Merge DICOM Toolkit™ V. 5.8.0

C/C++ USER'S MANUAL

Merge Healthcare
900 Walnut Ridge Drive
Hartland, WI 53029
USA

The symbols glossary is provided electronically at http://www.merge.com/Support/Resources.aspx.

Manufacturer's Address
Merge Healthcare Incorporated
900 Walnut Ridge Drive
Hartland, WI 53029

For assistance, please contact Merge Healthcare Customer Support:

- In North America, call toll free 1-800-668-7990, then select option 2
- International, call Merge Healthcare (in Canada) +1-905-672-7990, then select option 2
- Email MDTsupport@merge.com or MC3Support@ca.ibm.com

Part	Date	Revision	Description
COM-3440	December 2018	1.0	Updated bi-annually

Contents

Overview

This User's Manual is intended for developers of medical imaging applications who are using the Merge DICOM Toolkit to provide DICOM network or media functionality.

The Merge DICOM Toolkit supplies you with a powerful and simplified interface to DICOM. It lets you focus on the important details of your application and the immediate needs of your end users, rather than the complex and often 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 Merge DICOM Toolkit.

The DICOM Standard

The Digital Imaging and Communications in Medicine (DICOM) 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.

The DICOM Standard 2011 edition is composed of thousands of pages over 18 separate parts (parts 9 and 13 have been retired). 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

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

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

Part 16: DICOM Content Mapping Resource

Part 17: Explanatory Information

Part 18: Web Services

Part 19: Application Hosting

Part 20: Transformation of DICOM to and from HL7 Standards

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, where two DICOM applications mutually agree on 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. (Note that IPv4 and IPv6 are supported by DICOM and by Merge DICOM Toolkit.) This is termed the DICOM Upper Layer Protocol (DICOM UL).

Part 9 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 the Media Storage Application Profiles that standardize 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 displaying 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).

Part 19 defines an API such that a 'plug-in' Hosted Application written to the API would be able run in any environment provided by a Hosting System implementing the API.

Part 20 specifies transformations of DICOM data to and from HL7 standards.

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.

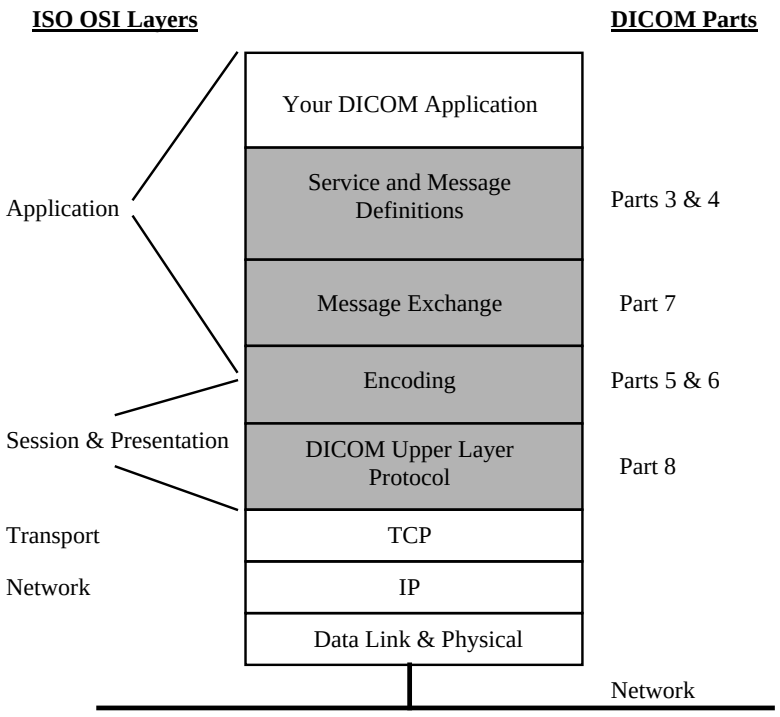

Figure 1: The DICOM Protocol Stack

Where to get the DICOM Standard

As a user of this toolkit, you should have access to the DICOM Standard. The Merge DICOM Toolkit 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 of the standard is available in PDF format and can be downloaded from NEMA's public ftp site at: ftp://medical.nema.org/medical/Dicom/2011.

Special Note!

Please note that 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 which you cannot find, obtain the proper supplement from NEMA. Other additions to the Standard (e.g., new image objects or documents) are also published as supplements. NEMA also makes all supplements to the standard freely available on their FTP server. You can reference these supplements at: ftp://medical.nema.org/medical/Dicom/Final/.

The Merge DICOM Toolkit

The Merge DICOM Toolkit provides a generalized implementation of DICOM in an ANSI-C Function Library which 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. The Merge DICOM Toolkit implements 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 application's underlying file system.

