX_1             = IMPLICIT_LITTLE_ENDIAN
```

If a server (SCP) accepts both of these presentation contexts, the client (SCU) must use the `MC_Set_Message_Transfer_Syntax( )` function to specify which presentation context to send a message over as follows:

```
MC_Status  =  MC_Set_Message_Transfer_Syntax(ImageMsgID,
                    JPEG_BASELINE);
```

Transfer Syntax Lists for SCPs

Server (SCP) applications are configured differently than client (SCU) applications. An SCP should include all of the transfer syntaxes a service supports in a single transfer syntax list. If more than one transfer syntax list is used for a service, server (SCP) applications will only support the transfer syntaxes contained in the first transfer syntax list. The following is an example configuration for a server (SCP):

```
[Storage_Service_List]
    SERVICES_SUPPORTED   = 1  # Number of Services
    SERVICE_1            = STANDARD_CT
    SYNTAX_LIST_1        = CT_Syntax_List_SCP

[CT_Syntax_List_SCP]
    SYNTAXES_SUPPORTED   = 4  # Number of Syntaxes
    SYNTAX_1             = JPEG_BASELINE
    SYNTAX_2             = EXPLICIT_LITTLE_ENDIAN
    SYNTAX_3             = IMPLICIT_LITTLE_ENDIAN
    SYNTAX_4             = EXPLICIT_BIG_ENDIAN
```

As discussed previously, for server (SCP) applications, the order in which transfer syntaxes are specified in a transfer syntax list dictates the priority MergeCOM-3 places on them during association negotiation. In this case, MergeCOM-3 would select `JPEG_BASELINE` if proposed, followed by `EXPLICIT_LITTLE_ENDIAN`, `IMPLICIT_LITTLE_ENDIAN`, and `EXPLICIT_BIG_ENDIAN`.

Network message exchange is discussed further in one of the following sections.

Message Objects

Objects are useful

MergeCOM-3 Advanced supplies several types of **objects**: application objects, association objects, message objects, file objects, and item objects. Whenever you are given an ID (e.g., Application ID, AssociationID, MessageID, FileID, or ItemID), it is a handle to an instance of one of these objects. Objects provide a convenient way for the tool kit to encapsulate related data while hiding unnecessary details from the application developer. ID's also provide a convenient shorthand when making calls to the MergeCOM-3 tool kit that operate on or make use of these objects.

A majority of the functionality supplied with the advanced tool kit deals with building, parsing, validation, and exchange of DICOM messages, files, and items. Your applications deal with network messages in MergeCOM-3 as message objects, and DICOM files as file objects. Item objects are used for attributes that are of VR Sequence of Item (SQ) within both messages and files.

This section deals with message objects, but many of these functions are polymorphic and also work on file and item objects. These polymorphic functions will be called out in this section. Additional functions that are particular to file objects or item objects will be described in later sections.

Private attributes in both message and file objects are handled in ways similar to those discussed in this section, but will also be described later in this document.

Building Messages

Opening a message

Before you can build a message, your application must create a message object using the `MC_Open_Message( )` function call. In this call, you specify a Service and Command name. The advanced library uses these parameters to reference the proper message info file along with the data dictionary and builds an unpopulated message object instance for your application to fill in. This message object contains empty attributes. A Message ID is returned to your application that identifies this message object.

Performance Tuning

An alternative method exists for creating an empty message object, `MC_Open_Empty_Message( )`, where a service and command are not specified. In this case, the message info and data dictionary files are not referenced and an empty message object instance is opened. This message object contains no attributes and the `MC_Set_Service_Command( )` function must be called to set the service and command for this message before it can be sent over the network. Since this approach avoids accessing the message info files, it is more efficient. However, this approach also penalizes you in terms of runtime error checking. This is discussed further later.

Filling a message with values

Once you have an open message object, use the `MC_Set_Value` family of functions to build your message. This family of functions can be broken into five types based on their functionality as specified in Table 10. Most of these functions have three parameters in common; a `MsgItemID` to specify the message object, a `Tag` to specify the attribute whose value is being set, and a `Value` being assigned to the attribute. See the Reference Manual for detailed specifications of these functions.

`MC_Set_Value` functions are polymorphic

The `MC_Set_Value` family of functions also operate on DICOM file objects and item objects, since both of these objects are also constructed of DICOM attributes.

Function Type	Description

MC_Set_Value...( )	Sets the first (and possibly only) value of an attribute in a message object. There are ten functions of this type, the one you use depends on the data type of the value you are assigning to the attribute (e.g., string, short int, long int, float, ...).
MC_Set_Next_Value...( )	Sets the next value of a multi-valued attribute in a message object. There are ten functions of this type, the one you use depends on the data type of the value you are assigning to the attribute (e.g., string, short int, long int, float, ...).
MC_Set_Value_To_NULL( )	Sets the value of an attribute in a message object to NULL. This means that the attribute is included in the message but has a NULL value.
MC_Set_Value_To_Empty( )	Clears the value of an attribute in a message object. This means the attribute has no value and is not included in the message.
MC_Set_Value_From_Function( )	Specifies a function that is called repeatedly to set the value of an attribute of VR OB, OW, OF, SL, SS, UL, US, AT, FL, or FD (e.g., pixel data or fixed width value) a 'chunk' at a time. These attributes tend to have very large values. **Note:** This callback function is different than the type of Callback Function registered using `MC_Register_Callback_Function( )`. See the later discussion of Callback Functions.

Table 10: The MC_Set_Value Family of Functions

Both the `MC_Set_Value` and `MC_Set_Next_Value` function types have several variants depending on the data type of the variable from which you are assigning the value (see Table 11). These variants can be very helpful because they perform the necessary type conversion from any reasonable ANSI-C data type to any compatible DICOM value representation. If a type conversion is not reasonable (e.g., from `short int` to LT), then `MC_INCOMPATIBLE_VR` will be returned as a MC_STATUS code. Also, other error statuses will be returned if the conversion was reasonable but the value stored in the variable made the conversion impossible (e.g., `MC_INVALID_VALUE_FOR_VR`, `MC_VALUE_OUT_OF_RANGE`,...).

Performance Tuning

If your message object was opened using `MC_Open_Message( )`, the status code `MC_INVALID_TAG` will be returned if you attempt to set the value of an attribute that is not a part of that message object. This additional level of validation is lost if you use `MC_Open_Empty_Message( )`, since this opens an empty message object without any attributes to check against. Applications where performance is critical may find it useful to use `MC_Open_Message( )` during initial development, and replace these calls with `MC_Open_Empty_Message( )` later in the development cycle after the implementation stabilizes.

MC_Set function type	Function may be used to set values of attributes with these Value Representations
MC_Set_Value_From_Int	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Set_Value_From_UInt	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Set_Value_From_ShortInt	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Set_Value_From_UShortInt	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Set_Value_From_LongInt	AT, DS, FD, FL, IS, SL, SS, UL, US, SQ

MC_Set_Value_From_ULongInt	AT, DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Set_Value_From_Float	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Set_Value_From_Double	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Set_Value_From_String	AS, AT, CS, DA, DS, DT, FD, FL, IS, LO, LT, PN, SH, SL, SS, ST, TM, UI, UL, US, UT, SQ
MC_Set_Value_From_Function	OB, OW, OF, SL, SS, UL, US, AT, FL, FD

Table 11: Acceptable MC_Set_Value/VR combinations

An example of opening a message for the BASIC_FILM_SESSION - N-CREATE-RQ service-command pair, and setting a few of its attributes follows. Note that after the message is sent, the message object is freed. Be sure to free message objects when your application is done with them. You can keep an old message object around if you plan to reuse the message object often, and have the available memory.

```
/*
 *   Send the Film Session Creation message
 */

status = MC_Open_Message(&messageID, "BASIC_FILM_SESSION",
                            N_CREATE_RQ);

if (status != MC_NORMAL_COMPLETION)
{
       printf("Unable to open request message:\n");
       printf("\t%s\n", MC_Error_Message(status));
       return 1;
}

session_sop = "1.2.840.10008.5.1.1.1";
status = MC_Set_Value_From_String(messageID, 0x00000002,
                         session_sop);
if (status != MC_NORMAL_COMPLETION)
{
       printf("MC_Set_Value_From_String failed:\n");
       printf("\t%s\n", MC_Error_Message(status));
       MC_Free_Message(&messageID);
       MC_Abort_Association(&associationID);
       MC_Release_Application(&applicationID);
       return 1;
}

Ssession_uid = "1.2.840.10008.75.89";
status = MC_Set_Value_From_String(messageID, 0x00001000,
                                  Ssession_uid);
if (status != MC_NORMAL_COMPLETION)
{
       printf("MC_Set_Value_From_String failed:\n");
       printf("\t%s\n", MC_Error_Message(status));
       MC_Free_Message(&messageID);
       MC_Abort_Association(&associationID);
       MC_Release_Application(&applicationID);
       return 1;
}

status = MC_Set_Value_From_String(messageID,
                     MC_ATT_NUMBER_OF_COPIES, "1");
if (status != MC_NORMAL_COMPLETION)
{
       printf("MC_Set_Value_From_String failed:\n");
       printf("\t%s\n", MC_Error_Message(status));
```

```
        MC_Free_Message(&messageID);
        MC_Abort_Association(&associationID);
        MC_Release_Application(&applicationID);
        return 1;
}

/*
 * Set other attributes here...
 */

status = MC_Send_Request_Message(associationID, messageID);
if (status != MC_NORMAL_COMPLETION)
{
        printf("MC_Send_Request_Message failed:\n");
        printf("\t%s\n", MC_Error_Message(status));
        MC_Free_Message(&messageID);
        MC_Abort_Association(&associationID);
        MC_Release_Application(&applicationID);
        return 1;
}

(void)MC_Free_Message(&responseMessageID);
```

Supplying pixel data

When setting the value of an attribute with a value representation of OB or OW (e.g., Pixel Data), you can use the MC_Set_Value_From_Function( ). Pixel Data tends to be very large and normally you use this function to supply the data value a 'chunk' or block at a time.

As an example, your application could define a callback function MyPDSupplyFunction( ) whose purpose is to supply Pixel Data. Pseudo-code for this function follows:

```
MC_STATUS MyPDSupplyFunction(int MsgID, unsigned long Tag,
int IsFirstCall, void *UserInfo, int *BlockSize, void
**DataBlock, int *IsLastBlock)
{
        if (IsFirstCall)
        {
                /*
                 * open pixel data source (e.g., file) here
                 * using Tag and/or *UserInfo as a guide.
                 */

                if (OpenFailed)
                        return(MC_CANNOT_COMPLY);

        }

        /*
         * Read next chunk of pixel data from source
         * into **DataBlock and set *BlockSize to the size of
         * the chunk read in.
         */

        if (ReadFailed)
                return(MC_CANNOT_COMPLY);

        if (LastBlockRead)
        {
                /*
                 * close Pixel Data source here.
                 */

                *IsLastBlock = TRUE;
        }

        return(MC_NORMAL_COMPLETION);
}
```

This callback function is called by the MergeCOM-3 Library only when triggered by your application. For example, your application might use MyPDSupplyFunction to set the value of the MC_ATT_PIXEL_DATA attribute (7FE0, 0010) as follows:

```
MC_Status = MC_Set_Value_From_Function( ImageMsgID,
       MC_ATT_PIXEL_DATA, NULL, MyPDSupplyFunction);
```

On making this call, the tool kit library will repetitively callback MyPDSupplyFunction( ) until it indicates that all the pixel data has been read in without any errors. In this case no user data is passed through to the callback function since *UserData is NULL.

Performance Tuning

Supplying Pixel Data a block at a time is especially useful for very large Pixel Data and/or on platforms with resource (e.g., memory) limitations. In this case you would also want to set LARGE_DATA_STORE to the value FILE in the Service Profile, and MergeCOM-3 Advanced will store the Pixel Data value in a temporary file.

If your application runs on a resource rich system, you should set LARGE_DATA_STORE to the value MEM in the Service Profile, and MergeCOM-3 Advanced will keep the Pixel Data values in the message object stored in memory rather than using temporary files. This should improve performance. Also, in this case you may want your callback function to supply the Pixel Data in fewer big blocks (or one large block).

While MyPDSupplyFunction( ) is a callback function in this example, it is not what is being referred to when we discuss Callback Functions (with a capital 'C' and capital 'F') in MergeCOM-3 Advanced. Callback Functions are another even more powerful way to handle large OB or OW data and are discussed later.

Parsing Messages

When your AE receives a DICOM message, it will most often need to examine the values contained in the message attributes to perform an action (e.g., store an image, print a film, change state...). If your application is a server, the message conveys the operation your server should perform and the data associated with the operation. If your application is a client, the message may be a response message from a server on the network resulting from a previous request message to that same server.

Reading values from a message

Once you have received a message object, use the MC_Get_Value family of functions to parse your message. This family of functions can be broken into five types based on their functionality as specified in Table 12. Most of these functions have three parameters in common; a MsgItemID to specify the message object, a Tag to specify the attribute whose value is being fetched, and a Value variable to which the value stored in the attribute is assigned. See the Reference Manual for detailed specifications of these functions.

MC_Get_Value functions are polymorphic

The MC_Get_Value family of functions also operate on DICOM file objects and item objects, since both of these objects are also constructed of DICOM attributes.

Function Type	Description
MC_Get_Value_Count( )	Returns the number of values assigned to an attribute in a message object. Multi-valued attributes can have more than one value assigned to them.

MC_Get_Value_Length( )	Returns the length of attribute's value in bytes.
MC_Get_Value...( )	Gets the first (and possibly only) value of an attribute in a message object. There are eleven functions of this type, the one you use depends on the data type of the variable to which you are assigning to the attribute's value (e.g., string, short int, long int, float, ...).
MC_Get_Next_Value...( )	Gets the next value of a multi-valued attribute in a message object. There are eleven functions of this type, the one you use can depend on the data type of the value you are assigning to the attribute (e.g., string, short int, long int, float, ...).
MC_Get_Value_To_Function( )	Specifies a function that is called repeatedly to get the value of an attribute of VR OB, OW, OF, SL, SS, UL, US, AT, FL, or FD (e.g., pixel data or fixed width value) a 'chunk' at a time. These attributes tend to have very large values. **Note:** This callback function is different than the type of Callback Function registered using `MC_Register_Callback_Function( )`. See the later discussion of Callback Functions.

Table 12: The MC_Get_Value Family of Functions

Both the `MC_Get_Value` and `MC_Get_Next_Value` function types have several variants depending on the data type of the variable to which you are assigning the retrieved value (see Table 13). These variants can be very helpful because they perform the necessary type conversion from any DICOM value representation to any compatible ANSI-C data type. If a type conversion is not reasonable (e.g., from LT to `short int`) then `MC_INCOMPATIBLE_VR` will be returned as a MC_STATUS code. Also, other error statuses will be returned if the conversion was reasonable but the value stored in the attribute of the message made the conversion impossible (e.g., `MC_INVALID_VALUE_FOR_VR`, `MC_VALUE_OUT_OF_RANGE`,...).

MC_Get function type	Function may be used to retrieve values from attributes with these Value Representations
MC_Get_Value_To_Int	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Get_Value_To_UInt	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Get_Value_To_ShortInt	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Get_Value_To_UShortInt	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Get_Value_To_LongInt	AT, DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Get_Value_To_ULongInt	AT, DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Get_Value_To_Float	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Get_Value_To_Double	DS, FD, FL, IS, SL, SS, UL, US, SQ
MC_Get_Value_To_String	AS, AT, CS, DA, DS, DT, FD, FL, IS, LO, LT, PN, SH, SL, SS, ST, TM, UL, UI, US, UT, SQ

	SS, ST, TM, UI, UL, US, UT, SQ
MC_Get_Value_To_Function	OB, OW, OF, SL, SS, UL, US, AT, FL, FD

Table 13: Acceptable MC_Get_Value/VR combinations

A special purpose function exists in the `MC_Get_Value` family, called `MC_Get_Value_To_Buffer( )`. This function is only used to retrieve the value of an attribute whose value representation is unknown. This occurrence should rarely or never happen but could happen if you needed to parse a message with non-standard attributes. Using `MC_Get_Value_To_Buffer( )` for attributes whose value representations are known will result in an error.

Below is example of parsing attributes from a PATIENT_STUDY_ONLY_QR_FIND - C-FIND-RQ message. This example reads in attributes that may contain query values. The application could use these values to query its own database and send a response message(s) with one or more matches. See the Query Retrieve Sample Application Guides for further details. Again, your application should free the message object when done using it.