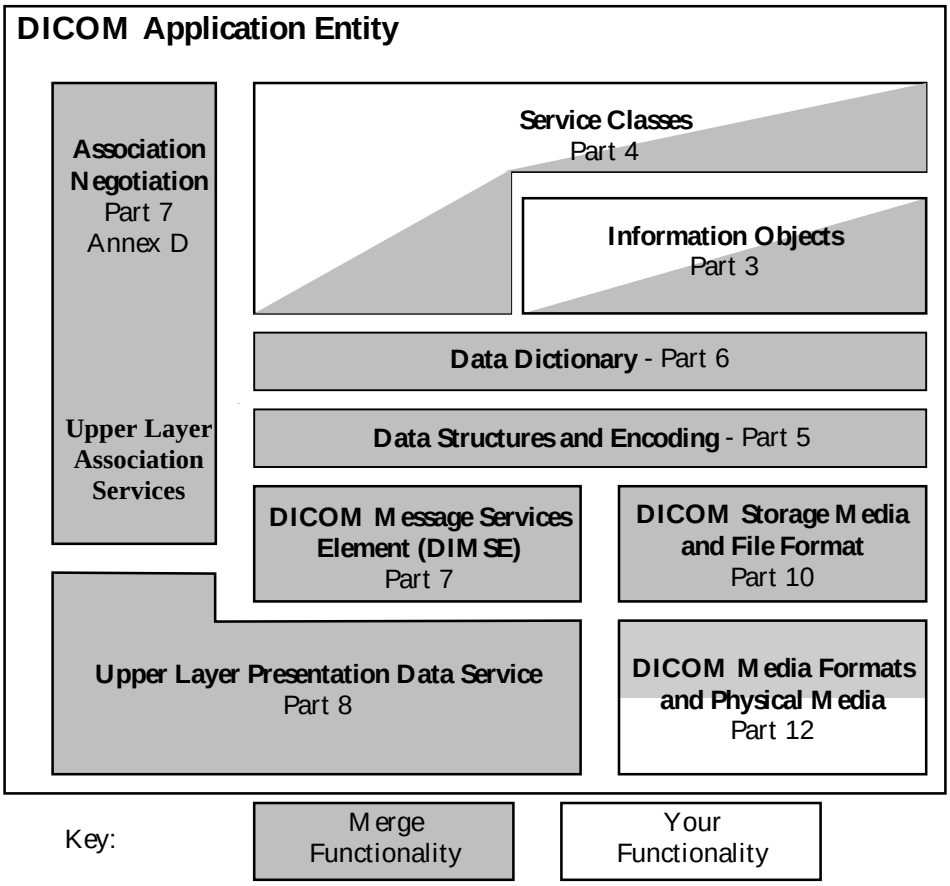

Figure 2: The DICOM Application Layer

The DICOM Toolkit also supplies useful utility programs for testing a DICOM network connection, creating sample DICOM messages and writing them to a file, and 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 DICOM Toolkit 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 Merge DICOM Toolkit Extended Toolkit is available.

Development Platform Requirements

To use the Merge DICOM Toolkit Library, you must run on a toolkit supported computing platform. The Toolkit was designed to be portable and is available for many platforms (e.g., Sun Solaris 8 Sparc, Sun Solaris 10 Intel, Linux, Windows XP/Vista/7/8, MacOS X, Android). If it is not currently available for your target platform, please contact Merge Healthcare. 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 2000/2003/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 toolkit 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 DICOM Toolkit 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 Merge DICOM Toolkit 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.

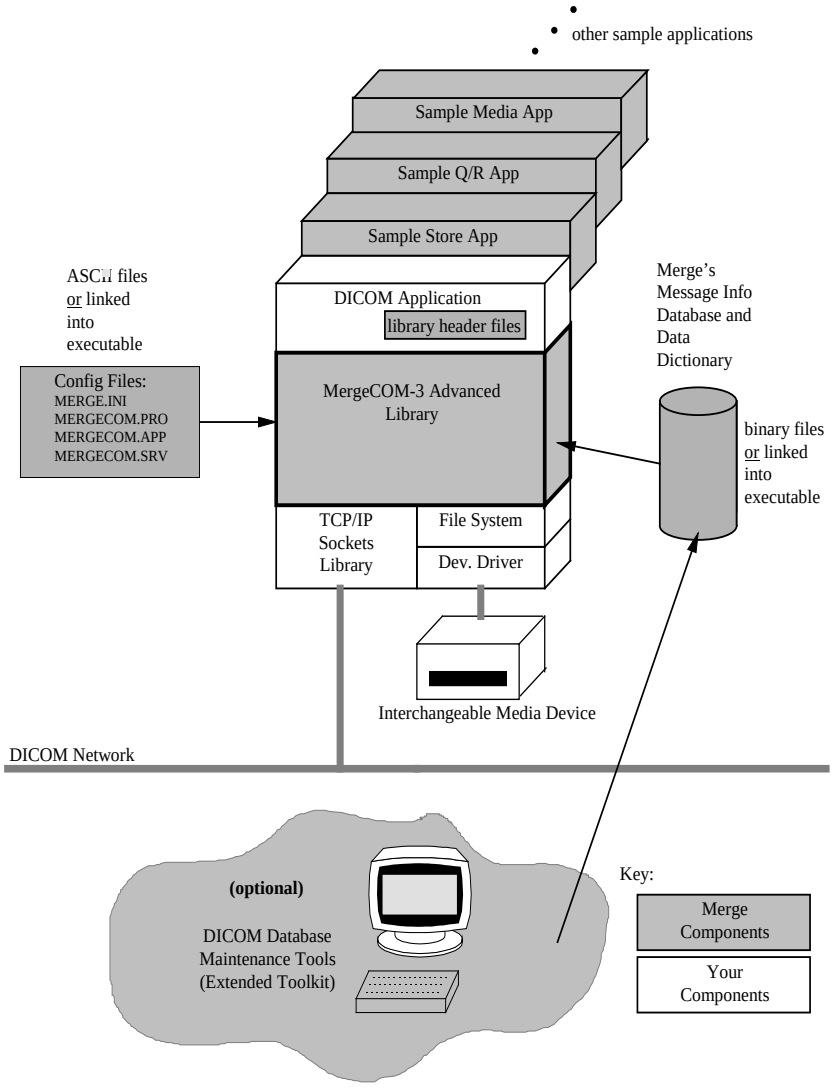

Figure 3: Merge DICOM Toolkit Library Organization

Header Files

Your applications interface to the DICOM Toolkit 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 toolkit 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 DICOM Toolkit 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 toolkit library functionality.

Merge DICOM Toolkit Library

The Merge DICOM Toolkit 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 DICOM Toolkit. In the process of servicing networking calls, the DICOM Toolkit 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 Merge DICOM Toolkit library has been carefully designed to be re-entrant and has been validated to be thread-safe on several multi-threading capable platforms. Note, however, that with only a few exceptions, Merge DICOM Toolkit assumes that objects are only accessed from one thread at a time. For instance, Merge DICOM Toolkit assumes that only a single thread will manipulate a message object at one time. Please check the platform notes to see if a platform supports multi-threading.

Shared libraries or dynamic link libraries (DLL's) are normally supplied by Merge DICOM Toolkit for platforms which support them.