```
/*
 * Example of parsing a C-FIND-RQ Message
 * /

/* First is Patient ID */

status = MC_Get_Value_To_String ( FindMessageID,
                                  MC_ATT_PATIENT_ID,
                                  64,
                                  PatientID );

/* Next is Patient name */

status = MC_Get_Value_To_String ( FindMessageID,
                                  MC_PATIENTS_NAME,
                                  64,
                                  szPatientName );


/* Next is patients birth day */

status = MC_Get_Value_To_String ( FindMessageID,
                                  MC_PATIENTS_BIRTH_DATE,
                                  8,
                                  szPatientBday );

/* Finally, patients sex */

status = MC_Get_Value_To_String ( iMessageID,
                                   MC_PATIENTS_SEX,
                                   16,
                                   szPatientSex );

/*
 * Now you can perform a search in your database for
 * matches for the attributes read in, and send the proper
 * response messages.
 */

/* We're through with the message: free it up */

status = MC_Free_Message ( &iMessageID );
```

Retrieving pixel data

When retrieving the value of an attribute with a value representation of OB or OW (e.g., Pixel Data) you can use the `MC_Get_Value_To_Function( )`. Pixel Data tends to be very large and normally you use this function to read the data value a

'chunk' or block at a time. This function is the complement to the MC_Set_Value_From_Function( ) described in the last section.

As an example, your application could define a callback function MyPDStoreFunction( ) whose purpose is to store Pixel Data to an external data sink so that your application uses less primary memory. Pseudo-code for this function follows:

```
MC_STATUS MyPDStoreFunction(int MsgID, unsigned long Tag,
void *UserInfo, int BlockSize, void *DataBlock, int
IsFirstCall, int IsLastBlock)
{
       if (IsFirstCall)
       {
              /*
               * open pixel data sink (e.g., file) here
               * using Tag and/or *UserInfo as a guide.
               */

              IF (OpenFailed)
                     return(MC_CANNOT_COMPLY);

       }

       /*
        * Take this chunk of pixel data from DataBlock
        * and store it to the pixel data sink.
        */

       if (StoreFailed)
             return(MC_CANNOT_COMPLY);

       if (isLastBlock)
       {
              /*
               * close Pixel Data sink here.
               */
       }

       return(MC_NORMAL_COMPLETION);
}
```

This callback function is called by the MergeCOM-3 Library only when triggered by your application. For example, your application might use MyPDStoreFunction to retrieve the value of the MC_ATT_PIXEL_DATA attribute (7FE0, 0010) as follows:

```
MC_Status = MC_Get_Value_To_Function( ImageMsgID,
       MC_ATT_PIXEL_DATA, NULL, MyPDStoreFunction);
```

On making this call, the tool kit library will repetitively callback MyPDStoreFunction( ) until all the pixel data has been read from the message object without any errors. In this case, no user data is passed through to the callback function since *UserData is NULL.

Performance Tuning

Storing or 'setting aside' Pixel Data a block at a time is especially useful for very large Pixel Data and/or on platforms with resource (e.g., memory) limitations. In this case, you would also want to set LARGE_DATA_STORE to the value FILE in the Service Profile, so that MergeCOM-3 Advanced will also maintain the pixel data value stored in the message object itself in a temporary file.

If your application runs on a resource rich system, you should set LARGE_DATA_STORE to the value MEM in the Service Profile, and MergeCOM-3 Advanced will keep the pixel data values in the message object stored in memory

rather than using temporary files. This should improve performance. Also, in this case you may want your callback function to store the Pixel Data in fewer big blocks (or one large block) and keep them in primary memory for rapid access. The `OBOW_BUFFER_SIZE` parameter specifies the size of the data blocks that will be supplied by the library in each call to your callback function.

Once again, while `MyPDStoreFunction( )` is a callback function in this example, it is not what is being referred to when we discuss Callback Functions (with a capital 'C' and capital 'F') in MergeCOM-3 Advanced. Callback Functions are discussed later.

8-bit Pixel Data

For DICOM's Implicit VR Little Endian transfer syntax, the pixel data attribute's `(7fe0,0010)` VR is specified as being OW (independent of what the bits allocated and bits stored attributes are set to). To reduce confusion, MergeCOM-3 Advanced sets the VR of pixel data for the other non-encapsulated transfer syntaxes to OW.

When retrieving or setting pixel data with the `MC_Get_Value_To_Function()` and `MC_Set_Value_From_Function()` calls, the tool kit assumes that the OW pixel data is encoded in the host system's native endian format as defined by DICOM. Figure 10 describes how 8-bit pixel data is encoded in an OW buffer for both big and little endian formats.

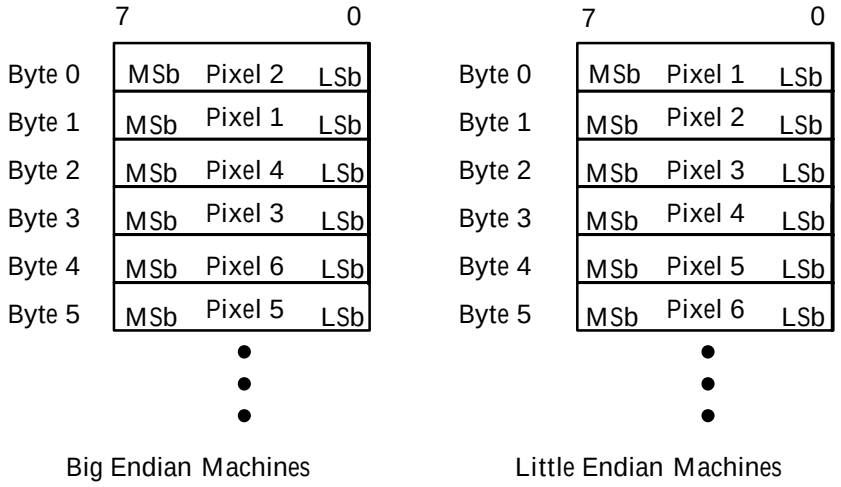

Figure 10: Sample Pixel Data Byte Streams for 8-bits Allocated, 8-bits Stored and a High bit of 7 (VR = OW)

The DICOM standard specifies that the first pixel byte should be set to the *least significant byte* of the OW value. The next pixel byte should be set to *the most significant byte* of the OW value. This implies that on big endian machines, 8-bit pixel data is byte-swapped from the OB encoding method. To make dealing with 8-bit pixel data easier on big endian machines, the tool kit has the function `MC_Byte_Swap_OBOW( )`. This function byte swaps OW data word by word. This function can be called after setting or before retrieving pixel data.

Encapsulated Pixel Data

MergeCOM-3 Advanced Toolkit supports handling of single frame and multi-frame pixel data in encapsulated transfer syntaxes, dealing with it in the same manner as standard pixel data, using the following calls:

```
MC_Set_Encapsulated_Value_From_Function( )
MC_Set_Next_Encapsulated_Value_From_Function( )
MC_Close_Encapsulated_Value( )
MC_Get_Encapsulated_Value_To_Function( )
MC_Get_Next_Encapsulated_Value_To_Function( )
```

MergeCOM-3 will encode supplied data in an encapsulated format and generate the basic offset table. MergeCOM-3 will provide data without the encapsulation delimiters. The data can be compressed or decompressed using a registered compression callback. (Registration of compression callbacks is described later in this document.) Compression libraries are also included on several platforms, including Windows, Sun Solaris and Linux.

An example of encapsulated pixel data is illustrated in Table 14.

Pixel Data Element									
Basic Offset Table with NO Item Value		First Fragment (Single Frame) of Pixel Data			Second Fragment (Single Frame) of Pixel Data			Sequence Delimiter Item	
Item Tag	Item Length	Item Tag	Item Length	Item Value	Item Tag	Item Length	Item Value	Sequence Delim. Tag	Item Length
(FFFE, E000)	0000 0000H	(FFFE, E000)	0000 04C6 H	Compressed Fragment	(FFFE, E000)	0000 024A H	Compressed Fragment	(FFFE, E0DD)	0000 0000H
4 bytes	4 bytes	4 bytes	4 bytes	04C6H bytes	4 bytes	4 bytes	024A H bytes	4 bytes	4 bytes

Table 14: Sample Encapsulated Pixel Data

As specified by the DICOM standard, the various elements shown in Table 14, excluding the compressed pixel data fragments, are encoded in little endian format. The compressed pixel data fragments are treated as OB, and thus do not have an endian. Figure 11 contains the sample pixel data of Table 14 in little endian format.

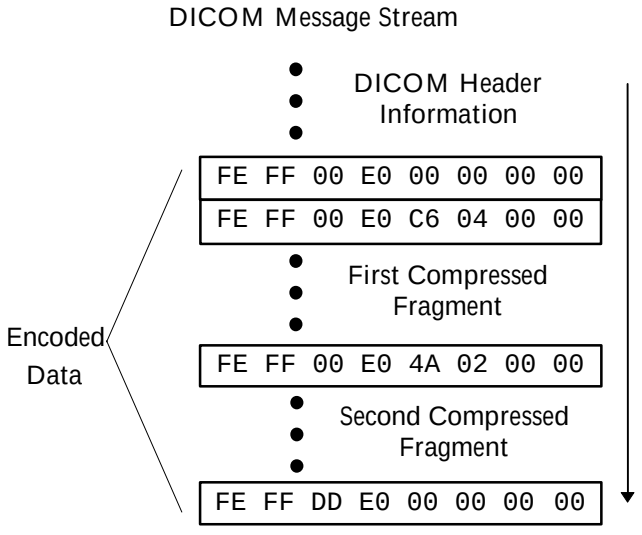

Figure 11: Example Encoding of Encapsulated Pixel Data in Table 14

Further examples of encapsulated pixel data encoding are contained in part 5 of the DICOM standard.

Icon Image Sequences

The Icon Image Sequence can contain small "thumbnail" images. This sequence also contains the Pixel Data Tag (7FE0,0010) just like the main message. Because this may or may not be compressed, some special considerations are necessary.

Sending on the network, writing a file, or writing a stream:

1) Message is uncompressed, Icon is uncompressed
There is no user intervention required. An association will negotiate one of the unencapsulated transfer syntaxes, and both the image and the icon_image will be sent as the negotiated unencapsulated transfer syntax.

2) Message is compressed, Icon is uncompressed
There is no user intervention required. An association will negotiate one of the encapsulated transfer syntaxes, and the image will be sent in this encapsulated transfer syntax, and the icon_image will be sent EXPLICIT_LITTLE_ENDIAN. EXPLICIT_LITTLE_ENDIAN is the default syntax for the non-pixel data portion of a message (including nested pixel data) when the "main" pixel data is encapsulated.

3) Message is compressed, Icon is compressed
Minor intervention is required.

Special creation of icon:
The only difference is `MC_Set_Message_Transfer_Syntax(sqID, <encapsulated transfer syntax>)` must now be called upon creation of the `ICON_IMAGE` item so that you may register compression callbacks and utilize them.

Reading from network, a file, or a stream:

No special conditions are required if the image is streamed in via `MC_Open_File( )`, `MC_Read_Message( )`, or `MC_Stream_To_Message( )`. The sequence item automatically assumes:

`EXPLICIT_LITTLE_ENDIAN` if the pixel data contained in `ICON_IMAGE` is of defined length
-OR-
transfer syntax of the parent if the pixel data contained in `ICON_IMAGE` is of undefined length.

MC_Duplicate_Message:

If duplicating from unencapsulated to an encapsulated (compressed) transfer syntax, then the icon will be unencapsulated by default. To change the behavior, set `DUPLICATE_ENCAPSULATED_ICON` to Yes in the mergecom.pro. This can be done dynamically via `MC_Set_Bool_Config_Value( )`.

Validating Messages

Once your application has a populated message object, either one that you have built or one that you have received and are about to parse, MergeCOM-3 Advanced supplies advanced DICOM message validation functionality. The `MC_Validate_Message( )` function will validate the specified message object instance against the DICOM Standard's specification for that service-command pair.

A file object validation function, `MC_Validate_File( )`, also exists in the Advanced Toolkit and will be discussed further in a later section.

message.txt can be very useful

Another file supplied with MergeCOM-3 Advanced is the message.txt file. This file contains a listing of all the messages supported by the tool kit and the parameters they are validated against. message.txt is a useful guide in your application development because it specifies the attributes that can make up the object instance portion of each message type (service-command pair) and is often easier to use as a quick reference than paging through two or three parts of the DICOM Standard. message.txt also specifies the contents of items and files (see discussions of Sequence of Items and DICOM Files later in this document). Remember though that the DICOM Standard is the final word and that message.txt has its limitations as described further below.

MC_Validate_Message( ) does not validate the attributes that make up the command portion of a DICOM message. Command attributes (attributes with a group number less than 0008) are also not specified in message.txt. The MergeCOM-3 Advanced Library sets as many of the command group attributes as possible automatically. In some services, your application will need to set command attributes (e.g., the 'Affected SOP Class UID' attribute (0000,0002) in the C-MOVE response message). These special cases are described further in the Application Guides and in Part 7 of the DICOM Standard.

An excerpt of message.txt follows for the service-command pair DETACHED_ PATIENT_ MANAGEMENT - N_ GET_ RSP as an illustration. For each attribute in the message, at least one line of data is specified. This first line includes the tag, attribute name, value representation, and value type. Additional lines may be included for the attribute to list conditions, enumerated values, defined terms, and item names for attributes with a VR of SQ. You should refer to the DICOM Standard (parts 3 and 4) for a detailed description of particular conditions and their meanings.

```
################################################################
DETACHED_PATIENT_MANAGEMENT - N_GET_RSP
################################################################

0008,0005    Specific Character set                        CS    1C
Condition: EXPANDED_OR_REPLACEMENT_CHARACTER_SET_USED
Defined Terms:     ISO-IR 100, ISO-IR 101, ISO-IR 109, ISO-IR 110,
ISO-IR144, ISO-IR 127, ISO-IR 126, ISO-IR 138, ISO-IR 148
0008,1110    Referenced Study Sequence                     SQ    2
Item Name(s): REF_STUDY
0008,1125    Referenced Visit Sequence                     SQ    2
Item Name(s):
0010,0010    Patient's Name                                PN    2
0010,0020    Patient ID                                    LO    2
0010,0021    Issuer of Patient ID                          LO    3
0010,0030    Patient's Birth Date                          DA    2
0010,0032    Patient's Birth Time                          TM    3
0010,0040    Patient's Sex                                 CS    2
Enumerated Values: M, F, O
0010,0050    Patient's Insurance Plan Code Sequence        SQ    3
Item Name(s): PATIENTS_INSURANCE_PLAN_CODE
0010,1000    Other Patient IDs                             LO    3
0010,1001    Other Patient Names                           PN    3
0010,1005    Patient's Birth Name                          PN    3
0010,1020    Patient's Size                                DS    3
0010,1040    Patient's Address                             LO    3
0010,1060    Patient's Mother's Birth Name                 PN    3
0010,1080    Military Rank                                 LO    3
0010,1081    Branch of Service                             LO    3
0010,1090    Medical Record Locator                        LO    3
0010,2000    Medical Alerts                                LO    3
0010,2110    Contrast Allergies                            LO    3
0010,2150    Country of Residence                          LO    3
0010,2152    Region of Residence                           LO    3
0010,2154    Patient's Telephone Numbers                   SH    3
0010,2160    Ethnic Group                                  SH    3
0010,21A0    Smoking Status                                CS    3
Enumerated Values: YES, NO, UNKNOWN
0010,21B0    Additional Patient History                    LT    3
0010,21C0    Pregnancy Status                              US    3
```

```
Enumerated Values: 0001, 0002, 0003, 0004
0010,21D0    Last Menstrual Date                          DA    3
0010,21F0    Patient's Religious Preference               LO    3
0010,4000    Patient Comments                             LT    3
0038,0004    Referenced Patient Alias Sequence            SQ    2
Item Name(s): REF_PATIENT_ALIAS
0038,0050    Special Needs                                LO    3
0038,0500    Patient State                                LO    3
```

What validation can do for you...

While MergeCOM-3's validation is not foolproof, it is very useful and will catch many standard violations. It validates the following:

- That the value assigned to an attribute is appropriate for that attributes VR.
- That all value type 1 attributes have a value, and that value is not null.
- That all value type 2 attributes have a value, and that value may be null.
- That a specified set of conditional attributes (value type 1C or 2C) are validated as value type 1 or 2 attributes when the specified condition is satisfied. See Table 15 for the conditions supported by the advanced tool kit. Not all conditions can be validated by the tool kit and those that cannot need to be checked by the application itself.
- That an attribute does not have too many or too few values for its specified value multiplicity.
- That an attribute that has enumerated values does not have a value that is not one of the enumerated values. A warning is also issued if an attribute that has defined terms has a value that is not one of those defined terms.
- That a non-private attribute is not included in the message that is not defined for that DICOM message (service-command pair).

Condition Name
ASPECT_RATIO_NOT_1_TO_1_AND_IMAGE_PLANE_MODULE_NOT_APPLICABLE_TO_IMAGE
COLLIMATOR_SHAPE_IS_CIRCULAR
COLLIMATOR_SHAPE_IS_POLYGONAL
COLLIMATOR_SHAPE_IS_RECTANGULAR
CT_MR_NM_IMAGES
CURVE_DATA_DESCRIPTOR_IS_USED
DEVICE_DIAMETER_PRESENT
DIMENSIONS_ARE_DESCRIBED_BY_INTERVAL_SPACING
EXCEPT_WHEN_SCANNING_SEQUENCE_IS_EP_AND_SEQUENCE_VARIENT_NOT_SK
EXPOSURE_IS_NOT_PRESENT
EXPOSURE_TIME_OR_XRAY_TUBE_CURRENT_NOT_PRESENT

EXPOSURE_TIME_OR_XRAY_TUBE_CURRENT_NOT_PRESENT
FRAME_INCREMENT_POINTER_CONTAINS_TAG_FOR_ANGULAR_VIEW_VECTOR
FRAME_INCREMENT_POINTER_CONTAINS_TAG_FOR_DETECTOR_VECTOR
FRAME_INCREMENT_POINTER_CONTAINS_TAG_FOR_ENERGY_WINDOW_VECTOR
FRAME_INCREMENT_POINTER_CONTAINS_TAG_FOR_PHASE_VECTOR
FRAME_INCREMENT_POINTER_CONTAINS_TAG_FOR_R_R_INTERVAL_VECTOR
FRAME_INCREMENT_POINTER_CONTAINS_TAG_FOR_ROTATION_VECTOR
FRAME_INCREMENT_POINTER_CONTAINS_TAG_FOR_SLICE_VECTOR
FRAME_INCREMENT_POINTER_CONTAINS_TAG_FOR_TIME_SLICE_VECTOR
FRAME_INCREMENT_POINTER_CONTAINS_TAG_FOR_TIME_SLOT_VECTOR
FRAME_INCREMENT_POINTER_POINTS_TO_FRAME_TIME
FRAME_INCREMENT_POINTER_POINTS_TO_FRAME_TIME_VECTOR
IMAGE_LEVEL_MOVE_OR_ABOVE
IMAGE_LEVEL_QUERY
IMAGE_PART_OF_SERIES_WHICH_DOES_NOT_REQUIRE_IMAGE_PLANE_MODULE
IMAGE_TYPE_VALUE_3_CONTAINS_WHOLEBODY
IMAGE_TYPE_VALUE_3_IS_BIPLANEA_BIPLANEB
IMAGE_TYPE_VALUE_3_IS_TOMO_GATEDTOMO_RECONTOMO_RECONGATEDTOMO
IMAGE_TYPE_VALUE_3_IS_WHOLE_BODY_STATIC
IMAGE_TYPE_VALUE_4_IS_TRANSMISSION
IMAGE_TYPE_VALUE_4_IS_TRANSMISSION_AND_VALUE_3_IS_NOT_TOMO
MASK_OPERATING_IS_AVG_SUB
MASK_OPERATION_IS_TID
MODALITY_LUT_SEQUENCE_NOT_PRESENT
NUMBER_OF_FRAMES_SENT
OVERLAY_DATA_IN_GROUP
PHOTOMETRIC_INTERPRETATION_HAS_VALUE_PALETTE_COLOR_OR_ARGB
PIXEL_COMPONENT_CALIBRATION_EXISTS
PIXEL_COMPONENT_ORGANIZATION_EXISTS
PIXEL_COMPONENT_ORGANIZATION_IS_BIT_ALIGNED
PIXEL_COMPONENT_ORGANIZATION_IS_RANGES

POSITIONER_MOTION_IS_DYNAMIC
PRIVATE_DIRECTORY_RECORD
REFERENCED_SOP_INSTANCE_UID_ABSENT
RESCALE_INTERCEPT_IS_PRESENT
RETRIEVE_AE_TITLE_NOT_PRESENT
SAMPLES_PER_PIXEL_HAS_VALUE_GREATER_THAN_1
SCAN_OPTIONS_WHICH_INCLUDE_HEART_GATING
SCANNING_SEQUENCE_HAS_VALUES_IR
SERIES_LEVEL_MOVE_OR_ABOVE
SERIES_LEVEL_QUERY_OR_BELOW
SHUTTER_SHAPE_IS_CIRCULAR
SHUTTER_SHAPE_IS_POLYGONAL
SHUTTER_SHAPE_IS_RECTANGULAR
STORAGE_MEDIA_FILE_SET_NOT_PRESENT
STUDY_LEVEL_MOVE_OR_ABOVE
STUDY_LEVEL_QUERY_OR_BELOW
TABLE_MOTION_IS_DYNAMIC
TRIGGER_VECTOR_USED
WINDOW_CENTER_SENT

Table 15: Conditions supported by MergeCOM-3 Advanced

and what validation cannot do for you.

As mentioned, MergeCOM-3 Advanced does not capture all standard violations, and the DICOM Standard itself should be considered the final word when validating a message. Important limitations of MergeCOM-3 validation include:

- DICOM Part 3 specifies Information Object Definitions (IOD's) as being composed of modules. Each module contains attributes. Only in the case of composite IOD's may an attribute be specified in DICOM Part 3 as being contained in either a User Optional or Conditional Module. MergeCOM-3 Advanced treats all such attributes as being value type 3 (optional).
- Also, only in the case of composite IOD's (e.g., Ultrasound Image Object) used in storage services, may certain modules be mutually exclusive (e.g., curve and overlay modules). The attributes defined in these modules are all treated as type 3.
- For normalized services using the N-EVENT-REPORT command, the actual contents of an N-EVENT-REPORT message are dependent on the Event Type ID being communicated. MergeCOM-3 advanced treats all Event Type ID's identically when performing message validation; namely it treats all attributes as type 3.

An example of the use of MC_Validate_Message( ) follows

An example...

```
status = MC_Validate_Message (iMessageID, &error_info,
      Validation_Level1);
if (status == MC_DOES_NOT_VALIDATE)
{
      printf("MC_Validate_Message tag: %lx error: %s",
            error_info->Tag,
            MC_Error_Message(error_info->Status));
      /*
       * you may want to abort the association in this
       * case as follows:
       *
       *           MC_Abort_Association(&associationID);
       */
}
```

In this example, the application validates the message object iMessageID at Validation_Level1 that reports only errors. Validation_Level2 could be used to report both warnings and errors, while Validation_Level3 could be used to report errors, warnings, and informational messages. If the status returned is anything other than MC_MESSAGE_VALIDATES, your application can look at the error_info structure passed back to decipher the violation. error_info is defined as follows:

```
/* ======================================================= *
 *         Structure containing Message Validation         *
 *                    Error Information                    *
 * ======================================================= */
typedef struct ValErr_struct
{
      unsigned long       Tag;         /* Tag with
                                          validation error */
      int                 MsgItemID;   /* ID of message or
                                          item object
                                          containing the tag*/
      int                 ValueNumber; /* Value number
                                          involved - zero
                                          if error is not
                                          value related */
      MC_STATUS           Status;      /* Error status code
                                          */
} VAL_ERR;
```

In this source, the offending attribute's tag is printed out along with the error string associated with Status. If the image object violates DICOM the output to standard output in this example might look like the following:

example output

```
MC_Validate_Message tag: 101010 error:
                  Invalid value for this tag's VR
```

This output states that the attribute (0010,1010) in the message has a value that violates the value representation for that attribute. If the application wished to go on to find other errors in the same message, it would call MC_Get_Next_Validate_Error( ) until the status returned equals MC_END_OF_LIST, meaning that no more errors exist in the message.

It is on the initial call to MC_Validate_Message( ) that all the validation takes place and that the results of the validation for the entire message are logged to the message log file. Subsequent calls to MC_Get_Next_Validate_Error( ) simply step through the results of the validation, passing additional errors found back to the application. In the above example the message log file contains the following report:

example log file

```
01-11 13:52:09.00 7919 MCserver(process_messages) T4: Processing a
C_STORE_RQ request
01-11 13:52:09.00 7919 MC3 T5: (0008,0005) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0008,0023) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0008,0033) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0010,1010) VE: [41Y ] Invalid value
for this tag's VR
01-11 13:52:09.00 7919 MC3 T5: (0018,0010) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0015) VE: Required attribute
has no value
01-11 13:52:09.00 7919 MC3 T5: (0018,0020) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0021) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0022) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0023) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0050) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0080) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0081) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0082) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0084) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0085) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,0091) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,1041) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,1060) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,1250) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0018,5101) VE: Required attribute
has no value
01-11 13:52:09.00 7919 MC3 T5: (0020,0032) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0020,0037) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0020,0052) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0020,0060) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0020,1040) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0020,1041) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0028,0006) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0028,0030) VW: Invalid attribute
for service
01-11 13:52:09.00 7919 MC3 T5: (0028,0034) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0028,1101) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0028,1102) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0028,1103) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0028,1201) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0028,1202) VI: Unable to check
condition
01-11 13:52:09.00 7919 MC3 T5: (0028,1203) VI: Unable to check
condition
```

Notice in this log file that all warnings and informational messages are also logged. This is always the case, although the first violation returned to the application was

an error because `Validation_Level1` was specified. The message log agrees in that the first VE (Validation Error) logged is for the attribute Patient's Age (0010,1010). The log states that the message contains "`41Y `" as the value for this attribute. Part 6 of DICOM clearly states that this attribute has a value representation of AS (Age String) and part 5 states that for this VR the value should have a leading zero and be represented as "`041Y`". There is also one other error flagged in this message. The required attribute View Position (0018,5101) had no value.

Many more details relating to the usage and behavior of `MC_Validate_Message( )` and `MC_Get_Next_Validate_Error( )` can be found in the Reference Manual.

Performance Tuning

DICOM message validation does involve processing overhead. The most significant overhead is in the accessing of the message info files, and significantly less overhead is involved in actually validating the contents of the message structure. It is **important** to understand that depending on the way in which your message object was created, this validation overhead can occur at different points in your application; see Table 16.

Message Object Creation Method	Point at which file access overhead for validation occurs
`MC_Open_Message( )`	MC_Open_Message( )
`MC_Open_Empty_Message( )`	MC_Validate_Message( ) **Note:** You must use MC_Set_Service_Command( ) before validating and/or sending a message created in this manner.
`MC_Read_Message( )`	MC_Validate_Message( )

Table 16: Point of performance overhead associated with message validation.

Using `MC_Open_Message( )` has an up-front performance cost but provides additional validation as you set the value of attributes in the message object. For the other two creation methods, the cost occurs on validation itself.

Many times the `MC_Validate_Message( )` is selectively used in an application: as a runtime option or conditionally compiled into the source code. Validation might only be used during integration testing or in the field for diagnostic purposes. Reasons for this include performance since the overhead associated with message validation may be an issue, especially for larger messages having many attributes or on lower-end platforms. Also, validation can clutter the message log with warnings and errors that may not be desirable in a production environment. Performance issues related to message handling are discussed further under Message Exchange later in this document.

Streaming Messages

When DICOM messages are exchanged over a network, they are in an encoded format specified by the DICOM standard and the negotiated transfer syntax. MergeCOM-3 Advanced calls this encoded format a **message stream** and supplies powerful functions that allow your applications to work directly with message streams.

When your application builds or parses messages as described earlier, it works with a MergeCOM-3 Advanced message object. This message object abstracts and

encapsulates the DICOM message and hides its details from the developer. When you send the DICOM message object over the network, MergeCOM-3 internally creates a DICOM message stream that is passed over the network. This message stream is an encoded stream of bytes that follows all the rules of DICOM.

MergeCOM-3 Advanced also supplies function calls to the developer to generate and read DICOM message streams directly (see Figure 12). `MC_Message_To_Stream( )` converts a message object to a message stream, while `MC_Stream_To_Message( )` converts a message stream into a message object. Also, `MC_Get_Stream_Length( )` is supplied to calculate the length of the DICOM stream that would result from using the `MC_Message_To_Stream( )` call.

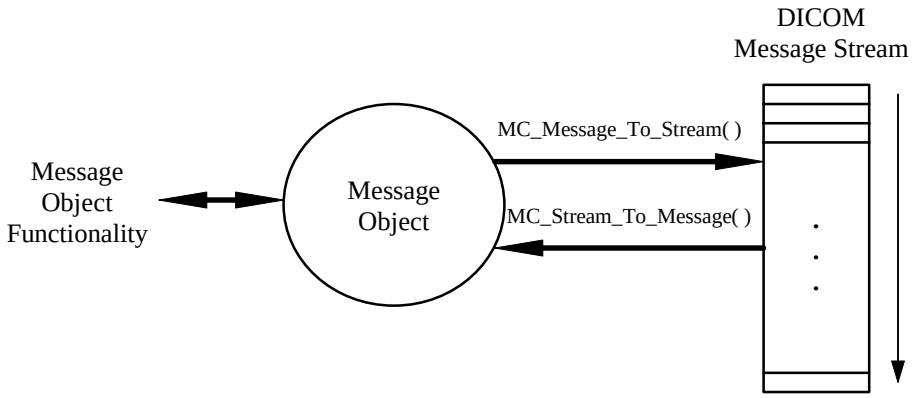

Figure 12: Relationship between a Message Object and a Message Stream

A call to `MC_Message_To_Stream( )` could look like the following:

```
Status = MC_Message_To_Stream(MyMessageID, 0x00080000,
0x7FDFFFFF, EXPLICIT_LITTLE_ENDIAN, NULL, MyStreamHandler);
```

This call converts the attributes from (0008,0000) through (7FDF,FFFF) in the message object identified by `MyMessageID` into a DICOM message stream using the explicit little endian transfer syntax. Explicit little endian transfer syntax is one of the three DICOM Transfer Syntaxes supported by MergeCOM-3 Advanced. DICOM defines two other transfer syntaxes: implicit little endian (the default DICOM transfer syntax) and explicit big endian. See Part 5 of the DICOM Standard for a detailed description of transfer syntaxes.

`MyStreamHandler( )` is a callback function you supply in your application that receives and manages the stream data a block at a time. This callback function is similar to the example in `MC_Get_Value_To_Function( )` except that it handles a message stream rather than Pixel Data. See the API description in the Reference Manual for further details.

Once your application has done the above and stored the stream somewhere, you could later rebuild a message object containing only group 0008 using:

```
Status = MC_Stream_To_Message(MyMessageID, 0x00080000,
0x0008FFFF, EXPLICIT_LITTLE_ENDIAN, NULL, MyStreamProvider);
```

This call converts only the attributes in group 0008 of the stream supplied by your callback function `MyStreamProvider( )` and places them in the message identified by `MyMessageID`. It is important that the transfer syntax specified in this call is identical to that used to create the stream or the call will fail with an error.

Performance Tuning

The same kind of performance issues apply in the callback functions discussed above as in those callbacks used with `MC_Get_Value_To_Function( )` and `MC_Set_Value_From_Function( )`. Namely, your settings of `LARGE_DATA_STORE` and `OBOW_BUFFER_SIZE` should take into consideration the capabilities of your platform.

Message streams can be very valuable to your application for debugging and validation purposes. By writing DICOM message streams out to a binary file, you have a compact and reproducible representation of a message. You can directly examine the binary message stream to see how the data would be sent over the network. Also, you can read this binary file in again later to reconstruct the original message object. Once you have the message object you can use the usual tool kit functions to examine or alter its contents.

deflated streams

The transfer syntax, Deflated Explicit VR Little Endian, gives you the ability to use the "deflate" algorithm to compress the entire data set. This transfer syntax was added mostly for structured reports, which are extremely redundant in their encoding, with considerable repetition of strings and tags. The toolkit uses zlib to implement deflate/inflate. This is an open source library that is built into the toolkit. Messages of this transfer syntax are still stored as message objects while the toolkit is handling them. Only when a message is "streamed" is the message deflated/inflated.

Message Exchange (Network Only)

General

We have discussed how associations are managed as well as how messages objects are populated and parsed. Now we discuss how these DICOM messages are exchanged with other application entities over the network.

The exchange of DICOM messages between AE's only occurs over an open association. After the DICOM client (SCU) application opens an association with a DICOM server (SCP), the client sends request messages to the server application. For each request message, the client receives back a corresponding response from the server. The server waits for a request message, performs the desired service, and sends back some form of status to the client in a response message. This process, along with the corresponding MergeCOM-3 Advanced Function calls, are pictured in Figure 13.

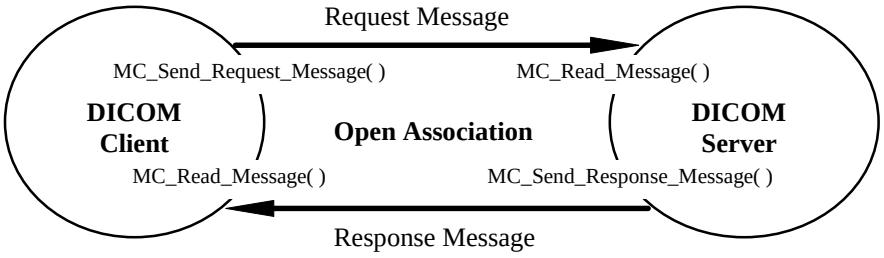

Figure 13: Message Exchange in MergeCOM-3 Advanced Applications

These three calls have the following form:

```
status = MC_Send_Request_Message(AssociationID,
RequestMessageId);

status = MC_Send_Response_Message(AssociationID,
ResponseStatus, ResponseMessageID);