When a Merge DICOM Toolkit Application is first run, it reads in its configuration files; usually named `merge.ini`, `mergecom.app`, `mergecom.pro`, and `mergecom.srv`. Toolkit 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 toolkit configuration is unlikely to be changed or it is desirable to make these changes within the running application, the toolkit 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 Merge DICOM Toolkit 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

Merge DICOM Toolkit 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 toolkit 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 DICOM Toolkit 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 Merge DICOM Toolkit 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 toolkit are sample applications in ANSI-C source code form and a `Makefile` that compiles and links the sample applications with the toolkit library. Sample client and server applications are supplied for Storage, Query/Retrieve, and Print services. Also, a compression sample and DICOM File Service application are 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 DICOM Toolkit API to perform DICOM services over a network.

Merge DICOM Toolkit Extended Toolkit

Merge Healthcare 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 DICOM Toolkit, 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 toolkit current. This also reduces the number of changes that must be made in the core DICOM Toolkit library over time.

The extended version of this toolkit makes some of these tools available to application developers who need to significantly extend the standard with private attribute and private service definitions. The files for the extended version are packaged with the standard toolkit. 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 toolkit at run time. In this way, you can extend the toolkit to validate your own private attributes and services.

Documentation Roadmap

The Merge DICOM Toolkit documentation is structured as shown 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 DICOM Toolkit. 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 DICOM Toolkit. This includes the Application Programming Interface (API), toolkit configuration, the runtime object database, and status logging. The DICOM Message Database Manual is an optional extension that describes the organization of the Merge DICOM Toolkit 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 DICOM Toolkit. 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 DICOM Toolkit on your target platform are specified in Platform Notes. This includes supported compilers, compiler options, link options, configuration, and run-time related issues.

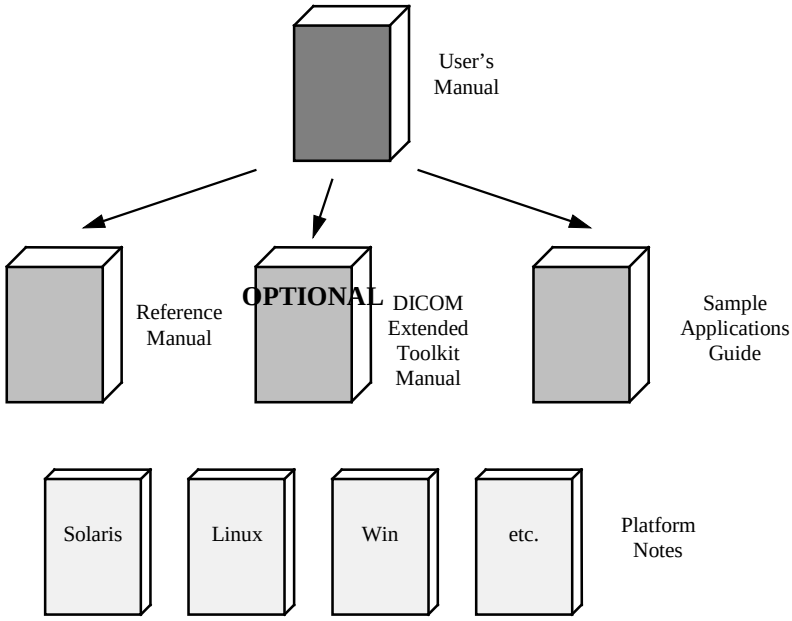

Figure 4: Merge DICOM Toolkit Documentation Roadmap

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.

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.**

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.

Understanding DICOM

The eighteen 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 DICOM Toolkit.

General Concepts

Some key concepts that must be understood to use the DICOM Toolkit 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 (AEs)**. 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 (SCUs)** and **Service Class Providers (SCPs)**. 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**. 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.

Table 1: DICOM Services Classes and their Component Services

Service Class	Services	Description
Verification	Verification	Verifies application level communication between DICOM application entities (AE's).
Storage	*12-lead ECG Waveform Storage* *Advanced Blending Presentation State Storage* *Ambulatory ECG Waveform Storage* *Arterial Pulse Waveform Storage* *Autorefraction Measurements Storage* *Basic Structured Display Storage* *Basic Text Structured Reporting*	Transfer of medical images and related standalone data between DICOM application entities, either over a network or using interchangeable

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

Service Class	Services	Description
	Basic Voice Audio Waveform Storage *Blending Softcopy Presentation State Storage* *Breast Projection X-Ray Image Storage - For Presentation* *Breast Projection X-Ray Image Storage - For Processing* *Breast Tomosynthesis Image Storage* *Cardiac Electrophysiology Waveform Storage* *Chest CAD SR* *Colon CAD SR* *Color Softcopy Presentation State Storage* *Computed Radiography Image Storage* *Comprehensive Structured Reporting* *CT Image Storage* *Deformable Spatial Registration Storage* *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* *Digital X-Ray Image Storage – For Presentation* *Digital X-Ray Image Storage – For Processing* *Encapsulated CDA Storage* *Encapsulated PDF Storage* *Encapsulated STL Storage* *Enhanced CT Image Storage* *Enhanced MR Image Storage* *Enhanced MR Color Image Storage* *Enhanced PET Image Storage* *Enhanced Structured Reporting* *Enhanced US Volume Storage* *Enhanced XA Image Storage* *Enhanced XRF Image Storage* *General Audio Waveform Storage* *General ECG Waveform Storage* *Generic implant Template Storage* *Grayscale Softcopy Presentation State Storage* *Hemodynamic Waveform Storage* *Implant Assembly Template Storage* *Implant Template Group Storage* *Implantation Plan SR Document Storage* *Intraocular Lens Calculations Storage* *Intravascular Optical Coherence Tomography Image Storage – For Presentation* *Intravascular Optical Coherence Tomography Image Storage – For Processing* *Keratometry Measurements Storage* *Key Object Selection* *Lensometry Measurements Storage* *Macular Grid Thickness and Volume Report* *Mammography CAD SR* *MR Image Storage*	media.