status = MC_Read_Message(AssociationID, Timeout, &MessageID,
&ServiceName, &Command);
```

sending messages

The parameters to the `MC_Send_Request_Message( )` and `MC_Send_Response_Message( )` include an `AssociationID` identifying the open association over which the message is to be sent and a `MessageID` identifying the message object to be sent. The `MC_Send_Response_Message( )` call includes one additional parameter, `ResponseStatus`, that must be set to a valid DICOM response status (#defined in mergecom.h). Example response status codes for the N_GET_RSP response message are summarized in Table 17. Response codes for other DICOM commands are described in the appropriate Applications Guides and in Part 4 of the DICOM Standard.

N_GET_RSP Status Codes
N_GET_SUCCESS
N_GET_WARNING_OPT_ATTRIB_UNSUPPORTED
N_GET_ATTRIBUTE_LIST_ERROR
N_GET_CLASS_INSTANCE_CONFLICT
N_GET_DUPLICATE_INVOCATION
N_GET_MISTYPED_ARGUMENT
N_GET_NO_SUCH_SOP_CLASS
N_GET_NO_SUCH_SOP_INSTANCE
N_GET_PROCESSING_FAILURE
N_GET_RESOURCE_LIMITATION
N_GET_UNRECOGNIZED_OPERATION

Table 17: Valid Response Message Status Codes for an N-GET Command

receiving messages

When your application makes an `MC_Read_Message( )` call, the `AssociationID` parameter specifies the association over which you wish to read the message. The `MessageID` returned to you identifies the message received. The messages `Service` and `Command` are also returned to your application to aid it in further processing.

The other important parameter in an `MC_Read_Message( )` call is `Timeout`. `Timeout` specifies, in seconds, how long your process will wait for a message before the `MC_Read_Message( )` call times out and returns control to your application code. If your application is running in a multi-tasking environment, your process will be blocked during this waiting period and the system processor

will be available for other processes. Setting `Timeout` to 0 is equivalent to polling, since `MC_Read_Message( )` returns immediately, whether a message has been received or not. A `Timeout` of -1 indicates wait forever, or until a message arrives, before returning.

Using `select( )` to handle asynchronous events

In specialized cases where the application reading the request or response message is waiting on several asynchronous events, not just the message event, the `MC_Get_Association_Info( )` call can be used to get the file descriptor for the socket over which message exchange will occur. The `select( )` system call can then be used to wait asynchronously for a DICOM request or response message. When select returns on the DICOM message file descriptor, `MC_Read_Message( )` can be called and will return immediately with the received message.

Your application may want to take advantage of MergeCOM-3's message validation functionality before sending a DICOM message out on the network, or before parsing and acting on a message received from some other device. Also, when constructing a request or response message, it is important to note that for some services, your application will need to set the value of command attributes in the message. Refer back to the *Validating Messages* section of this document for further discussion.

Asynchronous Communications

The DICOM standard defines an optional method for negotiation of an Asynchronous Operations Window. The Asynchronous Operations Window allows the client and server during association negotiation to define how many request messages can be sent over an association before a response message is required to be received. When the Asynchronous Operations Window is not negotiated (the default behavior of MergeCOM-3) only one request message can be sent before a response is received. Use of asynchronous operations can improve network performance when transferring a large number of messages over an association.

The specific fields negotiated over an association are the maximum number of operations, sub-operations, or notifications *invoked* by the requester of the association and the maximum number of operations, sub-operations, or notifications *performed* by the requester of the association. The client proposes settings for both of these fields, and the server responds with values less than or equal to the proposed values that are then used for the association.

The term *notifications* refers to `N-EVENT-REPORT` messages that are sent from an SCP to an SCU. These messages are used by DICOM services such as Print Job and Storage Commitment. The terms *operations* and *sub-operations* refer to all other message types. The term *sub-operations* specifically refers to services such as Query/Retrieve where multiple response messages are sent for a single request message.

Asynchronous definitions

For a client negotiating the SCU role, the *invoked* field refers to the number of *operations* that could be sent without receiving a response message and the *performed* field would specify the maximum number of *notifications* received before the client is required to send a response message. For a client negotiating the SCP role and an asynchronous window, the *invoked* field would refer to the maximum *notifications* sent before receiving a response and the *performed* field would refer to the maximum *operations* received before the client is required to send a response.

Performance Tuning

Although asynchronous operations can be used for all DICOM service classes, this feature is most useful with the Storage Service Class. Asynchronous operations can

be utilized to improve the network performance of transferring a large number of C-STORE messages over an association. During normal synchronous operations, there typically is no network activity while a Storage SCU waits for a response message from an SCP. Because there is a time when no data is being sent from the SCU to the SCP, a network is typically underutilized by synchronous DICOM transfers. Also, sending a large number of small images typically is slower than sending a smaller number of large images because a higher percentage of the association time is spent waiting on response messages. Figure 14 illustrates how an SCU waits on a response from the SCP while the SCP processes a message.

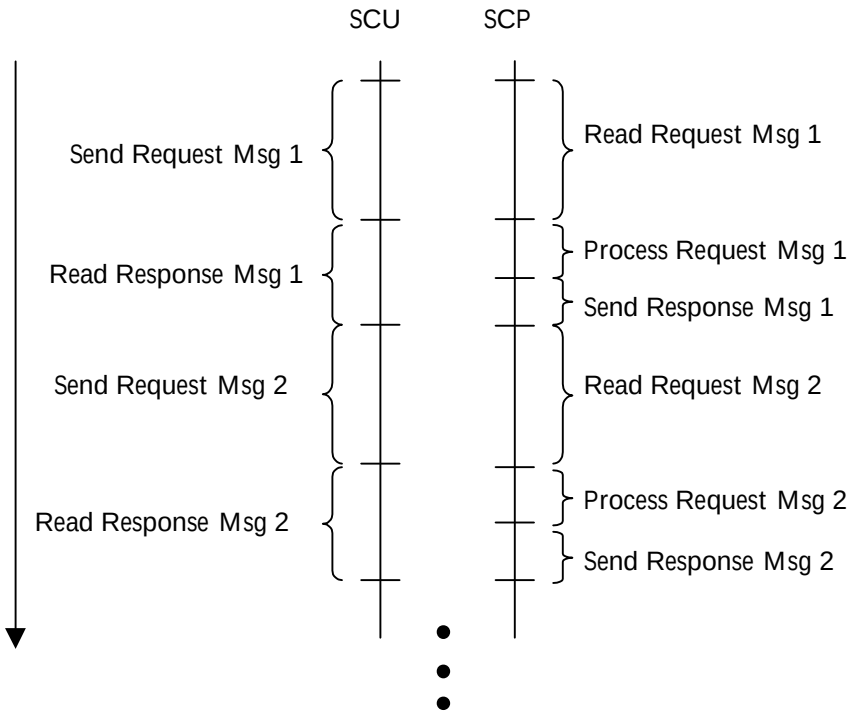

Figure 14: Timing of Message Exchange During Synchronous Transfers

When asynchronous operations are negotiated, the SCU can poll for a response message, but if a message is unavailable, it can start sending the next request message right away (if the max operations will not be exceeded). This allows an SCU to more fully utilize the network bandwidth. There is only a small time when the SCU polls for a response message during which data is not being sent from the SCU to the SCP.

The majority of changes required to implement asynchronous communications are on the SCU side. A traditional SCP can effectively support asynchronous communications by simply enabling its negotiation over associations. It is possible, however, to do further optimization of SCP applications. On operating systems that MergeCOM-3 supports threading, an SCP can be written to process messages in the background as it is reading request messages and to send response messages over the association when processing is completed. In this scenario, after reading a message, the SCP would pass the received message to another thread for processing and freeing. The main SCP thread would then go to reading the next message. Once processing is complete for a message, the background thread would signal the main thread to send a response message for the request. This allows the network bandwidth to be more fully utilized by having the SCP reading data off the network as much as possible. The SCP in this case must monitor the negotiated max operations so that it is not exceeded.

Figure 15 shows an example message exchange between an SCU and SCP when using asynchronous communications. This example shows how the SCU spends less time reading the response messages and moves to sending the next request message to increase bandwidth utilization.

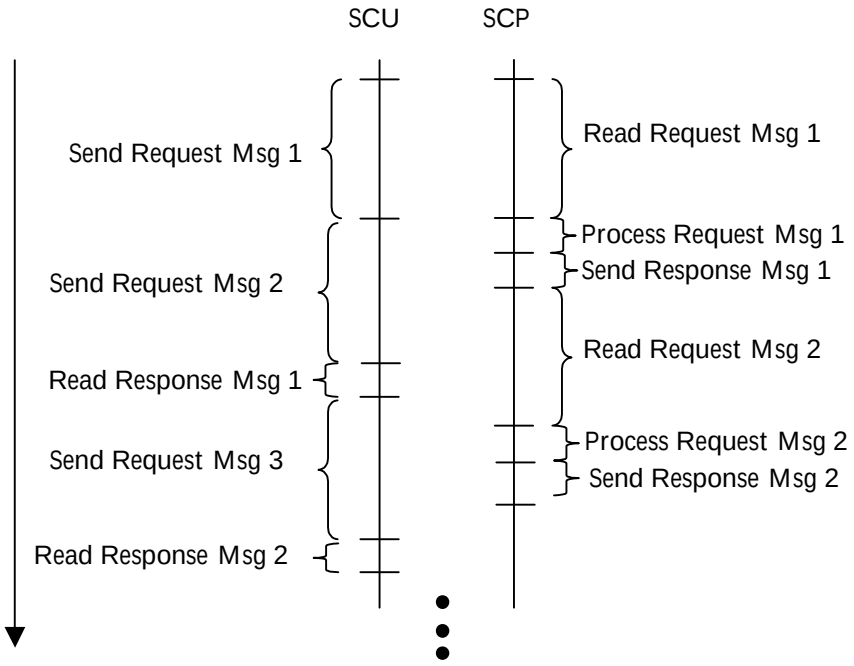

Figure 15: Timing of Message Exchange During Asynchronous Transfers

Message ID must be set for asynchronous communications!

Asynchronous operations are implemented utilizing the standard MergeCOM-3 network functions: `MC_Send_Request_Message( )`, `MC_Send_Response_Message( )`, `MC_Read_Message( )`, `MC_Read_Message_To_Tag( )`, `MC_Continue_Read_Message_To_Tag( )`, and `MC_Continue_Read_Message_To_Stream( )`. One additional requirement, however, is for the application to keep track of the *Message ID* (0000,0110) tag and the *Message ID Being Responded To* (0000,0120) tags. *Message ID* contains an integer uniquely identifying a request message transferred over an association. MergeCOM-3 will automatically insert this tag when sending a request message over the network. *Message ID Being Responded To* is contained in response messages, and contains the *Message ID* that corresponds to the request message that a response is for.

During normal synchronous transfers, MergeCOM-3 keeps track of the *Message ID* of the last request message, and automatically inserts this into the *Message ID Being Responded To* tag when sending a response. For asynchronous operations to work properly, the application is required to keep track of these tags and appropriately fill them in response messages. For applications sending request messages, after a call to `MC_Send_Request_Message( )`, the application should retrieve the *Message ID* tag from the message, and save it until a response message has been received. For applications sending response messages, the *Message ID* should be retrieved out of the request message, and set in the *Message ID Being Responded To* tag for the response message when a response is sent.

Service lists are utilized to configure the *invoked* and *performed* operations for an SCU and SCP. The configuration of service lists is discussed in the previous section of this document describing the Application Profile configuration file. The `MC_Get_Association_Info( )` function can be utilized by an SCU or SCP application to determine the asynchronous operations *invoked* and *performed* that

were negotiated for an association. MergeCOM-3 also keeps track of the asynchronous operations negotiated and will prevent the user from exceeding the *operations*, *sub-operations*, and *notifications* performed or invoked that were negotiated.

When writing asynchronous applications, care should be taken in keeping track of the resultant status of all request messages that were sent. In particular, if an association is aborted, all outstanding *operations* or *notifications* should be treated as failed and resent at a later time.

Deadlocks possible with asynchronous negotiation!

It is possible to create deadlock situations when writing an asynchronous application. A situation may arise where both the client and server are attempting to send data to each other that would cause a deadlock. In general, applications should poll for incoming messages before sending a new message. In particular, a Storage SCU should poll for a response message before sending a new request message. Because of the size of the messages exchanged, it is most likely that a Storage SCU would deadlock if it were not checking for response messages.

Performance Tuning

Finally, the configuration options `TCPIP_SEND_BUFFER_SIZE` and `TCPIP_RECEIVE_BUFFER_SIZE` are important for maximizing network performance. The send buffer specifies the amount of data that can be queued in the TCP/IP stack of the OS without actually being sent. The receive buffer specifies the amount of data that can be received by the TCP/IP stack of the OS before a MergeCOM-3 application must start reading the data. These options allow data to be queued by the OS in the background while a MergeCOM-3 application is doing other activities. For instance, an entire response message can usually be stored in a the send buffer and a call to `MC_Send_Response_Message()` may return before any data has even been sent over the network. Similarly, an application calling `MC_Send_Request_Message( )` will return before all data has been sent. This allows the application to start preparing the next message to be sent while data is still being transferred. The maximum settings for these options is operating system dependent. It is suggested that these options be configured to the maximum setting for an operating system. If the settings are too high, an error will be logged to the merge.log file.

Use Full Duplex Networks with asynchronous commnications!

It is important that devices using asynchronous communications be configured to use full duplex network connections. Using asynchronous communications in half duplex mode would greatly degrade performance. Performance would more than likely be lower than if only synchronous communications were used.

Using Compression/Decompression Callback Functions

The purpose of registering a compression/decompression callback is to more easily support compressed transfer syntaxes in DICOM. Compressed transfer syntaxes are used to take advantage of the decreased image size that goes along with compressed pixel data. This is important when dealing with large images or series of images that need to be stored or transmitted across a network.

The callbacks, when registered, are utilized any time the functions in Table 18 are called, *and* the transfer syntax of a message has been set to `JPEG_BASELINE`, `JPEG_EXTENDED_2_4`, `JPEG_LOSSLESS_HIER_14`, `JPEG_2000`, `JPEG_2000_LOSSLESS_ONLY`.

Function	Callback Utilized
`MC_Set_Encapsulated_Value_From_Function()`	Compressor
`MC_Set_Next_Encapsulated_Value_From_Function()`	Compressor

MC_Get_Encapsulated_Value_To_Function()	Decompressor
MC_Get_Next_Encapsulated_Value_To_Function()	Decompressor

Table 18: Callbacks Utilized by Functions that set and get Pixel Data

If there are no callbacks registered, the above functions will work the same as MC_Get_Value_To_Function( ) and MC_Set_Value_From_Function( ), except encapsulation delimiters will be removed and inserted, respectively.

MC_Register_Compression_Callbacks( ) is used to register the callback functions that take in pixel data, and return a compressed image, or take in compressed data, and return the uncompressed pixel data. You may provide your own function(s) to do this, or you may use the built in compressor and decompressor supplied by MergeCOM-3.

How to register a Compression Callback Function

The MC_Register_Compression_Callbacks( ) can be used to register a Callback Function with the tool kit as follows:

```
status = MC_Register_Compression_Callbacks(msgID,
                MyCompressionCallback,
                MyDecompressionCallback);
```

This call registers My(De)compressionCallback( ) with the Advanced Tool Kit library to handle data to be compressed/decompressed for a message object when the above functions are called. Only one compressor and one decompressor may be registered for a message at a time.

User Defined Compressor and/or Decompressor

If you use your own callback functions, both callbacks are prototyped the same. They must be prototyped as follows:

```
MC_STATUS My(De)compressionCallback(
              int       CBMessageID,
              void**    CBContext,
              unsigned long CBdataLength,
              void*     CBdataValue,
              unsigned long* CBoutdataLength,
              void**    CBoutdataValue,
              int       CBisFirst,
              int       CBisLast,
              int       CBrelease);
```

CBMessageID allow a single Callback Function to handle requests for different messages. CBContext is a pointer to a user defined structure that contains data that must be maintained between (de)compression calls because the pixel data may be presented and received over multiple callbacks.

CBdataLength, CBdataValue, CBisFirst, and CBisLast are used to manage the data flow into the callback function. When the callback is called with data, a 'chunk' is being supplied. MergeCOM-3 has set CBdataLength to the number of bytes of data it is providing at CBdataValue. The length of the chunk must be less than INT_MAX.

MergeCOM-3 will set CBisFirst to TRUE (non zero) the first time it is providing data (i.e., when it is providing the first block of data). MergeCOM-3 sets CBisLast to TRUE (non zero) the last time it will be providing data (i.e., when it is providing the last block of data).

The callback function must provide all of the compressed/decompressed data when CBisLast is received. CBoutdataLength should be set to the number of bytes of data the callback is providing at CBoutdataValue.

A call with CBrelease set to TRUE (non zero), should release all memory that the compressor/decompressor has allocated.

SPECIAL NOTE!

The callback should return MC_NORMAL_COMPLETION to indicate success, or MC_CANNOT_COMPLY for any failure.

When decompressing an image without an offset table, the toolkit does not know if it has reached the end of a fragment, or the end of a frame. The decompressor MUST return data whenever it is available so the toolkit can keep track of the image data returned. When the toolkit determines it has received an entire image and the image has been decompressed, the isLast will be set, and no data will be passed to the callback.

Built in Compressor and Decompressor

If using the built in compressor or decompressor, the callbacks should be registered as follows:

```
status = MC_Register_Compression_Callbacks(msgID,
                MC_Standard_Compressor,
                MC_Standard_Decompressor);
```

The MergeCOM-3 built in compressor and decompressor utilize libraries from Pegasus Imaging Corporation (www.jpg.com). They support the JPEG_BASELINE, JPEG_EXTENDED_2_4, JPEG_LOSSLESS_HIER_14, JPEG_2000, JPEG_2000_LOSSLESS_ONLY transfer syntaxes with photometric interpretations MONOCHROME1, MONOCHROME2, RGB, and YBR. There are limits on the performance of the Pegasus libraries.

For JPEG_BASELINE, JPEG_EXTENDED_2_4, and JPEG_LOSSLESS_HIER_14, images can be compressed or decompressed at a maximum rate of 3 images (or frames) per second. For JPEG_2000 and JPEG_2000_LOSSLESS_ONLY, a dialog will be displayed on Windows each time the compressor or decompressor is used. For other platforms a message will be displayed to stdout and a several second delay will occur. Full licenses can be purchased from Pegasus and configured in MergeCOM-3 to remove these compression and decompression limits. The licenses can be configured in the mergecom.pro configuration file.

The JPEG_BASELINE transfer syntax is UID 1.2.840.10008.1.2.4.50, JPEG Baseline (Process 1): Default Transfer Syntax for Lossy JPEG 8 Bit Image Compression, and uses Pegasus libraries 6420/6520. Table 19 details the photometric interpretation and bit depths supported by the standard compressor and decompressor for this transfer syntax. When lossy compressing RGB data, the standard compressor by default compresses the data into YBR_FULL_422 format. The compressor can also compress in YBR_FULL format if the COMPRESSION_RGB_TRANSFORM_FORMAT configuration option is set to YBR_FULL. The Photometric Interpretation tag must be changed by the application after compressing RGB data. Similarly, the Photometric Interpretation tag should be changed back to RGB when decompressing YBR_FULL or YBR_FULL_422 data.

JPEG Baseline			
Photometric Interpretation	MONOCHROME1 MONOCHROME2	RGB	YBR_FULL_422 YBR_FULL
Bits Stored	8	8	8
Bits Allocated	8	8	8
Samples Per Pixel	1	3	2

Table 19: JPEG Baseline Supported Photometric Interpretations and Bit Depths

The JPEG_EXTENDED_2_4 transfer syntax is UID 1.2.840.10008.1.2.4.51, JPEG Extended (Process 2 & 4): Default Transfer Syntax for Lossy JPEG 12 Bit Image Compression (Process 4 only), and uses Pegasus libraries 6420/6520. Table 20 details the photometric interpretation and bit depths supported by the standard compressor and decompressor for this transfer syntax. When lossy compressing RGB data, the standard compressor by default compresses the data into YBR_FULL_422 format. The compressor can also compress in YBR_FULL format if the COMPRESSION_RGB_TRANSFORM_FORMAT configuration option is set to YBR_FULL. The Photometric Interpretation tag must be changed by the application after compressing RGB data. Similarly, the Photometric Interpretation tag should be changed back to RGB when decompressing YBR_FULL or YBR_FULL_422 data.

JPEG Extended (Process 2 & 4)					
Photometric Interpretation	MONOCHROME1 MONOCHROME2			RGB	YBR_FULL_422 YBR_FULL
Bits Stored	8	10	12	8	8
Bits Allocated	8	16	16	8	8
Samples Per Pixel	1	1	1	3	2

Table 20: JPEG Extended Supported Photometric Interpretations and Bit Depths

The JPEG_LOSSLESS_HIER_14 transfer syntax is UID 1.2.840.10008.1.2.4.70, JPEG Lossless, Non-Hierarchical, First-Order Prediction (Process 14 [Selection Value 1]): Default Transfer Syntax for Lossless JPEG Image Compression, and uses Pegasus libraries 6220/6320. Table 21 details the photometric interpretation and bit depths supported by the standard compressor and decompressor for this transfer syntax. The standard compressor does not do a color transformation to RGB data when compressing with JPEG_LOSSLESS_HIER_14. The Photometric Interpretation tag should be left as RGB in this case.

JPEG Lossless Non-Hierchical Process 14					
Photometric Interpretation	MONOCHROME1 MONOCHROME2				RGB
Bits Stored	8	10	12	16	8
Bits Allocated	8	16	16	16	8
Samples Per Pixel	1	1	1	1	3

Table 21: JPEG Lossless Supported Photometric Interpretations and Bit Depths

The JPEG_2000 transfer syntax is UID 1.2.840.10008.1.2.4.91, JPEG 2000 Image Compression, and uses Pegasus libraries 6820/6920 for lossy or lossless. Table 22 details the photometric interpretation and bit depths supported by the standard compressor and decompressor for this transfer syntax.

JPEG 2000 (When used for Lossy)						
Photometric Interpretation	MONOCHROME1 MONOCHROME2				YBR_ICT	RGB
Bits Stored	8	10	12	16	8	8
Bits Allocated	8	16	16	16	8	8

Samples per Pixel	1	1	1	1	3	3

Table 22: JPEG 2000 Lossy Supported Photometric Interpretations and Bit Depths

The JPEG_2000_LOSSLESS_ONLY transfer syntax is UID 1.2.840.10008.1.2.4.90, JPEG 2000 Image Compression (Lossless Only), and uses Pegasus libraries 6820/6920 for lossless. Table 23 details the photometric interpretation and bit depths supported by the standard compressor and decompressor for this transfer syntax.

JPEG 2000 Lossless						
Photometric Interpretation	MONOCHROME1 MONOCHROME2				YBR_RCT	RGB
Bits Stored	8	10	12	16	8	8
Bits Allocated	8	16	16	16	8	8
Samples Per Pixel	1	1	1	1	3	3

Table 23: JPEG 2000 Lossless Supported Photometric Interpretations and Bit Depths

SPECIAL NOTES!

When using the standard compressor, all data needs to be right justified, i.e. bit 0 contains data, but the highest bits may not. RGB and YBR must be non-planar (R1G1B1, R2G2B2, ... or Y1Y2B1R1, Y3Y4B3R3,...)

JPEG_2000/JPEG_2000_LOSSLESS_ONLY will cause a irreversible, or reversible color transformation when compressing RGB data. The Photometric Interpretation MUST be changed from RGB to:

- YBR_ICT if JPEG_2000 is used with COMPRESSION_WHEN_J2K_USE_LOSSY = Yes (Lossy color transform for lossy compression)
- YBR_RCT if JPEG_2000_LOSSLESS_ONLY, or JPEG_2000 with COMPRESSION_WHEN_J2K_USE_LOSSY = No (Lossless color transform for lossless compression).

Similarly, on the decompression end, the Photometric Interpretation should be changed back to RGB, but the Lossy Image Compression attribute should indicate it has been lossy compressed.

Using Callback Functions

The Callback Functions (with a capital 'C' - capital 'F') discussed in this section, exhibit one significant difference from the callback functions used in the MC_Get_Value_To / From_Function( ) functions and stream handling functions described earlier. Callback Functions 'throttle' the data flow as the message object is communicated over the network. Rather than storing attributes with large OB/OW values within the message object itself, your application is responsible for maintaining the value of these attributes. This can be useful on systems with limited resources or when dealing with very large pixel data elements, such as large multi-frame images.

MC_Register_Callback_Function( ) is used when performing DICOM interchangeable media operations; when the MC_Open_File_Bypass_OBOW( ) function is called to read in a DICOM file. See further discussion of this function in the DICOM Files section later in this manual.

server Callbacks

A server (SCP) application can call MC_Register_Callback_Function( ) to register a Callback Function that will be called repetitively as the attribute's value arrives on an association during an MC_Read_Message( ) call. By the time the MC_Read_Message( ) returns to the application, the attribute value will already have been handled by your registered Callback Function. The Callback Function could be used by the server to treat this large block of OB/OW data (usually pixel data) specially (e.g., store in a frame buffer, filter through decompression hardware, write to disk...) without any overhead introduced by the message object.

client Callbacks

A client (SCU) application can call MC_Register_Callback_Function( ) to register a Callback Function that will be called repetitively as the attribute's value is transmitted over an association during an MC_Send_Request_Message( ) or MC_Send_Response_Message( ) call. During either of these calls, the attribute value will be handled by your registered Callback Function before these calls can return to your application. The Callback Function can also be used by the client to specially manage OB/OW data (e.g., read from a frame buffer, filter through compression hardware or software, read from disk...) without any overhead introduced by the message object.

other uses of Callbacks

Callback Functions can also be used in other situations through the use of MC_Set_Message_Callbacks( ). Normally Callback Functions are associated with a message or file when a MergeCOM-3 function contains an application ID as a parameter (or when an association ID is associated with an application ID that is used as a parameter). The MC_Set_Message_Callbacks( ) function allows the user to directly associate Callback Functions for an application with a message or file object. After calling MC_Set_Message_Callbacks( ), subsequent calls to functions such as MC_Open_File( ), MC_Message_To_Stream( ), or MC_Stream_To_Message( ) can use Callback Functions for managing pixel data.

Use of empty messages require adding the pixel data tag!

Note that when creating a message with MC_Open_Empty_Message( ) or MC_Create_Empty_File( ) it is necessary to add the pixel data tag into the message for Callback Functions to work properly. This can be done through the use of MC_Add_Standard_Attribute( ). Without a placeholder tag in the message or file to signal MergeCOM-3 that pixel should be included in the message or file, MergeCOM-3 will not know that Callback Functions should be used with the message or file. When creating messages with MC_Open_Message( ) or MC_Create_File( ), this is not necessary.

how to register a Callback Function

The MC_Register_Callback_Function( ) can be used to register a Callback Function with the tool kit as follows:

```
Status = MC_Register_Callback_Function (myApplicationID,
       MC_ATT_PIXEL_DATA, myUserInfo, MyCallbackFunction);
```

This call registers MyCallbackFunction( ) with the Advanced Tool Kit library to handle the pixel data (7FE0,0010) attribute for myApplicationID. myUserInfo can be set to NULL or point to a user defined structure that can be used to communicate application specific data to the Callback Function. A single Callback Function can be multiply registered to handle many tags. Also, a single Callback Function will handle both transmittal and reception of the data associated with the Tag(s).

MyCallbackFunction( ) is prototyped as follows:

the Callback Function

```
MC_STATUS MyCallBackFunction (int CBMessageID,
      unsigned long CBtag, void* myUserInfo,
      CALLBACK_TYPE CBtype, unsigned long* CBdataSizePtr,
      void** CBdataBufferPtr,  int CBisFirstPtr,
      int* CBisLastPtr);
```

CBMessageID and CBtag allow a single Callback Function to handle requests for different message and tag combinations. myUserInfo is a pointer to the user defined structure passed in from the MC_Register_Callback_Function( ) described above.

Based on the value of CBtype, your Callback Function determines how it is to behave; whether it is to supply/receive the length of the entire attributes value or supply/receive a 'chunk' of value data. CBdataSizePtr, CBdataBufferPtr, CBisFirstPtr, and CBisLastPtr are used to manage the data flow. Table 24 describes this behavior.

Value of Cbtype	Required Callback Behavior
REQUEST_FOR_DATA_LENGTH	Callback is supplying OB/OW/OF data and the length of the entire attributes value is being requested. The length is passed back using *CBdataSizePtr. CBdataBufferPtr is not used.
REQUEST_FOR_DATA	Callback is supplying OB/OW data and a 'chunk' of this data is being requested. Callback must set *CBdataSizePtr to the number of bytes of data you are providing at *CbdataBufferPtr. The length of the 'chunk' must be less than INT_MAX. MergeCOM-3 will set CBisFirstPtr to TRUE (not zero) the first time it is requesting data for this attribute's value (i.e., when you are to provide the first block of data). Callback must set *CbisLastPtr to TRUE (not zero) the last time you are providing data for this attribute's value (i.e., when you are providing the last block of data).
PROVIDING_DATA_LENGTH	Callback is receiving OB/OW/OF data and the length of the entire value for this attribute is being supplied. The length is passed in using *CbdataSizePtr. CbdataBufferPtr is not used.
PROVIDING_MEDIA_DATA_LENGTH	Callback is receiving the offset of the value of a DICOM file attribute of Value Representation OB, OW or OF from the beginning of the file. CbdataBufferPtr points to a unsigned long that contains this offset. *CbdataSizePtr is set to the length of the value contained at the offset. This value for CBtype only occurs as a result of the MC_Open_File_Bypass_OBOW( ) function call. See later sections of this manual dealing with DICOM file access and the Reference Manual for more information.
PROVIDING_OFFSET_TABLE	Callback is receiving offset table of encapsulated OB/OW/OF data. MergeCOM-3 has set *CbdataSizePtr to the number of bytes of data it is providing at *CbdataBufferPtr. MergeCOM-3 will set CbisFirstPtr to TRUE (not zero) and *CbisLastPtr to TRUE (not zero). The entire offset table is passed in one call to the registered callback function. Note that the callback can be repeatedly called
PROVIDING_DATA	Callback is receiving OB/OW/OF data and a 'chunk' of this data is being supplied. MergeCOM-3 has set *CBdataSizePtr to the number of bytes of data it is providing at *CbdataBufferPtr. The length of the chunk must be less than INT_MAX. MergeCOM-3 will set CbisFirstPtr to TRUE (not zero) the first time it is providing data for this attribute's value (i.e., when it is providing the first block of data). MergeCOM-3 sets *CbisLastPtr to TRUE (not zero) the last time it will be providing data for for this attribute's value (i.e., when it is providing the last block of data).

Table 24: Callback Function Behavior Based on Value of CBtype parameter

Because a single Callback Function's behavior and usage of parameters must vary based on the value of `CBtype` this can all seem very confusing. You may want to break for a beverage and read this section once more. Also, be sure to read the detailed specification of `MC_Register_Callback_Function( )` in the Reference Manual.

Sequences of Items

The DICOM Value Representation SQ is used to indicate an attribute in a DICOM message that contains a value that is a sequence of items. A sequence of items is a set of object instances, where each object instance can also contain attributes that have a VR of SQ. This powerful capability allows the nesting of objects, or the definition of 'container' objects (such as folders, film boxes, directories, etc.). One can think of these nested objects as message objects minus the command portion.

Figure 16 shows a DICOM message containing a sequence of items running two levels deep. Note that these nested sequences are contained **within** the same Message Stream. Sequences of items can also be contained in a DICOM file, and we will see that they are contained in DICOMDIR's. An attribute whose value is a sequence of items is simply an attribute that has a potentially large and complex value. Fortunately, MergeCOM-3 Advanced allows your application to deal with sequences of items an item at a time and hierarchically, as pictured in Figure 15, and takes care of the encoding of the sequence within the DICOM message stream.

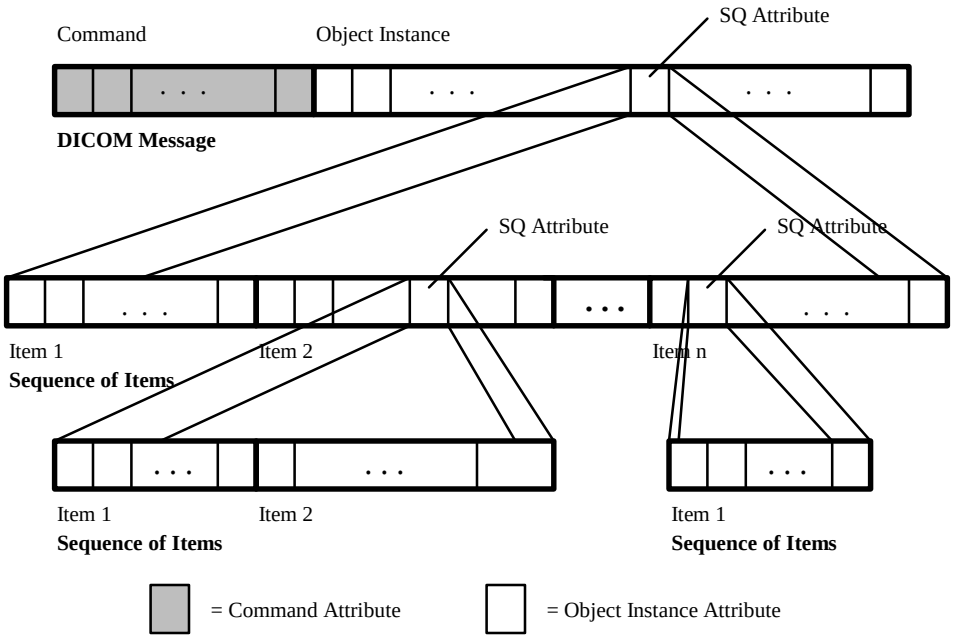

Figure 16: A DICOM Message containing doubly nested sequences of items.

encoding and decoding attributes in an item

Each item object in a sequence is a special class, or subclass in object-oriented terminology, of a message object. All the message building and parsing functionality described in previous sections of this manual also applies to item objects. The `MC_Get_Value_. . . ( )` and `MC_Set_Value_. . .( )` families of functions work on item objects as well as message objects; simply

specify an ItemID in the first parameter to these functions, rather than a MessageID.

Four additional MergeCOM-3 Advanced API functions are required for Items, and are specific to items. `MC_Open_Item( )` is used to create an Item with a given **`ItemName`** and returns an ItemID handle. `MC_Free_Item( )` is used to release the items resources back to the operating system. `MC_Empty_Item( )` is used to empty the values of all the attributes of an item and `MC_List_Item( )` lists the contents of an item object to an output stream.

`ItemName`'s are strings used to identify each item and are listed in the `message.txt` file for attributes in messages having a VR of SQ. The contents of each item are also listed in the `message.txt` file. Below are two excerpts of `message.txt`, one showing a reference to the Referenced Film Box Item, and the other the contents of that item.

```
################################################################
BASIC_FILM_SESSION - N_SET_RQ
################################################################

0008,0005    Specific Character set                        CS    3
Defined Terms:     ISO-IR 100, ISO-IR 101, ISO-IR 109, ISO-IR 110,
ISO-IR144, ISO-IR 127, ISO-IR 126, ISO-IR 138, ISO-IR 148
0008,0012    Instance Creation Date                        DA    3
0008,0013    Instance Creation Time                        TM    3
0008,0014    Instance Creator UID                          UI    3
0008,0016    SOP Class UID                                 UI    3
0008,0018    SOP Instance UID                              UI    3
2000,0010    Number of Copies                              IS    3
2000,0020    Print Priority                                CS    3
Enumerated Values: HIGH, MED, LOW
2000,0030    Medium Type                                   CS    3
Defined Terms:     PAPER, CLEAR FILM, BLUE FILM
2000,0040    Film Destination                              CS    3
Enumerated Values: MAGAZINE, PROCESSOR
2000,0050    Film Session Label                            LO    3
2000,0060    Memory Allocation                             IS    3
2000,0500    Referenced Film Box Sequence                  SQ    3
Item Name(s): REF_FILM_BOX

                                .
                                .
                                .

================================================================
Item Name: REF_FILM_BOX
================================================================

0008,1150    Referenced SOP Class UID                      UI    1
0008,1155    Referenced SOP Instance UID                   UI    1
```

encoding items in a sequence

To encode an item into an attribute of Value Representation SQ, treat the attribute as a multi-valued integer, where each value is an ItemID. This means using the `MC_Set_Value_From_Int( )` and `MC_Set_Next_Value_From_Int( )` functions where the Value parameter is the ItemID. Similarly, to decode an item, use the `MC_Get_Value_To_Int( )` and `MC_Get_Next_Value_To_Int( )` functions where the value returned is the ItemID.

The following sample code fragment gives an example of encoding a Pre-formatted Grayscale Image Item into a sequence:

```
status = MC_Open_Item(&itemID,
        "PREFORMATTED_GRAYSCALE_IMAGE");
if (status != MC_NORMAL_COMPLETION)
{
    printf("Unable to open request message:\n");
    printf("\t%s\n", MC_Error_Message(status));
```

```
        return 1;
    }

    status = MC_Set_Value_From_String(itemID,
         MC_ATT_PIXEL_ASPECT_RATIO, "1");
    if (status != MC_NORMAL_COMPLETION)
    {
        printf("MC_Set_Value_From_String failed:\n");
        printf("\t%s\n", MC_Error_Message(status));
        MC_Free_Item(&itemID);
        return 1;
    }

    status = MC_Set_Next_Value_From_String(itemID,
           MC_ATT_PIXEL_ASPECT_RATIO, "1");
    if (status != MC_NORMAL_COMPLETION)
    {
        printf("MC_Set_Value_From_String failed:\n");
        printf("\t%s\n", MC_Error_Message(status));
        MC_Free_Item(&itemID);
        return 1;
    }

    callbackInfo.messageID = itemID;

                .
                .
                .

/* encode other item attributes here */

                .
                .
                .

/* now encode the item into the sequence */

    status = MC_Set_Value_From_Int(messageID,
           MC_ATT_PREFORMATTED_GRAYSCALE_IMAGE_SEQUENCE,
            itemID);
    if (status != MC_NORMAL_COMPLETION)
    {
        printf("MC_Set_Value_From_Int failed:\n");
        printf("\t%s\n", MC_Error_Message(status));
        MC_Free_Message(&messageID);
        MC_Free_Item(&itemID);
        return 1;
    }
```

This excerpt is taken from the sample print application supplied with MergeCOM-3 Advanced. See that application's code for further example of encoding and decoding of sequences of items.

DICOM Files

Maintaining a DICOM file set is a matter of maintaining various DICOM files and a single DICOM directory file (DICOMDIR). First, the functions supplied by MergeCOM-3 Advanced that operate on all DICOM files are described; followed by a description of those functions that are especially suited for the complexities of the DICOMDIR file.

Figure 17 summarizes the DICOM file access function calls described in this section.

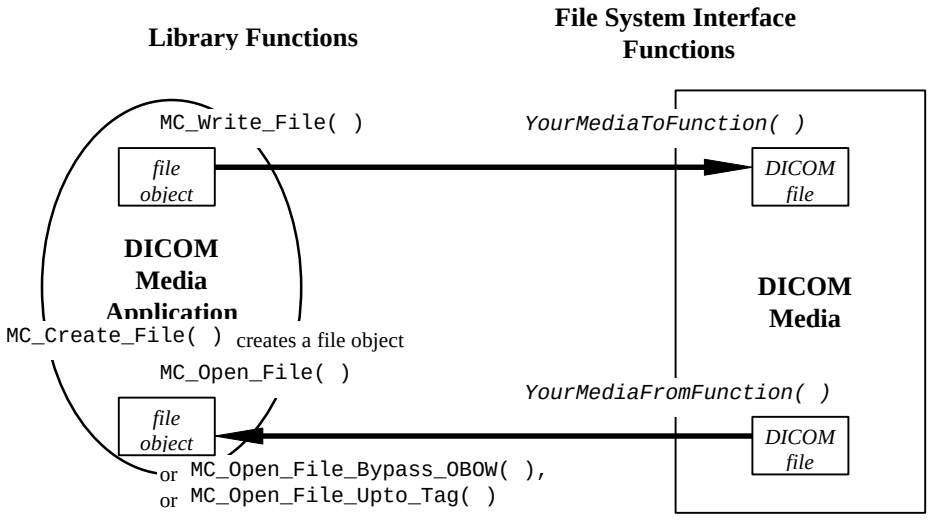

Figure 17: The DICOM File Access Functions in MergeCOM-3 Advanced

File System Interface Functions

This may sound strange, but all the media interchange functionality of the Advanced Toolkit relies on functions that you supply to interface with the particular physical medium and file system format on your target device. This approach was chosen because of the wide variety of media and file system configurations allowed by the DICOM Standard and the potentially unlimited combination of media devices, device drivers, and file system combinations for which DICOM media interchange applications may be developed.

you must interface with the selected media device

Your application need only supply two file system interface functions that the Advanced Library calls back; a *YourFromMediaFunction( )* that reads from your media of choice, and a *YourToMediaFunction( )* that performs the write operations to the same. These functions will be described in greater detail in the following sections, and they are often very simple to write (see the Sample Media Application supplied with the toolkit).

You will find that the Advanced Tool Kit provides powerful DICOM media functionality by supplying your application with:

- a greatly simplified way to deal with the complex encoding and decoding required within a DICOM file.
- very powerful DICOMDIR file navigation and maintenance functionality.
- an API that is very consistent with that used for the maintenance of DICOM messages used in network functionality; many of the encoding and decoding functions already described apply equally well to DICOM file objects.

To perform all this functionality on your medium of choice, you need only supply the two file system interface functions just discussed.

Creating a File Object

Before the contents of an existing DICOM file can be read in, or a new DICOM file can be created, a file object must be created. The `MC_Create_File( )` call creates the file object whose type is specified by the supplied service-command pair. For example, the following call creates an DICOM MR image file object.