Service Class	Services	Description
	MR Spectroscopy 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* *Multiple Volume Rendering Volumetric Presentation State Storag* *Nuclear Medicine Image Storage* *Ophthalmic 8 bit Photography Image Storage* *Ophthalmic 16 bit Photography Image Storage* *Ophthalmic Axial Measurements Storage* *Ophthalmic Optical Coherence Tomography B-scan Volume Analysis Storage* *Ophthalmic Optical Coherence Tomography En Face Image Storage* *Ophthalmic Tomography Image Storage* *Ophthalmic Visual Field Static Perimetry Measurements Storage* *Patient Radiation Dose SR Storage* *Parametric Map Storage* *Positron Emission Tomography Image Storage* *Procedure Log* *Protocol Approval Storage* *Pseudo-Color Softcopy Presentation State Storage* *Raw Data Storage* *Real World Value Mapping Storage* *RT Beams Delivery Instruction Storage* *RT Beams Treatment Record Storage* *RT Brachy Treatment Record Storage* *RT Dose Storage* *RT Image Storage* *RT Ion Beams Treatment Record Storage* *RT Ion Plan Storage* *RT Plan Storage* *RT Structure Set Storage* *RT Treatment Summary Record Storage* *Secondary Capture Image Storage* *Segmentation Storage* *Segmented Volume Rendering Volumetric Presentation State Storage* *Spatial Fiducials Storage* *Spatial Registration Storage* *Spectacle Prescription Report Storage* *Stereometric Relationship Storage* *Subjective Refraction Measurements Storage* *Surface Segmentation Storage* *Ultrasound Image Storage* *Ultrasound Multi-frame Image Storage* *Video Endoscopic Image Storage* *Video Microscopic Image Storage*	

Service Class	Services	Description
	Video Photographic Image Storage *Visual Acuity Measurements Storage* *VL Endoscopic Image Storage* *VL Microscopic Image Storage* *VL Photographic Image Storage* *VL Slide-Coordinates Microscopic Image Storage* *VL Whole Slide Microscopy Image Storage* *Volume Rendering Volumetric Presentation State Storage* *Wide Field Ophthalmic Photography 3D Coordinates Image Storage* *Wide Field Ophthalmic Photography Stereographic Projection Image Storage* *XA/XRF Grayscale Softcopy Presentation State Storage* *X-Ray Angiographic Image Storage* *X-Ray Radiation Dose SR* *X-Ray Radiofluoroscopic Storage* *X-Ray 3D Angiographic Image Storage* *X-Ray 3D Craniographic Image 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 Protocol Approval Information Model Find Protocol Approval Information Model Move Protocol Approval Information Model 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.

Service Class	Services	Description
Basic Worklist Management	Modality Worklist Find	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.
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	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.

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 DICOM Toolkit. 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*.

Table 2: DICOM Commands

DICOM Commands	Description
C-STORE	Transfer an object instance to a remote AE.
C-GET	Retrieve from a remote AE object instance(s) whose attributes match a specified set of attributes.
C-MOVE	Move object instance(s) whose attributes match a specified set of attributes from a remote AE to yet another remote AE (or possibly your own AE - which would be another form of retrieval).

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

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

DICOM Commands	Description
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.

Where your application takes over...

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 Merge DICOM Toolkit Library leaves off, and your Application begins. While Merge DICOM Toolkit 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 toolkit. While you will see that the toolkit 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 Merge DICOM Toolkit 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).

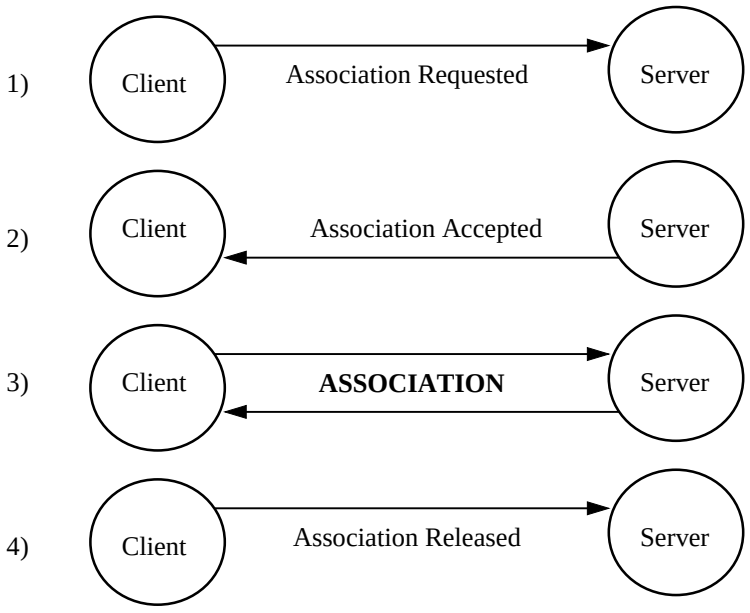

Figure 5: A Successful DICOM Association

Messages

Service-Command Pair

Once an association is established, services are performed by AEs 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.

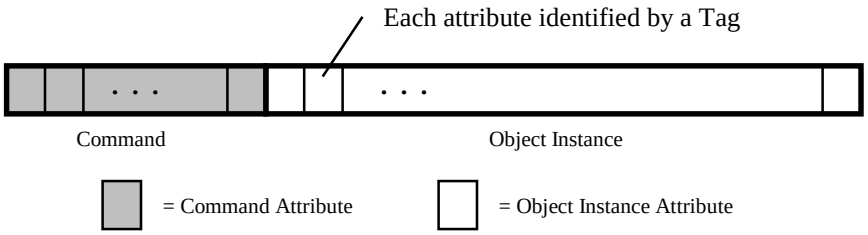

Figure 6: A DICOM Message

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

VRs defined by DICOM are listed in *Table 3*. You should refer to Part 5 of the standard for a detailed description of their allowed values and formats.

Table 3: DICOM Value Representations (VR's)

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		

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 *Table 1* and *Table 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 Merge DICOM Toolkit 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).

Table 4: DICOM Value Types (VT's)

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.

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. Merge DICOM Toolkit 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 (see *Figure 7*).