```
/*
 * Create new MR Image file object
 */

status = MC_Create_File(&fileID, fileName, "STANDARD_MR",
C_STORE_RQ);

if (status != MC_NORMAL_COMPLETION)
{
   PrintError("Unable to create file object",status);
   exit ( EXIT_FAILURE );
}
```

The fileID variable is used to return a handle to the new file object, fileName is a string variable containing the DICOM file ID (file name); e.g., "PNR1/HDR3/AMR62", that will be given to the new object. "STANDARD_MR" is the service name, and C_STORE_RQ is the command name identifying the class of file object being created (see Table 5).

It is important to realize that `MC_Create_File( )` does not create the physical DICOM file out on the media; the `MC_Write_File( )` call described later does that. `MC_Create_File( )` corresponds to the `MC_Open_Message( )` call used in networking; it creates references to the proper message info file along with the data dictionary and builds an unpopulated file object instance for your application to fill in. This file object contains empty attributes.

Performance Tuning

Just as there is an `MC_Open_Empty_Message( )` available for networking, there is also a `MC_Create_Empty_File( )` available for media interchange. In this case, the message info and data dictionary files are not referenced and an empty message object instance is opened. This message object contains no attributes and the `MC_Set_Service_Command( )` function must be called to set the service and command for this file before it can be written to the file set. As in the case of networking, this approach is more efficient but penalizes you in the area of run-time error checking.

When your application is done using a file object, the file object should be freed using the `MC_Free_File( )` call.

Reading Files

To read in the contents of a DICOM file for analysis or parsing, you must open the file. Opening a DICOM file in the Advanced Tool Kit API means that a complete file object is filled in from an existing physical file. This means the entire DICOM file is read in on the open.

The following code reads a file into the file object:

```
/*
 * Read in the DICOM file
 */

status = MC_Open_File(myApplicationID, fileID, NULL,
MediaToFile);

if (status != MC_NORMAL_COMPLETION)
{
```

```
        PrintError("Unable to read file from media",status);
        exit ( EXIT_FAILURE );
    }
```

myApplicationID is the handle to your application entity obtained when you registered your application with the Tool Kit Library, fileID is the handle of a previously created file object, and NULL specifies that you aren't passing any user information to MediaToFile. MediaToFile( ) is the file system interface callback function you must author to read from your media device; described earlier as the *YourFromMediaFunction( )*.

The parameters for this callback function must agree with the following prototype:

```
MC_STATUS MediaToFile( char*          filename,
                       void*          userInfo,
                       int*           dataSize,
                       void**         dataBuffer,
                       int            isFirst,
                       int*           isLast);
```

fileName identifies the file you should be reading (e.g., "/PNR1/HDR3/AMR62"), dataSize specifies the even number of bytes of data you are providing , and dataBuffer is address of the data. isFirst indicates whether this is the first chunk of data read (e.g., you should open the file), isLast is set to a non-zero value by your application when it has finished returning data (e.g., has closed the file). You can decide for yourself how you wish to read in the data from the physical file, all at once or a block at a time. The library will callback MediaToFile( ) until you indicate that all data has been read. The userInfo parameter can be used to pass application specific data between the sample application and the callback function.

See the reference manual for a more detailed description of MC_Open_File( ) and the *YourFromMediaFunction( )* callback function.

Once the file object has been opened, the same MC_Get_Value . . . ( ) family of parsing functions used for message objects (described earlier) can be used to read attribute values from the file object. Also, the same MC_Set_Value . . . ( ) family of functions can be used to modify the values of file attributes.

Performance Tuning

Two special variants on MC_Open_File( ) are also provided by the toolkit:

- MC_Open_File_Bypass_OBOW( ) will not read in an attribute's value that is of type OB or OW if and only if you have registered a Callback Function with MC_Register_Callback_Function( ) for that attribute. Instead, the offset of the attribute's value from the beginning of the file along with the length of the value will be passed to the user's registered Callback Function. The user's Callback Function can then deal with the Pixel Data as it sees fit; e.g., ignore it, read it in later, stream in it to rapid decompression or display hardware directly from the file.

- MC_Open_File_Upto_Tag( ) will read attributes of the file into the file object only up to a specified tag. The offset in bytes from the beginning of the file to the beginning of the first attribute equal to or greater than the specified tag is returned. The user's application must then deal with reading in the rest of the DICOM file from media. This function is most useful when the DICOM file contains pixel data (7FE0, 0010) as its last attribute and this pixel data is very large. In these instances, you may wish to ignore the pixel data, read it in later, or process it directly from the file using your own special filters or hardware.

See the Reference Manual for a detailed description of the use of these more advanced functions.

Creating and Writing Files

Once a DICOM file object has been created using the `MC_Create_File( )` function and filled in by using either the `MC_Open_File( )` or the `MC_Set_Value . . . ( )` functions (or a combination of both), the file can be written out to media using a single `MC_Write_File( )` function:

```
status = MC_Write_File(fileID, 2, NULL, FileToMedia);

if (status != MC_NORMAL_COMPLETION)
{
   printf("%s\tError on MC_Write_File:\n", prefix);
   printf("%s\t\t%s\n", prefix, MC_Error_Message(status));
   return status;
}

MC_Free_File(&fileID);   /* Free the file object once
                                written to media */
```

`fileID` is the handle to the file object being written. `2` is the value given to the `NumBytes` parameter and means that the file will be padded, if necessary, to have a total length that is a multiple of 2 using the file padding attribute (FFFC, FFFC). If this value is set to `0`, no file padding will occur. `NULL` specifies that you aren't passing any user information to `FileToMedia`. `FileToMedia( )` is the file system interface callback function you must author to write from your media device; described earlier as the *`YourToMediaFunction( )`*.

The parameters for this callback function must agree with the following prototype:

```
static MC_STATUS FileToMedia( char*       filename,
                              void*       userInfo,
                              int         dataSize,
                              void*       dataBuffer,
                              int         isFirst,
                              int         isLast);
```

`fileName` identifies the file you should be writing (e.g., "/PNR1/HDR3/AMR62"), `dataSize` specifies the even number of bytes of data you should write, and `dataBuffer` is address of the data. `isFirst` indicates whether this is the first chunk of data being written (e.g., you should open the file), `isLast` is set to `MTI_TRUE` by the library when it is requesting that you write the last block of data (e.g., you can now close the file). The library will callback `FileToMedia( )` until all data has been written. The `userInfo` parameter can be used to pass application specific data between the sample application and the callback function.

See the reference manual for a more detailed description of `MC_Write_File( )` and the *`YourToMediaFunction( )`* callback function.

Performance Tuning

One special variant on `MC_Write_File( )` is also provided by the toolkit:

- `MC_Write_File_By_Callback( )` functions similar to `MC_Write_File()` except that it allows for attributes whose values are of type OB or OW to be supplied by a Callback Function if and only if you have registered it with `MC_Register_Callback_Function( )`. The user's Callback Function can then deal with the Pixel Data as it sees fit.

Other Useful File Object Functions

Other useful functions that operate on file objects include:

- `MC_Reset_Filename( )` allows your application to change the file name associated with an existing file object. This could be useful if you have read in a DICOM file, modified its contents, and then wish to write the file out with a new file name.
- `MC_Empty_File( )` clears the value from all attributes in the file object. This function provides a more efficient mechanism for writing out several file objects of the same type. Your application can simply empty and refill the attribute values rather than inducing the processing overhead of freeing and creating a whole new file object.
- `MC_Set_File_Preamble( )` and `MC_Get_File_Preamble( )` allow specialized application to examine and modify the 128 byte DICOM file preamble. The Advanced Toolkit Library defaults the preamble to all zeroes (as directed by the standard) so in most cases your application will not need to use these functions.
- `MC_List_File( )` is the analogue of `MC_List_Message( )` except that it lists the contents of a file object (including the file preamble) rather than a message object. This is a very useful function for validation and diagnosis of your application. See the description of `MC_List_Message( )` earlier in this document.

See the Reference Manual and sample media application for more information on these calls.

File Validation

File validation occurs in much the same manner as message validation. Before the file can be validated it must be read into a file object created with an `MC_Create_File( )` call. If the file object was created using `MC_Create_Empty_File( )`, then `MC_Set_Service_Command( )` must be called to identify the type of file object before validation can occur. Please see the earlier discussion of message validation, as almost all of it applies equally well to file validation.

Performance Tuning

DICOM file validation does involve processing overhead. The most significant overhead is in the accessing of the message info files, and significantly less overhead is involved in actually validating the contents of the file object structure. It is **important** to understand that depending on the way in which your message object was created, this validation overhead can occur at different points in your application; see Table 25.

File Object Creation Method	Point at which file access overhead for validation occurs
`MC_Create_File( )`	`MC_Create_File( )`
`MC_Create_Empty_File( )`	`MC_Validate_File( )` **Note:** You must use `MC_Set_Service_Command( )` before validating and/or writing a file created in this manner.

Table 25: Point of performance overhead associated with file validation.

Using MC_Create_File( ) has an up-front performance cost but provides additional validation as you set the value of attributes in the file object. With the MC_Create_Empty_File( ) method, the cost occurs on validation itself.

Many times the MC_Validate_File( ) is selectively used in an application: as a runtime option or conditionally compiled into the source code. Validation might only be used during integration testing or in the field for diagnostic purposes. Reasons for this include performance since the overhead associated with file validation may be an issue, especially for larger files having many attributes or on lower-end platforms. Also, validation can clutter the message log with warnings and errors that may not be desirable in a production environment.

Converting Files to/from Messages

Two very useful and powerful functions are supplied for converting file objects to message objects, and vice versa. These functions are MC_File_To_Message( ) and MC_Message_To_File( ). Remember that most DICOM files (other than the DICOMDIR file) are simply the information object portion of a DICOM message encapsulated within a DICOM file (surrounded by a file preamble, meta information, and optional padding).

MC_File_To_Message( ) has a single FileID parameter and returns a status. This call converts the FileID to a message handle by removing the file preamble, meta information, and optional padding from the file object and adding the command attributes. While MergeCOM-3 Advanced sets the values of many of these command type attributes automatically, some services require the application to set them. Once converted to a message object, the object can be sent and received over the network using the calls detailed earlier.

Conversely, MC_Message_To_File( ) has two parameters, MessageID and FileName, and also returns a status. This call converts the MessageID to a file handle by removing the command attributes from the message object and adding the file preamble, and meta information attributes. The file meta information attributes must be set by the user. Optional padding can be added if required using the MC_Write_File( ) call. If the message was originally opened as an empty message and the command and service were not set, then MC_Set_Service_Command( ) must be called before the file object can be validated.

These two functions are most often useful when reading and writing image files to and from DICOM media that were received (or will need to be transmitted) over the network as C-STORE request messages.

DICOMDIR

As discussed earlier, in each DICOM File Set (containing many DICOM files) there must exist a single DICOM File with the reserved File ID "DICOMDIR". This file contains identifying information for the file set that most often includes a directory of the file sets contents. A media interchange application would make use of and maintain the DICOMDIR to locate a particular file within the file set for processing.

Structure

An information object portion of a DICOMDIR file has a special structure that is described in Part 3 (PS 3.3) of the DICOM Standard. We described this structure earlier in this document (see Figure 8) as a hierarchy of directory entities, where each directory entity contains a set of semantically related directory records. These

directory records can have a one-to-one relationship to a DICOM file within the file set described by the DICOMDIR, and can also reference another (lower-level) semantically related directory entity. Directory records do not have to reference a DICOM file, they can be used solely to contain information that helps an application navigate down the directory hierarchy to locate the desired DICOM file.

As an example, the Root directory entity might contain two Patient directory records and a Topic directory record. One of the Patient directory records references a directory entity containing multiple Series records and a Film Session record for that Patient. Each of these Series records reference directory entities containing Image records for that patient. It is these Image records that reference the DICOM file containing the image objects acquired for the Patient whose directory hierarchy we have traversed. (See Table 6 for a description of the allowed entity hierarchies).

This directory entity hierarchy is encoded within the DICOMDIR as a single, potentially very complex sequence of items, where each item is a directory record. Byte offset attributes within the directory records are used to point to other directory records within the same directory entity, as well as lower-level directory entities (if one exists) referenced by a directory record. DICOM File ID's are encoded in the directory record if the record references a particular DICOM file in the file set.

DICOMDIR's are ugly!

The key observation here is that rather than using nested Sequences of Items to encode the DICOMDIR hierarchy, the standard chose to use a single, potentially very large, sequence of items and byte offsets. The standard defines these byte offsets as being measured "from the first byte of the file meta-information". As you might well imagine, the complexity of maintaining these byte offsets accurately, as directory records are added to or removed from directory entities within the DICOMDIR file, is very great and can be very cumbersome.

but we pretty them up

Fortunately, the MergeCOM-3 Advanced toolkit supplies functions that make DICOMDIR maintenance much simpler for your application. These functions are now described.

Opening and Navigation

Opening a DICOMDIR file is performed just like the opening of any other DICOM file, except that the file name is "DICOMDIR" and the service and command names are DICOMDIR and C_STORE_RQ, respectively (see Table 5).

navigating through a DICOMDIR

Once open, navigating a DICOMDIR usually involves calling `MC_Dir_Root_Entity( )` to get to the root of the DICOMDIR. From here `MC_Dir_Next_Entity( )` is used to traverse directory entities; while `MC_Dir_First_Record( )` and `MC_Dir_Next_Record( )` is used to traverse records within a directory entity. The function prototype for the `MC_Dir_Next_Entity( )` follows:

```
MC_STATUS EXP_FUNC MC_Dir_Next_Entity(
       int             AdirID,
       int             AitemID,
       int*            AnextEntityID,
       int*            AnextItemID,
       char**          AnextItemType,
       int*            AisLast);
```

`AdirID` is the file handle for the DICOMDIR file and is passed in by your application. `AitemID` is the Item object ID or handle for the current item in the DICOMDIR, and is also passed in. The values returned from this function call

include AnextEntityID, which is a handle to the next directory entity; AnextItemID, which is a item handle to the first directory record contained in the next directory entity; AnextItemType, a character string containing the directory record type (e.g., "DIR_REC_PATIENT", "DIR_REC_STUDY", "DIR_REC_FILM_BOX"); and AisLast, which is set to a non-zero value if the directory record returned is the only (and therefore last) directory record in the next entity.

The other four navigation functions have similar parameters; please refer to the Reference Manual and to the Media Sample Application source code and manual for further details. Also, note in the Reference Manual that the these navigation functions will return and log errors if an illegal directory record hierarchy is detected. Finally, since no functions are supplied for backward traversal (due to the nature of the DICOMDIR structure), your application may want to 'cache' certain key directory record items so that it can quickly back up within the directory record hierarchy.

Adding and Deleting Records

The addition or deletion of directory records are handled through two simple calls; MC_Dir_Add_Record( ) and MC_Dir_Delete_Record( ). These calls are prototyped as follows:

```
MC_STATUS EXP_FUNC MC_Dir_Add_Record (
       int             AdirID,
       int             AentityID,
       char*           AnewItemType,
       int*            AnewItemID);

MC_STATUS EXP_FUNC MC_Dir_Delete_Record (
       int             AdirID,
       int             AitemID);
```

Before adding the directory record, you should use MC_Open_Item( ) and the MC_Set_Value_...( ) family of function calls to create and populate the contents of the directory record item.

When adding a record using MC_Dir_Add_Record( ), you pass in handles to the DICOMDIR file object (AdirID) and the current directory entity (AentityID) to which you wish to add the directory record. You also pass in the string identifying the directory record item type (AnewItemType). For example, if you wished to add a Study record, AnewItemType would be a pointer to a string containing "DIR_STUDY". AnewItemID is returned and contains a handle to the new item object.

automatic deletion of referenced items

When deleting a record using MC_Dir_Delete_Record( ), the only parameters required are a handle to the DICOMDIR (AdirID), and a handle to the directory record item (AitemID) to be deleted . When a directory record is deleted, all lower level directory entities (and the directory records contained within them) are also freed. If the directory record deleted is the last in a directory entity, the directory entity is also freed. Finally, if the deleted directory record references any MRDR's (see below) the reference count in the MRDR is adjusted.

Make sure you are committed!!

The MergeCOM-3 Advanced toolkit updates and maintains all the byte offsets that are part of the DICOMDIR structure automatically. <u>But, one important note: All the changes to a DICOMDIR are made in memory and are not committed to media until a MC_Write_File( ) call is made</u>.

MRDR's

In some cases, it is desirable for a single DICOM file within a file set to be referenced by multiple directory records within the file set's DICOMDIR. In this case, it is important on a delete operation for the application to know if the DICOM file can actually be deleted (e.g., are there any other directory records pointing to this file). A special type of directory record is defined by DICOM for this purpose; the **Multi-Referenced Directory Record (MRDR)**.

Whenever a DICOM File is to be referenced by more than one directory record, all references must be redirected through a MRDR. In these cases the directory records referencing the same DICOM file all reference an MRDR within the DICOMDIR and it is the MRDR that references the DICOM file within the file set. In other words, MRDR's provide indirect DICOM file referencing. See Figure 18.

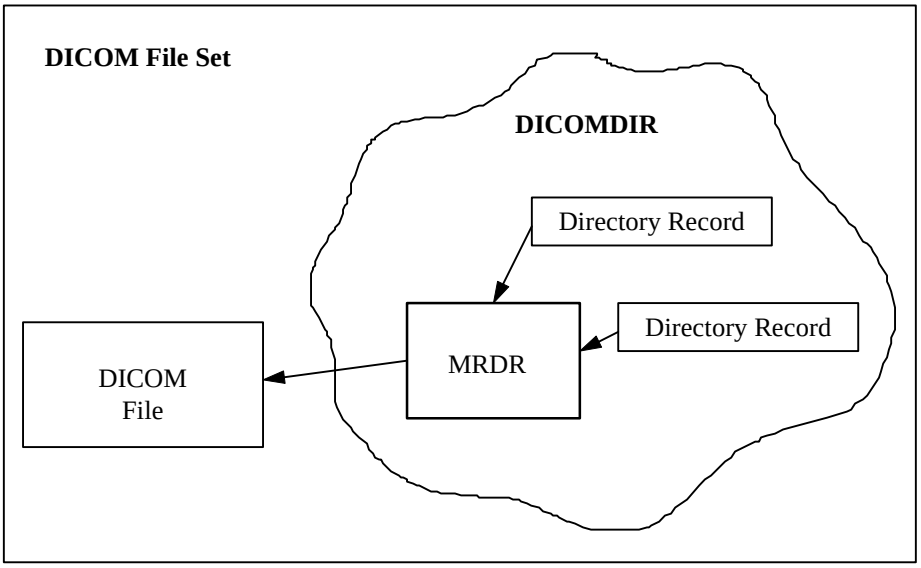

Figure 18: Referencing a DICOM File Indirectly through a MRDR

The usual procedure for making use of a MRDR is to use the `MC_Dir_Open_MRDR( )` function to first create a new MRDR that is referenced by an existing directory record item in the DICOMDIR. The function prototype follows:

```
MC_STATUS MC_Dir_Open_MRDR (
       int             AdirID,
       int             ArefItemID,
       char*           AnewMrdrType,
       int*            AnewMrdrID);
```

`AdirID` identifies the DICOMDIR object handle, `ArefItemID` is the ID or handle of the existing directory record item that will reference the new MRDR, `AnewMrdrType` is the directory record type that will be referenced by the MRDR (e.g., "`DIR_REC_IMAGE`"), and `AnewMrdrID` is the handle returned for the new MRDR record.

To add an additional reference to a MRDR, use the `MC_Dir_Reference_MRDR( )` function call. The parameters for this function are: `AdirID`, `ArefItemID`, and `AmrdrID`. The first two parameters are used as above, while `AmrdrID` is the handle of the existing MRDR that you wish to reference.