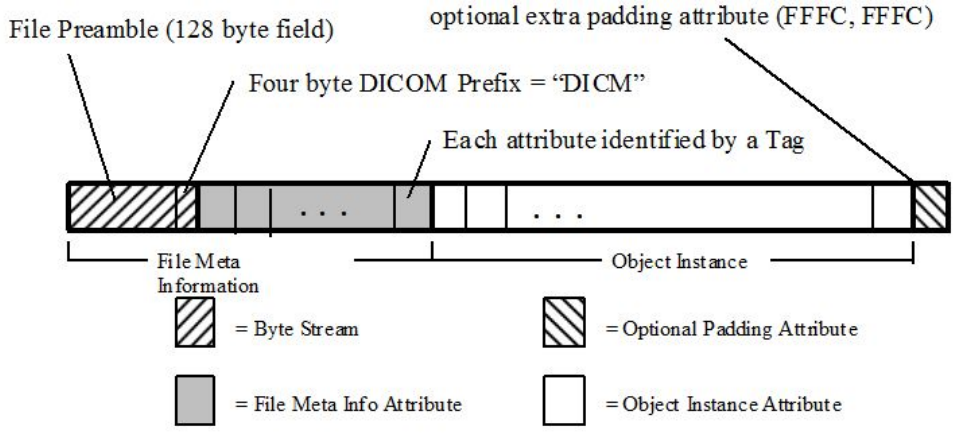

Figure 7: A DICOM File

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: 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.

Table 5: Service-Command Pairs Specifying Object Instances that can be Stored in a DICOM File

Service	Command
12-lead ECG Waveform Storage	C-STORE
Advanced Blending Presentation State Storage	C-STORE
Ambulatory ECG Waveform Storage	C-STORE
Arterial Pulse Waveform Storage	C-STORE
Autorefraction Measurements Storage	C-STORE
Basic Color Image Box	N-SET
Basic Film Box	N-CREATE
Basic Film Session	N-CREATE
Basic Grayscale Image Box	N-SET
Basic Structured Display Storage	C-STORE
Basic Text Structured Reporting	C-STORE
Basic Voice Audio Waveform Storage	C-STORE
Blending Softcopy Presentation State Storage	C-STORE

Service	Command
Breast Projection X-Ray Image Storage - For Presentation	C-STORE
Breast Projection X-Ray Image Storage - For Processing	C-STORE
Breast Tomosynthesis Image Storage	C-STORE
Cardiac Electrophysiology Waveform Storage	C-STORE
Color Softcopy Presentation State Storage	C-STORE
Comprehensive Structured Reporting	C-STORE
Computed Radiography Image Storage	C-STORE
CT Image Storage	C-STORE
Chest CAD SR	C-STORE
Colon CAD SR	C-STORE
Deformable Spatial Registration Storage	C-STORE
Detached Interpretation Management	N-GET
Detached Patient Management	N-GET
Detached Results Management	N-GET
Detached Study Management	N-GET
Detached Study Component Management	N-GET
Detached Visit Management	N-GET
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
Encapsulated CDA Storage	C-STORE
Encapsulated PDF Storage	C-STORE
Encapsulated STL Storage	C-STORE
Enhanced CT Image Storage	C-STORE

Service	Command
Enhanced MR Color Image Storage	C-STORE
Enhanced MR Image Storage	C-STORE
Enhanced PET Image Storage	C-STORE
Enhanced Structured Reporting	C-STORE
Enhanced US Volume Storage	C-STORE
Enhanced XA Image Storage	C-STORE
Enhanced XRF Image Storage	C-STORE
General Audio Waveform Storage	C-STORE
General ECG Waveform Storage	C-STORE
Generic implant Template Storage	C-STORE
Grayscale Softcopy Presentation State Storage	C-STORE
Hanging Protocol Storage	C-STORE
Hemodynamic Waveform Storage	C-STORE
Implant Assembly Template Storage	C-STORE
Implant Template Group Storage	C-STORE
Implantation Plan SR Document Storage	C-STORE
Intraocular Lens Calculations Storage	C-STORE
Intravascular Optical Coherence Tomography Image Storage – For Presentation	C-STORE
Intravascular Optical Coherence Tomography Image Storage – For Processing	C-STORE
Keratometry Measurements Storage	C-STORE
Key Object Selection	C-STORE
Lensometry Measurements Storage	C-STORE
Macular Grid Thickness and Volume Report	C-STORE
Mammography CAD SR	C-STORE
Media Storage Directory Storage	C-STORE*
MR Image Storage	C-STORE

Service	Command
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
Multiple Volume Rendering Volumetric Presentation State Storage	C-STORE
Nuclear Medicine Image Storage	C-STORE
Ophthalmic 8 bit Photography Image Storage	C-STORE
Ophthalmic 16 bit Photography Image Storage	C-STORE
Ophthalmic Axial Measurements Storage	C-STORE
Ophthalmic Optical Coherence Tomography B-scan Volume Analysis Storage	C-STORE
Ophthalmic Optical Coherence Tomography En Face Image Storage	C-STORE
Ophthalmic Tomography Image Storage	C-STORE
Ophthalmic Visual Field Static Perimetry Measurements Storage	C-STORE
Parametric Map Storage	C-STORE
Patient Radiation Dose SR Storage	C-STORE
Positron Emission Tomography Image Storage	C-STORE
Procedure Log	C-STORE
Protocol Approval Storage	C-STORE
Pseudo-Color Softcopy Presentation State Storage	C-STORE
Raw Data Storage	C-STORE
Real World Value Mapping Storage	C-STORE
Respiratory Waveform Storage	C-STORE
RT Beams Delivery Instruction Storage	C-STORE
RT Beams Treatment Record Storage	C-STORE
RT Brachy Treatment Record Storage	C-STORE

Service	Command
RT Dose Storage	C-STORE
RT Ion Beams Treatment Record Storage	C-STORE
RT Ion Plan 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
Segmentation Storage	C-STORE
Segmented Volume Rendering Volumetric Presentation State Storage	C-STORE
Subjective Refraction Measurements Storage	C-STORE
Surface Segmentation Storage	C-STORE
Spatial Registration Storage	C-STORE
Spatial Fiducials Storage	C-STORE
Spectacle Prescription Report 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
Stereometric Relationship 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
Visual Acuity Measurements Storage	C-STORE
VL Endoscopic Image Storage	C-STORE