```
MC_STATUS MC_Dir_Reference_MRDR (
       int             AdirID,
       int             ArefItemID,
       int             AmrdrID);
```

Finally, the `MC_Dir_Remove_Ref_MRDR( )` function is used to remove a directory record's reference to a MRDR. The directory record is identified by `AdirID` and the MRDR is identified by `ArefItemID`.

```
MC_STATUS MC_Dir_Remove_Ref_MRDR (
       int             AdirID,
       int             ArefItemID);
```

reference counts are automatically maintained

All of these functions automatically maintain the reference count within the MRDR, and `MC_Dir_Remove_Ref_MRDR( )` automatically deletes the MRDR from the DICOMDIR object when the reference count reaches zero. Otherwise, your application is responsible for maintaining attribute values in the MRDR and directory records.

Adding and Deleting Entities

As stated earlier, one can think of a directory entity solely as a collection of semantically related directory records. For this reason, when your application creates a new directory entity, it must create it in the context of a directory record that is to be contained in that entity. Additional directory records can then be added to or removed from the new directory entity.

Use the `MC_Dir_Add_Entity( )` call to create a new directory entity. When making the call you must specify the DICOMDIR object (`AdirID`) to which you are adding the entity, the directory record item (`AitemID`) which will reference the new directory entity , and the directory record type (`AnewItemType`) for the first directory record of the new directory entity. The function returns the directory entity handle (`AnewEntityID`) for the new directory entity, along with the directory record handle (`AnewItemID`) for the new directory record contained in this entity.

```
extern MC_Dir_Add_Entity (
       int             AdirID,
       int             AitemID,
       char*           AnewItemType,
       int*            AnewEntityID,
       int*            AnewItemID);
```

To delete a directory entity, and thereby delete all the directory entities and directory records it in turn references, you should call `MC_Dir_Delete_Referenced_Entity( )`. This function only requires a DICOMDIR handle (`AdirID`) and a handle to the directory record item (`AitemID`) that references the directory entity to be deleted.

```
extern MC_Dir_Delete_Referenced_Entity (
       int             AdirID,
       int             AitemID);
```

Private Attributes

Private attributes supply a mechanism for applications to extend standard message objects and were discussed earlier in this document. Private attributes in message objects are handled in much the same way as standard attributes with three major exceptions:

1. standard attributes in the MergeCOM-3 API are referenced by `Tag`, while private attributes are referred to by `PrivateCode`, `Group`, and `ElementByte`.
2. The `Group` number of a private attribute must always be odd, while for a standard attribute it is always even.
3. Before encoding a private attribute into a message object, your application must allocate a private block for that attribute, and then add the attribute to that private block in that message object.

adding private attributes to a message

Private attributes are added to a message object using the `MC_Add_Private_Block( )` and `MC_Add_Private_Attribute( )` calls. `MC_Add_Private_Block( )` is used to reserve a block of up to 255 private attributes. Taking the example earlier in this document, to add a private block to group 1455 of `myMessage` with a `PrivateCode` of 'ACME_IMG_INC' you would make the call:

```
Status = MC_Add_Private_Block( myMessageID, "ACME_IMG_CORP",
             0x1455);
```

Once reserved, a private block is referenced using a `PrivateCode`. The following call could then be used to add an attribute that contains the name of a field engineer to the private block:

```
Status = MC_Add_Private_Attribute( myMessageID,
             "ACME_IMG_CORP", 0x1455, 00, PN);
```

Up to 255 other private attributes could be added to the ACME_IMG_CORP private block in group 1455 using the above call and `ElementByte` values of `01` through FF. If more attributes are required, another private block (with a different `PrivateCode`) will need to be added.

`PrivateCodes` must be used to refer to private attributes, because private blocks may be placed in different locations within a private group, depending on what other blocks of private attributes have already been reserved. `PrivateCodes` are a way to refer to these blocks, independently of their physical location in the message stream.

assigning values to private attributes

Once private attributes have been added, they can be assigned values identically to standard attributes, except that all the `MC_Set_Value_ . . . ( )` functions are replaced with `MC_Set_pValue_ . . . ( )` functions. The `MC_Set_pValue_ . . . ( )` functions require `PrivateCode`, `Group`, and `ElementByte`, rather than `Tag`, to identify an attribute. For example, to assign `"Adams^John Robert Quincy"` as the name of the field engineer, your application could call:

```
Status = MC_Set_pValue_From_String( myMessageID,
       "ACME_IMG_CORP", 0x1455, 00,
       "Adams^John Robert Quincy");
```

retrieving values from private attributes

Similarly, retrieving values from private attributes makes use of `MC_Get_pValue...( )` rather than `MC_Get_Value...( )` functions. The `MC_Get_pAttribute_Info( )` call can also be very useful in retrieving information about private attributes before your application begins to process them. See the Reference Manual for further details on the handling of private attributes with these and other calls.

Memory Management

Performance Tuning

MergeCOM-3 contains its own memory management routines that are optimized for how it uses memory. They have been adapted to manage specific data structures that are frequently allocated by MergeCOM-3. These include but are not limited to data structures for associations, messages, and tags. The memory management routines have the characteristic that they do not actually "free" the memory that has been acquired. Instead, they mark the data as being free and place the memory in a list for reuse later. These routines have been optimized to quickly acquire and free memory being used by MergeCOM-3. They also allow MergeCOM-3 to not depend on the memory management of a particular operating system.

These memory routines have also been extended for use with variable sized memory buffers. MergeCOM-3 uses these routines to allocate buffers in sizes between 4 bytes and 28K. When an allocation is requested, MergeCOM-3 will take the smallest buffer that will fit the bytes requested. These buffers will be kept in MergeCOM-3's internal memory pool and never freed. For allocations larger than 28K, MergeCOM-3 will simply use the 'C' functions `malloc()` and `free()`. Under most conditions, MergeCOM-3 breaks up large DICOM data elements such as pixel data into chunks of data smaller than 28K so that they can be managed through these routines.

The end result of these routines is that applications using MergeCOM-3 expand to the maximum amount of memory used at one time. The total memory allocation will not shrink from this point. In applications that repeatedly perform a consistent operation, the memory being used by MergeCOM-3 should stabilize and not increase in size. In applications using MergeCOM-3 from multiple threads, this memory usage is not as consistent and depends on the timing of the threads using MergeCOM-3. As a result of these routines, the first time an application performs a DICOM operation is typically slower than subsequent operations.

When developing a DICOM application with MergeCOM-3, the most memory intensive operation is dealing with image data. The following sections discuss various MergeCOM-3 functions. A description is given of how these functions manage memory in conjunction with various MergeCOM-3 configuration settings.

Assigning Pixel Data

MC_Set_Value_From_Function

`MC_Set_Value_From_Function` is used to assign OB, OW, or OF data to a DICOM tag. These value representations are used to store image data or other large data elements. `MC_Set_Value_From_Function` is described in further detail elsewhere in this manual.

Data can be passed to MergeCOM-3 via `MC_Set_Value_From_Function` in several ways. The entire data can be passed in a single call, or the data can be supplied in several smaller chunks. When passed data, `MC_Set_Value_From_Function` will allocate a buffer the size of the chunk passed to it and copy the data into this buffer for storage.

The size of data passed to `MC_Set_Value_From_Function` will dictate how the image data is stored. If the data is passed in chunks smaller than 28K, MergeCOM-3's internal memory management code will be used. If the chunks are larger than 28K, `malloc()` will be used to allocate storage for the buffers. If large images are being dealt with, it may be desirable to pass this data in chunks larger than 28K, so the memory is freed after processing has been completed for the image. This will keep the nominal memory usage of MergeCOM-3 lower. When passing data in chunks less than 28K, it is recommended that sizes of 16K, 20K,

24K, or 28K be used. Using these size chunks will reduce the overhead in storing the data.

MC_Set_Value_From_Function can also be directed to store data in temporary files. The LARGE_DATA_STORE and LARGE_DATA_SIZE configuration options in the mergecom.pro file dictate when data is stored in temporary files. When the LARGE_DATA_STORE option is set to FILE, data elements that are larger than configured by the LARGE_DATA_SIZE option are stored in temporary files. The size of buffer passed to MC_Set_Value_From_Function does not have an effect on memory usage.

Reading Messages from the Network

MC_Read_Message

MergeCOM-3 has a single function for reading messages from the network. MC_Read_Message creates a message object and loads the message into memory while reading from the network. When using MergeCOM-3's standard memory management routines, the method for storing the image data can be influenced.

Data is read from the network by PDUs. However, it is stored internally in sizes dictated by the WORK_BUFFER_SIZE configuration value. If a chunk of data read is smaller than the value for the WORK_BUFFER_SIZE, the chunk will simply be stored. If it is larger, the data will be stored internally in WORK_BUFFER_SIZE buffers.

By supporting a maximum PDU size and WORK_BUFFER_SIZE larger than 28K, MergeCOM-3 will store the buffers in memory allocated with the 'C' function malloc(). This can be used to reduce MergeCOM-3's typical memory usage. Note, however, that SCU systems do not necessarily size their PDUs according to the Maximum PDU size negotiated. This solution does not guarantee that image data will be stored with malloc().

As is the case when assigning image data with MC_Set_Value_From_Function, the LARGE_DATA_STORE and LARGE_DATA_SIZE configuration options can be used to store the data in temporary files.

Loading Messages from Disk.

MC_Open_File, MC_Stream_To_Message

This functionality shares the same characteristics as when data is being read from the network with MC_Read_Message. MC_Stream_To_Message is used to read DICOM "stream" objects, and MC_Open_File, MC_Open_File_Bypass_OBOW, and MC_Open_File_Upto_Tag are used to read DICOM Part 10 format files. Whereas objects being read from the network determine memory usage by the PDU size, these functions determine memory usage by the size buffers passed from their callback functions. The WORK_BUFFER_SIZE configuration value has the same impact as when reading from the network.

If the data is stored in file format, the MC_Open_File_Bypass_OBOW or MC_Open_File_Upto_Tag functions can be used to leave the image data on disk until it is sent over the network.

Using Registered Callbacks

MC_Register_Callback_Function

MergeCOM-3 also supplies a method to allow the user to manage image data through the use of registered callback functions. MC_Register_Callback_Function associates a callback function with a DICOM attribute such as pixel data. These callbacks are limited to attributes with the value representations of OB, OW, or OL. When encountered, the attribute's data is passed to the registered callback function instead of being stored within MergeCOM-3. The callback is also used to supply the attribute's data. The size of

data elements for which to use callbacks can also be specified. The CALLBACK_MIN_DATA_SIZE configuration option can specify the minimum size or length required for use of a registered callback function.

There are three models in which MC_Register_Callback_Function can be used. First, it can be used to seamlessly replace MergeCOM-3's memory management functions. Use of this function can for the most part be hidden from the application. Secondly, the function can be used as an interface to receive or supply data only when it is needed. When writing a network application, the image data can be supplied to the user directly as it is read off the network. The data can also be supplied when it is about to be written to the network. This functionality can also be used when creating and reading DICOM files. Finally, MC_Register_Callback_Function can be used to save an image to disk as it is received over the network.

Replacing MergeCOM-3's Memory Management Functions For Pixel Data

When using MC_Register_Callback_Function to replace MergeCOM-3's memory management functions, the user would still use MC_Get_Value_To_Function and MC_Set_Value_From_Function to access the image. When requested, MergeCOM-3 will receive or supply the attribute's value to the registered callback. There are several additional requirements for this to function properly. MC_Set_Message_Callbacks must be called for the message or item before calling MC_Set_Value_From_Function. MC_Set_Message_Callbacks associates the callback function registered to a specific application to a message. Also, when a message or file object's memory is released, the registered callback function is not notified. There must be a link in the user's application between the registered callback and the code that is freeing the object.

Accessing Data When Needed

When dealing with large multi-frame images, it is sometimes impractical to load the entire image into memory at once. MC_Register_Callback_Function can be used to access image data only when needed. The memory requirements of an application can be greatly reduced by using this functionality.

When reading messages from the network, MC_Read_Message supplies the user's registered callback function with the image data. If the data does not need to be byte swapped into the system's native endian, the amount of data supplied with each call is dictated by the PDU size of the data received. When the data is byte swapped, the length of data is specified by the WORK_BUFFER_SIZE configuration value. As the data is received, it would typically be written to disk in this scenario. When MC_Read_Message returns, the user is given the message read from the network. The message object still contains a link to the registered callback function. This link can be removed by calling MC_Set_Value_To_Empty. The header data can then be examined and later written to disk.

When sending data over the network, MC_Send_Request_Message will call the user's registered callback function for the image data. The data can be supplied to MergeCOM-3 in any length as required by the user's application. The data is typically read from disk at this point and directly passed to MergeCOM-3. After MC_Send_Request_Message receives the data, it byte swaps the data if needed, and then writes it to the network.

This functionality is conducive to storing a message's header data separately from its image data. Depending on system requirements, this may be an aid in quickly

loading image data while bypassing MergeCOM-3. The complete image file can be reassembled later using MergeCOM-3.

Saving Received Images Directly to Disk

In conjunction with the registered callback function, data can also be stored directly to disk when it is being read. The image's header data can be written to disk from within the registered callback. The user must write the attribute tag, value representation if needed, and the length of the image data attribute to the file. The image data is written to the file in subsequent calls to the user's registered callback function.

When `MC_Read_Message` is parsing a message being received, it will notify the user's registered callback function when it has parsed the header information and determines the image data's length. The registered callback function will be called with the `PROVIDING_DATA_LENGTH` flag and is supplied the Message ID of the message being read. At this point, the user can stream the header file to disk with `MC_Message_To_Stream`. As the image data is received, it can be added to the end of this file.

Data can also be stored as DICOM files with this method. The message cannot be converted into a file object at this point with `MC_Message_To_File` as would normally done. So, a separate file must be created to add the DICOM Part 10 Meta Header information. This header can be written out from within the callback. After the end of the meta header, the message can be streamed to disk with a call to `MC_Message_To_Stream` in the transfer syntax specified in the Meta Header. As subsequent image data is passed to the user's callback function, the data can be written to file. Because the endian of the transfer syntax being written may be different than the endian of the system being used, there may be a need for byte swapping of the pixel data in this implementation.

There is a potential risk with this implementation. Although the current definition of the DICOM image types does not include any data elements after the pixel data, future versions may add data elements there.

Deploying Applications

There are several issues to consider when deploying a MergeCOM-3 based application. These include deciding which MergeCOM-3 files are needed for your application, how to set important configuration options to reduce problems in the field, and how to deal with potential UN VR problems. The following sections describe these issues in further detail.

MergeCOM-3 Required Files

There are a limited number of files required by MergeCOM-3 applications. These files are described in Table 26. Note that the use of some of these files can be avoided by using the `genconf` and `gendict` utilities. These utilities generate a source file from the configuration files that can be compiled and linked into your application.

File	Description and Use
`Merge.ini`	MergeCOM-3 initialization file. This file contains logging information and pathnames for the other configuration files. Use of this file can be avoided by using the `genconf` utility to link the file into the

	MergeCOM-3 application.
Mergecom.pro	MergeCOM-3 system profile. This file contains general run-time configuration options. Use of this file can be avoided by using the `genconf` utility to link it into the MergeCOM-3 application.
Mergecom.app	MergeCOM-3 application profile. This file contains configuration information about the services supported by the MergeCOM-3 application and information about remote DICOM applications. Use of this file can be avoided by using the `genconf` utility to link it into the MergeCOM-3 application.
Mergecom.srv	MergeCOM-3 services file. This file contains information about the services supported by MergeCOM-3. Use of this file can be avoided by using the `genconf` utility to link it into the MergeCOM-3 application.
Mrgcom3.msg	MergeCOM-3 message information file. This file contains validation information for DICOM messages. This file is required if the `MC_Open_Message`, `MC_Create_File`, `MC_Validate_Message`, `MC_Validate_File`, or `MC_Validate_Attribute` functions are called from the MergeCOM-3 application.
Mrgcom3.dct	MergeCOM-3 data dictionary file. This file contains information about all of the DICOM attributes. Use of this file can be avoided by using the `gendict` utility to link it into the MergeCOM-3 application.

Table 26: Files needed when deploying an application.

Configuration Options

The majority of MergeCOM-3's configuration options can be used to solve interoperability problems in the field. There are some options, however, that can be set before deploying a MergeCOM-3 application to help reduce potential problems. These options are listed in Table 27 with descriptions of how they can be set.

Configuration Option	Description
ACCEPT_ANY_APPLICATION_TITLE	When set to NO, MergeCOM-3 requires that the Application Entity title sent in an association request match one of the registered application titles for the SCP. When there is no match, the association will be automatically rejected. Setting this option to YES will eliminate some association negotiation problems in the field for SCP applications.
ACCEPT_ANY_HOSTNAME	When set to NO, MergeCOM-3 will attempt to resolve the IP address of the SCU application into a hostname. If this resolution cannot be done, the association will automatically be rejected. Setting this option to YES will reduce configuration problems in the field for SCP applications.
EXPORT_UN_VR_TO_MEDIA	Setting this option to YES will cause UN VR attributes to not be exported when writing DICOM Part 10 format files with `MC_Write_File` or `MC_Write_File_By_Callback`. See the following sections for a further discussion of UN VR.
EXPORT_UN_VR_TO_NETWORK	Setting this option to YES will cause UN VR attributes to not be exported over the network with `MC_Send_Request_Message`. See the following

	sections for a further discussion of UN VR.
IMPLEMENTATION_CLASS_UID	The Implementation Class UID is used to identify in a unique manner a specific class of implementation. PS3.7 of DICOM states: "(The Implementation Class UID) is intended to provide respective (each network node knows the other's implementation identity) and non-ambiguous identification in the event of communication problems encountered between two nodes." PS3.7 of DICOM further defines how this UID should be defined: "different equipment of the same type or product line (but having different serial numbers) shall use the same Implementation Class UID if they share the same implementation environment (i.e., software)."
IMPLEMENTATION_VERSION	The Implementation Version is intended to distinguish between software versions of an implementation. It should be set to the version of the MergeCOM-3 application.

Table 27: Configuration options to consider when deploying an application.

Configuring Remote Nodes for SCP Applications

Configuring remote nodes at run-time

Typical MergeCOM-3 SCU applications use the `mergecom.app` configuration file to configure SCP applications that it communicates with. Using this configuration file requires that remote applications be configured before the library is initialized. It may be desirable to configure remote nodes at run-time. The following example illustrates how `MC_Open_Association` can be used to specify remote node information:

```
MC_STATUS   mcStatus;
int         applicationID;
int         associationID;
char        remoteAE[64+2];
char        remoteHostname[100];
char        serviceList[100];
int         remotePort;

strcpy(remoteAE, "MERGE_STORE_SCP");
strcpy(remoteHostname, "myhost.merge.com");
strcpy(serviceList, "Storage_Service_List");
remotePort = 104;

mcStatus = MC_Open_Association( applicationID,
                                &associationID,
                                remoteAE,
                                &remotePort,
                                remoteHostname,
                                serviceList );
```

Note that the service list used to negotiate with the remote node still must be pre-configured. It is assumed that the services supported by an SCU application are predetermined.

UN VR

UN VR interoperability problems

DICOM Supplement 14, Unknown Value Representation, became a part of the DICOM standard on June 3, 1997. This supplement adds a new value representation, UN, to the DICOM standard. It was developed to fix two related holes in the DICOM standard:

- When standard or private attributes were received in an implicit value representation (VR) transfer syntax, and the user does not have a knowledge of

the VR of the attributes, there is no way to represent the VR for these attributes in an explicit VR transfer syntax.

- Every time a new VR is added to the standard, there is no way to determine if the length field in explicit value representation transfer syntaxes should be encoded as 2 bytes or 4 bytes, so a general parser could not be properly written to handle future VRs.

The need for this supplement is mainly for use in "archive" systems. An "archive" will typically want to preserve the private attributes contained within a message for later use. There also may be a need to add support for new image objects with new VRs to an "archive" system without having to change the software.

Unfortunately, the method that Supplement 14 specifies for encoding UN value representation attributes is typically not compatible with older DICOM implementations. Versions previous to 2.2.2 of the MergeCOM-3 tool kit do not parse these attributes properly. The `MC_Read_Message()` function call will fail and the association will be aborted if a UN VR attribute is received. This has obviously caused a variety of interoperability problems in the field.

The typical DICOM scenario where UN VR can cause a DICOM communication failure is the following: a modality exports a series of images to a PACS or "archive" system via the DICOM storage service class. The images were encoded in the implicit VR little endian transfer syntax and contain multiple private attributes. Later, a DICOM workstation decided to retrieve the images from the "archive" or PACS system. The workstation does not yet support UN VR, however, the PACS or "archive" system does. The workstation uses the DICOM query/retrieve service class to retrieve the series of images. When the images are exported to the workstation with an explicit VR transfer syntax, the workstation fails to parse the first image received when it encounters the first UN VR attribute, and the association is automatically aborted by the workstation.

We have added several methods to solve this interoperability problem through the MergeCOM-3 tool kit's configuration files. For SCU systems that are exporting UN VR tags to systems that cannot handle them, the following can be done:

- Configure the SCU to only use the Implicit VR Little Endian transfer syntax when exporting objects. This can be done through the use of transfer syntax lists within the `mergecom.app` file or through commenting out the UID definitions for the other transfer syntaxes within the mergecom.pro file.

- Set the **`UNKNOWN_VR_CODE`** configuration option in the mergecom.pro file to 'OB'. This forces unknown VR attributes to be encoded as OB instead of as UN. All implementations can handle OB encoding. There are several drawbacks to this option. If the attributes are encoded as OB, it is harder for these attributes to be converted back to their normal VR. Secondly, this option changes all instances of the UN VR into OB. Systems that can handle the UN VR will now also receive these attributes as OB.

- Set the **`EXPORT_UN_VR_TO_NETWORK`** configuration option to 'No'. This will cause the MergeCOM-3 tool kit to not export attributes encoded as UN VR to the network. This option is being added to release 2.3.0 of the MergeCOM-3 tool kit.

For SCP systems receiving UN VR tags when they cannot handle them, the following can be done:

- Configure the SCP to only negotiate the Implicit VR Little Endian transfer syntax when receiving objects.

With the help of these options, most UN VR problems in the field can be fixed simply by changing configuration values with the MergeCOM-3 tool kit.

Appendix A: Frequently Asked Questions

This appendix lists some frequently asked questions by tool kit users.

1. *I am running the tool kit's sample applications for the first time. I have set the MERGE_INI environment variable to point to the merge.ini file. However, the **MC_Library_Initialization** call is still returning **MC_CONFIG_INFO_ERROR**. What is the cause of this problem?*

   This is usually only a problem under Windows. The merge.ini file contains several entries that point to the locations of the other tool kit configuration files. These entries contain relative pathnames for the other files. If the sample applications are not executed from the directory where the configuration files are located, the tool kit will be unable to find the files and produce this error. Changing these paths to absolute paths will fix the problem.

2. *It is inconvenient to set absolute paths for the various configuration options in the merge.ini and mergecom.pro files that need them. Is there a way to make these pathnames be configurable at run-time?*

   MergeCOM-3 allows the placement of environment variables in these pathnames. This allows setting of a root directory for these pathnames. The following is an example of how this functionality is used in our configuration files:

   ```
   MERGECOM_PRO = $(MERGE_ROOT)\mc3apps\mergecom.pro
   ```

   In this example, `MERGE_ROOT` would be an environment variable set in a similar fashion as the `MERGE_INI` environment variable.

3. *I am testing the sample applications for the first time and cannot get the client (SCU) application to connect to the server (SCP) for any of the sample applications. The **MC_Open_Association** function is returning **MC_SYSTEM_ERROR**. It appears as though the connection is opening, but it is quickly dropped. Why is this happening?*

   As a security measure, the `MC_Wait_For_Association()` function used in SCPs attempts to determine the hostname of SCUs connecting to it. If it cannot determine the remote hostname, it will drop the connection. The `MC_Wait_For_Association()` function uses the local system's host file or its configured domain name server to translate the SCU's IP address into its hostname. By configuring the SCU's hostname in your local hosts file, this problem will be eliminated. Also, the **ACCEPT_ANY_HOSTNAME** configuration value in the mergecom.pro file disables this checking.

4. *What can be done to reduce the memory requirements of the MergeCOM-3 Advanced Tool Kit?*

Performance Tuning

   There are several methods for reducing the memory requirements of MergeCOM-3. The first method is to use either the `MC_Open_Empty_Message()` or `MC_Create_Empty_File()` functions when creating message and file objects. These functions reduce memory by not reading in all of the information needed for validation of messages and files respectively. These functions will also improve performance.

There are several configuration values that reduce MergeCOM-3's memory requirements. The following describes each of these options:

Reducing memory usage for large DICOMDIRs

FORCE_OPEN_EMPTY_ITEM: This configuration option performs the same function as using `MC_Open_Empty_Message()`, except that it is for items. It is especially useful for reducing the amount of memory used when working with large DICOMDIRs.

LARGE_DATA_STORE and **LARGE_DATA_SIZE**: These options control the ability of MergeCOM-3 to store pixel data in temporary files instead of RAM. This functionality is enabled by setting **LARGE_DATA_STORE** to **FILE**, and adjusting **LARGE_DATA_SIZE** to the size of data element that you want spooled to temporary file. Note that this will decrease performance.

5. *What can be done to increase the performance of the MergeCOM-3 Advanced Tool Kit?*

   There are several MergeCOM-3 configuration values that impact performance in different ways. The following is a summary of these options:

   ELIMINATE_ITEM_REFERENCES: This option improves the performance of the `MC_Empty_Message()`, `MC_Free_Message()`, `MC_Empty_File()`, `MC_Free_File()`, and `MC_Free_Item()` functions. This option will disable functionality within the tool kit that causes the tool kit to search all currently open message objects for references to an item that is being freed by one of these calls. This call is especially useful when your application uses very large DICOMDIR files.

   PDU_MAXIMUM_LENGTH: This option sets the maximum sized PDU that the tool kit will receive. If during association negotiation the maximum sized PDU of the system negotiating with the tool kit application is larger than this value, the PDU size will be limited to this value. Increasing this value increases the amount of data that is passed to the TCP/IP level. This will increase network performance of the library.

   WORK_BUFFER_SIZE: This option specifies how the tool kit buffers data before storing it or passing it to a user's callback function. Setting higher values for this option will increase performance.

   TCPIP_RECEIVE_BUFFER_SIZE: This option sets the TCP/IP receive buffer size. Higher values for this buffer generally will increase the network performance of the tool kit for server (SCP) applications. This value should also be slightly larger than the **PDU_MAXIMUM_LENGTH** to increase performance.

   TCPIP_SEND_BUFFER_SIZE: This option sets the TCP/IP send buffer size. Higher values for this buffer generally will increase the network performance of the tool kit for client (SCU) applications. This value should also be slightly larger than the **PDU_MAXIMUM_LENGTH** to increase performance.

   EXPORT_UNDEFINED_LENGTH_SQ: This option determines how MergeCOM-3 encodes sequences within all non-DICOMDIR messages and files. When set to Yes, the sequences are encoded as undefined length. This eliminates the need for MergeCOM-3 to determine the length of sequences and increases performance.

EXPORT_GROUP_LENGTHS_TO_NETWORK: This option determines if MergeCOM-3 encodes group length attributes when writing to the network (if they are included in the message being sent). Setting this option to No increases MergeCOM-3 network performance. This eliminates the need for MergeCOM-3 to determine the length of groups when streaming to the network.

EXPORT_GROUP_LENGTHS_TO_MEDIA: This option determines if MergeCOM-3 encodes group length attributes when writing to files. Setting this option to No increases MergeCOM-3 performance. This eliminates the need for MergeCOM-3 to determine the length of groups when writing to media.

EXPORT_UNDEFINED_LENGTH_SQ_IN_DICOMDIR: This option determines how MergeCOM-3 exports sequence attributes in DICOMDIRs. When set to Yes, the sequences in DICOMDIRs are encoded as undefined length. This greatly improves performance when writing DICOMDIRs because MergeCOM-3 no longer needs to calculate the length of sequence attributes in DICOMDIRs.

6. *Which of the options listed above have the greatest impact on network performance?*

   The **TCPIP_RECEIVE_BUFFER_SIZE** and **TCPIP_SEND_BUFFER_SIZE** configuration options have the greatest impact on network performance. Setting these to higher values directly increases the network performance of MergeCOM-3.

   EXPORT_UNDEFINED_LENGTH_SQ can have a large impact if many sequence attributes are included in the message being transferred.

7. *I am sending 8-bit images with MergeCOM-3, however, after sending the data to another system, the pixel data is byte swapped incorrectly. What is causing this problem?*

   The MergeCOM-3 Advanced Users Manual contains the section "8-bit Pixel Data" which describes this problem. This is typically only a problem on Big Endian machines. To summarize the problem, on big endian machines, we expect 8-bit data to be byte swapped. We do not look at the "bits allocated" and "bits stored" tags to determine that the pixel data itself is 8-bit data, we always treat pixel data `(7fe0,0010)` as OW. The pixel data must be assigned as byte swapped, or the function `MC_Byte_Swap_OBOW()` should be called after setting the pixel data.

8. *I recently upgraded to a new release of the MergeCOM-3 Advanced Tool Kit. Since this upgrade, I have been having problems with the **MC_Set_Value_...()** functions returning **MC_INVALID_TAG**. This code worked before the upgrade. What is causing these problems?*

   The MergeCOM-3 data dictionary changes from release to release. In some cases, the identification number for a particular message type changes. When upgrading, if you do not change all of the data dictionary files, this error will occur. The following files should be upgraded with each release:

   ```
   diction.h

   mergecom.srv

   mrgcom3.msg
   ```

```
mrgcom3.dct
```

9. *What are the differences between the MC_NULL_VALUE, MC_EMPTY_VALUE, and MC_INVALID_TAG return values of the* ***MC_Get_Value_...()*** *functions?*

   The MC_NULL_VALUE return value is used to identify when an attribute within a DICOM message has zero length. DICOM allows attributes that have a Value Type of 2 to be set to zero length when their value is unknown. (An attribute can be set to zero length in MergeCOM-3 with MC_Set_Value_To_NULL.)

   The MC_EMPTY_VALUE and MC_INVALID_TAG return values both mean that a message does not contain a value for the specified attribute. The use of these return values depends on how the message, file, or item containing the attribute was created.

   When using the MC_Open_Message, MC_Create_File or MC_Open_Item function to create an object, MergeCOM-3 loads a list of all of the valid attributes for the object. For these types of objects, the MC_Get_Value_... functions will return MC_EMPTY_VALUE when an attribute defined for the object does not have a value. They will return MC_INVALID_TAG for attributes that are not defined for the object.

   When the object has been created using MC_Open_Empty_Message, MC_Create_Empty_File, or MC_Read_Message, MergeCOM-3 will return MC_INVALID_TAG for any attribute that does not have a value defined.

10. *I am trying to assign the value to a DICOM attribute within a message, but MergeCOM-3 will not allow me to do this. When I call the* ***MC_Set_Value_...()*** *functions, they are returning* ***MC_INVALID_TAG****. How can I add this attribute?*

    This problem occurs when using MC_Open_Message or MC_Create_File to create a message or file of a particular type. These functions restrict the attributes that can be added to a message. Only those attributes that have been defined for the message type (and can be found in our message.txt file) can be assigned to the message or file.

    When adding an attribute that has not been defined for a message, MC_Add_Standard_Attribute can be called to add the tag to the definition of the message. Subsequent calls to the MC_Set_Value…( ) functions will then allow the user to assign the attribute.

11. *How can I encode tags in a message that are invalidly encoded according to DICOM? When I call the* ***MC_Set_Value_From_String()****,it does not return* ***MC_NORMAL_COMPLETION****.*

    The MC_Set_Value_From_String function's return values for invalid DICOM encoding are actually warning return values, and not failures. When MC_INVALID_CHARS_IN_VALUE and MC_INVALID_VALUE_FOR_VR are returned, the value is still encoded. It is a common mistake in MergeCOM-3 applications for it to fail when MC_Set_Value_From_String returns a value other than MC_NORMAL_COMPLETION. If desired, the above return values can be ignored and treated as normal completion.

Appendix B: Unique Identifiers (UIDs)

UIDs provide the capability to identify many different types of items. The purpose of UIDs are to guarantee the uniqueness of these different types of items. DICOM uses UIDs to uniquely identify items such as SOP classes, image instances and network negotiation parameters. Part 5, Section 9 along with Annexes B and C of the DICOM Standard discusses how UIDs are composed, encoded and registered.

Summary of UID Composition

A UID is composed of a number of numeric values as defined by ISO 8824. The following is a typical example of a UID:

```
1.2.840.10008.2.45.1.12345
```

A UID is composed of two parts: a <root> and a <suffix> and has the following form:

```
UID = <root>.<suffix>
```

where <root> is assigned by a registration authority (e.g., ANSI) with the distinguishing component being the organization ID. The <root> portion of the UID uniquely identifies an organization while the <suffix> portion is used to uniquely identify a specific object within the scope of the organization. While the <root> component of the UID stays constant, the <suffix> portion will change in a manner that will provide uniqueness for objects that need UIDs. Note: this implies that the organization is responsible for maintaining the uniqueness of the <suffix>.

For example, using the UID above, <root> = 1.2.840.10008 and <suffix> = 2.45.1.12345. Where the organization ID portion of the <root> (10008) distinguishes organizations from each other.

Note: The above example is typical for UIDs obtained by ANSI during the time when the DICOM standard was first released. The organization ID of 10008 has actually been assigned to NEMA and is used as part of the <root> for DICOM standard UIDs such as SOP Classes, Transfer Syntaxes, etc. For example, vendors creating images need to obtain their own organization ID and cannot use 10008.

For future UIDs, ISO has developed a joint relationship with CCITT and has changed the <root> structure. Therefore, new UIDs from ANSI will no longer be of the form 1.2.840.xxxxx. but are currently assigned using the form, <root> = 2.16.840.1.10008, where, of course, 10008 is the organization ID.

Obtaining a UID

The <root> portion of the UID should be registered by an organization that guarantees global uniqueness. The American National Standards Institute (ANSI) is the registration authority for the United States. Other national registration authorities exist for nations throughout the world such as IBN in Belgium, AFNOR in France, BSI in Great Britain, DIN in Germany, and COSIRA in Canada.

Obtaining a UID From ANSI

ANSI is the registration authority for the US for organization names (i.e., <root>) under the global registration process established by the International Standards Organization (ISO) and the International Telegraph and Telephone Consultative

Committee (CCITT). ANSI's registration service conforms with CCITT X.660 and ISO/IEC 9834-1. The ANSI organization name registration service assigns one name component to the hierarchy defined by CCITT and ISO/IEC.

An organization seeking registration may do so by submitting a Request for Registration application form along with a fee (as of August 1996 the fee is $1,000) to the Registration Coordinator. The Request for Registration application form can be obtained from ANSI by use of the following information:

American National Standards Institute
11 West 42nd Street
New York, New York 10036

TEL: 212.642.4900
FAX: 212.398.0023

Obtaining a UID From Merge eFilm

Merge eFilm has obtained a unique organization ID from ANSI. Merge can also provide a UID that is guaranteed to be unique by using our unique organization ID. If you desire to obtain a unique organization from Merge for a fee (as of August 1998 = $500), contact us at:

Merge eFilm.
1126 South 70th Street, Suite S107B
Milwaukee, WI 53214-3151

TEL: 414.977.4000
FAX: 414.977.4200

Please note that it is not Merge's intent to compete with ANSI or other registration authorities. We encourage organizations to use their services and only offer UID registration for the purpose of ease when using our tool kit (especially for smaller companies).