Service	Command
VL Microscopic Image Storage	C-STORE
VL Photographic Image Storage	C-STORE
VL Slide-Coordinates Microscopic Image Storage	C-STORE
VL Whole Slide Microscopy Image Storage	C-STORE
Volume Rendering Volumetric Presentation State Storage	C-STORE
Wide Field Ophthalmic Photography 3D Coordinates Image Storage	C-STORE
Wide Field Ophthalmic Photography Stereographic Projection Image Storage	C-STORE
XA/XRF Grayscale Softcopy Presentation State Storage	C-STORE
X-Ray Angiographic Image Storage	C-STORE
X-Ray Radiofluoroscopic Image Storage	C-STORE
X-Ray Radiation Dose SR Storage	C-STORE
X-Ray 3D Angiographic Image Storage	C-STORE
X-Ray 3D Craniofacial Image Storage	C-STORE

* Merge DICOM Toolkit defines a C-STORE command for the Media Storage Directory (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 IDs (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 and file set ID 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 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.

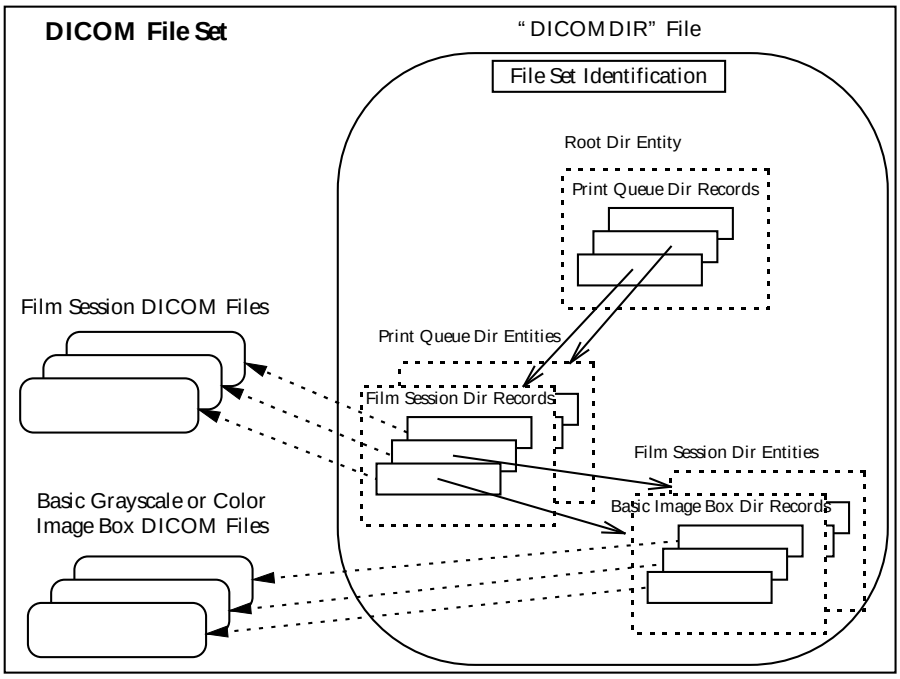

Figure 8: A DICOM Directory File (DICOMDIR) within a DICOM File Set

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.

Table 6: Allowed Directory Entity

Directory Record Type	Record Types which may be included in the next lower-level Directory Entity
(Root Directory Entity)	PATIENT, HANGING PROTOCOL, PALETTE, IMPLANT, IMPLANT ASSY, IMPLANT GROUP, PRIVATE
PATIENT	STUDY, HL7 STRUC DOC, PRIVATE
STUDY	SERIES, PRIVATE
SERIES	IMAGE, RT DOSE, RT STRUCTURE SET, RT PLAN, RT TREAT RECORD, PRESENTATION, WAVEFORM, SR DOCUMENT, KEY OBJECT DOC, SPECTROSCOPY, RAW DATA, REGISTRATION, FIDUCIAL, ENCAP DOC, VALUE MAP, STEREOMETRIC, PLAN, MEASUREMENT, SURFACE, PRIVATE
IMAGE	PRIVATE
RT DOSE	PRIVATE
RT STRUCTURE SET	PRIVATE
RT PLAN	PRIVATE
RT TREAT RECORD	PRIVATE
PRESENTATION	PRIVATE
WAVEFORM	PRIVATE
SR DOCUMENT	PRIVATE
KEY OBJECT DOC	PRIVATE
SPECTROSCOPY	PRIVATE
RAW DATA	PRIVATE
REGISTRATION	PRIVATE
FIDUCIAL	PRIVATE
HANGING PROTOCOL	PRIVATE
ENCAP DOC	PRIVATE
HL7 STRUC DOC	PRIVATE
VALUE MAP	PRIVATE
STEREOMETRIC	PRIVATE

Directory Record Type	Record Types which may be included in the next lower-level Directory Entity
PALETTE	PRIVATE
IMPLANT	PRIVATE
IMPLANT ASSY	PRIVATE
IMPLANT GROUP	PRIVATE
PLAN	PRIVATE
MEASUREMENT	PRIVATE
SURFACE	PRIVATE
PRIVATE	PRIVATE, (any of the above as privately defined)

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:

Table 7: DICOM File Services

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.

The Merge DICOM Toolkit supplies families of functions that perform the first two file services. The Toolkit 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 Application Entities 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.

Table 8: Media Application Operations and 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

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); and
- 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 DICOM Toolkit. Most parameters are configurable by your application. The conformance statement for the Merge DICOM Toolkit 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 Merge DICOM Toolkit Extended Toolkit to assist in defining these services so that they can be supported by the toolkit.

Using Merge DICOM Toolkit

You can use the Merge DICOM Toolkit 'out of the box' by using its supplied utility programs and sample applications. In this section we discuss how to configure the toolkit and to use the utility programs. Use of the sample applications is described in the sample application guides. Later, we discuss how to develop your own DICOM applications using the Merge DICOM Toolkit library.

Configuration

Merge DICOM Toolkit 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:

- initialization file
- application profile
- system profile
- service profile

Each of these configuration files is 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 Merge DICOM Toolkit Initialization File (usually called `merge.ini`) provides the DICOM Toolkit 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 Merge DICOM Toolkit initialization files, specifies characteristics of the message/error log kept by the DICOM Toolkit 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.

See the Appendix B: Configuration Parameters of the Reference Manual for details on the toolkit's configuration.

Message Logging

Merge DICOM Toolkit 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 toolkit 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 include 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).

The trace logging levels are intended strictly for debugging purposes. If left on, they can seriously degrade toolkit performance. In particular, the T2, T7 and T9 levels should be turned off in normal operation.

An excerpt from a Merge DICOM Toolkit 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 many DICOM Toolkit computing platforms, additional information is logged, such as process and thread id numbers identifying where the message was generated.

Utility Programs

The Merge DICOM Toolkit 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 Merge DICOM Toolkit 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. Note that on platforms that supporting the Pegasus libraries for compression, the utility can also compress and decompress images. The syntax for the `mc3conv` utility is the following:

```
mc3conv input_file output_file [-t <syntax>] [-p] [-m] [-x]
      [-s <syntax>] [-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
                    'ib' for implicit big endian
                    'jb' for jpeg baseline
                    'je' for jpeg extended 2_4
                    'jl' for jpeg lossless hier 14
                    'j2lo' for jpeg 2000 lossless only
                    'j2' for jpeg 2000
                    'rle' for rle
-m    Specify format of 'output_file' to be DICOM-3 media
      (Part 10) format.
-s    Specify transfer syntax for 'input_file'
-p    Just extract the pixel data from 'input_file' into
      'output_file'.  If multiframe and encapsulated,
      '_x' is appended to 'output_file' for each frame
-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"
-x    Specify format of 'output_file' to be XML format
-h    Get help - print this usage description

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

mc3echo

Do a DICOM 'ping'

The `mc3echo` utility validates application level communication between two DICOM AEs. 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 DICOM Toolkit 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 is
          "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>] [-p] [-q]
      [-t <syntax>]

<filename> Filename containing message to validate
-e         Display error messages only (optional)
-w         Display error and warning messages (optional,
           default)
-i         Display informational, error, and warning messages
           (optional)
-s <serv>  Force the message to be validated against service
           name "serv", used along with '-c' (optional)
-c <cmd>   Force the message to be validated against command
           name "cmd", used along with '-s' (optional)
-q         Disable prompting for correct service-command pairs
           (optional)
-p         Use message template to validate message against
           (optional, maintained for backward compatibility
           only)
-t        Specify transfer syntax of message, where
           syntax = "il" (implicit little endian)
                  = "el" (explicit little endian)
                  = "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 <file> 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 <file> 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 Merge DICOM Toolkit 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 DICOMDIRs. DICOM Toolkit 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 Merge DICOM Toolkit 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 trade-off 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 Merge DICOM Toolkit 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. Tooltkit allows you to initialize an internal system exception handler and add the user-defined exception handler, which will be called in case of severe system error or signal. These functions are detailed in the Reference Manual.

If you change the configuration of the library at runtime using the `MC_Set_..._Config_Value()` 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 Return Codes

You should always check the return code from any function call to Merge DICOM Toolkit 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 toolkit 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 access files. These initialization functions are generated by two tools supplied with Merge DICOM Toolkit: `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 toolkit 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 Merge DICOM Toolkit. The toolkit 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 toolkit 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 DICOM Toolkit 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 Merge DICOM Toolkit 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

There are two methods to wait for associations for server applications. The first is to use the `MC_Wait_For_Association()` call. The second method is to use the `MC_Wait_For_Connection()` call in conjunction with the `MC_Process_Association_Request()` call. The `MC_Wait_For_Association()` call is the traditional method for waiting for associations with Merge DICOM Toolkit. The `MC_Wait_For_Connection()` call was introduced to address an inherent design problem with `MC_Wait_For_Association()`: specifically, if a remote system connecting is slow in sending its association negotiation information, `MC_Wait_For_Assocation()` does not process any other connections while it is waiting for this information. It is recommended that `MC_Wait_For_Connection()` be used instead for any new applications being developed with Merge DICOM Toolkit.

When using the `MC_Wait_For_Association()` call, you can specify a timeout 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 Merge DICOM Toolkit Reference manual for more information about the `MC_Wait_For_Secure_Association()` call.

Server Side Example of `MC_Wait_For_Association()`:

```
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
 */
```

MC_Wait_For_Connection

When using the `MC_Wait_For_Connection()` call in a server application, the time to wait is specified and a pointer to a socket variable is supplied. Upon successful completion, the socket for the incoming connection is returned and must be passed to `MC_Process_Association_Request( )` to actually process the connection. After `MC_Wait_For_Connection()` completes, a typical server application will create a child thread or process to handle the incoming connection and return back to calling `MC_Wait_For_Connection()` to wait for the next incoming association.

You also have the option of using the `MC_Process_Secure_Association_Request()` call for processing secure associations. 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 Merge DICOM Toolkit Reference manual for more information about the `MC_Process_Secure_Association_Request()` call.

Server Side Example of `MC_Wait_For_Connection()`:

```
MC_SOCKET theSocket;
status = MC_Wait_For_Connection(-1, &theSocket);
if(status != MC_NORMAL_COMPLETION)
{
     printf("\tError on MC_Wait_For_Connection:\n");
     printf("\t\t%s\n", MC_Error_Message(status));
     printf("\t\tProgram aborted.\n");
```

```
        abort();
    }

    status = MC_Process_Association_Request(theSocket,
            "Service_List_1", &calledApplicationID, &associationID);
    if(status != MC_NORMAL_COMPLETION)
    {
        printf("\tMC_Process_Association_Request error:\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)

Merge DICOM Toolkit 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 toolkit, although the toolkit will not do the actual compression and decompression. Encoding of this pixel data is discussed below.

For DICOM Toolkit users, the toolkit 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 Merge DICOM Toolkit Application Profile. When support within a server (SCP) is limited to the three non-encapsulated DICOM transfer syntaxes, the toolkit 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 Merge DICOM Toolkit are also defined through the Merge DICOM Toolkit Application Profile. The following is a typical client (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 context. 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 Merge DICOM Toolkit places on them during association negotiation. In this case, Merge DICOM Toolkit 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.

Dynamic Service Lists

Service lists defined at runtime

In addition to defining service lists in the Application Profile, Merge DICOM Toolkit has mechanisms to define service lists and transfer syntax lists at run-time. A number of functions exist to create transfer syntaxes and service lists in various formats. The following example shows how to create two transfer syntax lists and a service list.

```
TRANSFER_SYNTAX syntaxIds[3];
char *serviceList[3];

syntaxIds [0] = JPEG_BASELINE;
syntaxIds [1] = (TRANSFER_SYNTAX)0;

status = MC_NewSyntaxList("BASELINE",syntaxIds);
if (status != MC_NORMAL_COMPLETION)
   BadStatus(status);

testSyntaxIds[0] = EXPLICIT_LITTLE_ENDIAN;
testSyntaxIds[1] = IMPLICIT_LITTLE_ENDIAN;
testSyntaxIds[2] = (TRANSFER_SYNTAX)0;

status = MC_NewSyntaxList("DEFAULT",syntaxIds);
if (status != MC_NORMAL_COMPLETION)
   BadStatus(status);

status = MC_NewServiceFromName("CT_JPEG", "STANDARD_CT",
       "BASELINE",0,1);
if(status != MC_NORMAL_COMPLETION)
   BadStatus(status);

status = MC_NewServiceFromName("CT_DEFAULT", "STANDARD_CT",
       "DEFAULT",0,1);
if(status != MC_NORMAL_COMPLETION)
   BadStatus(status);

serviceList[0] = "CT_JPEG";
serviceList[1] = "CT_DEFAULT";
serviceList[2] = 0;

status = MC_NewProposedServiceList("TEST_LIST1", serviceList);
if(status != MC_NORMAL_COMPLETION)
   BadStatus(status);
```

In this example, `MC_NewSyntaxList()` is used to create a transfer syntax list. This routine is passed an array of transfer syntaxes that are placed in the list and the user specifies a name for the syntax list. Similar to the creation of syntax lists in the application profile, the order in which transfer syntaxes are defined in the list dictates the priority Merge DICOM Toolkit places on the transfer syntaxes when negotiating an association.

The `MC_NewServiceFromName()` routine is used to create individual services within a service list. Each service in a dynamic service list must be created in this

way. The `MC_NewProposedServiceList()` routine can then be used to create a new service list consisting of services created with `MC_NewServiceFromName()`. It is passed an array of pointers to the names of services to be contained in the service list.

In addition to these functions, Merge DICOM Toolkit supplies a number of other routines for freeing service lists that are created dynamically and for creating service lists. Please reference the Merge DICOM Toolkit Reference Manual for further details on these functions.

Message Objects

Objects are useful

Merge DICOM Toolkit 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 toolkit to encapsulate related data while hiding unnecessary details from the application developer. IDs also provide a convenient shorthand when making calls to the Merge DICOM Toolkit that operate on or make use of these objects.

A majority of the functionality supplied with the DICOM Toolkit deals with building, parsing, validation, and exchange of DICOM messages, files, and items. Your applications deal with network messages in Merge DICOM Toolkit as message objects, and DICOM files as file objects. Item objects are used for attributes that are of VR Sequence of Items (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 DICOM Toolkit 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 9*. 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.

Table 9: The MC_Set_Value Family of Functions

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 or UT (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 the `MC_Register_Callback_Function( )`. See the later discussion of Callback 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 10*). 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 an `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.

Table 10: Acceptable MC_Set_Value/VR combinations

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, UT
`MC_Set_Value_From_Buffer`	UNKNOWN_VR

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;
}
session_uid = "1.2.840.10008.75.89";

status = MC_Set_Value_From_String(messageID, 0x00001000,
       Session_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(&messageID);
```

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 Merge DICOM Toolkit 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 toolkit library will repeatedly call back `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 `*UserInfo` 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 Merge DICOM Toolkit 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 Merge DICOM Toolkit 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 Merge DICOM Toolkit. 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 11*. 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.

Table 11: The MC_Get_Value Family of Functions

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, FD, UN or UT (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.

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 12*). 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 an `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`,…).

Table 12: Acceptable MC_Get_Value/VR combinations

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, UI, UL, US, UT, SQ
`MC_Get_Value_To_Function`	OB, OW, OF, SL, SS, UL, US, AT, FL, FD, UN, UT
`MC_Get_Value_To_Buffer`	UNKNOWN_VR, OB, OW

Note: The same table of acceptable conversions applies for the `MC_Get_Next_Value_To...`, `MC_Get_pValue_To...` and `MC_Get_Next_pValue_To...` families of functions.

The user should be aware that, while the toolkit makes reasonable efforts to ensure correct conversions between data representations, the `MC_Get_Value_To...` functions should be used with caution in some circumstances.

For instance, loss of precision is possible when `MC_Get_Value_To_String` is called to return the value of a float or a double attribute as a string. Let's assume the value of the attribute (of VR=FL) is 123.456789. Internally, the toolkit converts the value to string using the %g format specification. The returned result is the "123.457" string (the rounded value with the default precision).

A better approach in this case would be to retrieve the attribute in its native representation using `MC_Get_Value_To_Float` and then convert it to string outside of the toolkit, using the desired precision.

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 a binary attribute or an attribute whose value representation is unknown.

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 section in the Sample Application Guide 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 Merge DICOM Toolkit 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 toolkit library will repeatedly call back `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 `*UserInfo` 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 Merge DICOM Toolkit 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 Merge DICOM Toolkit 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.

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 Merge DICOM Toolkit. 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, Merge DICOM Toolkit 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 toolkit assumes that the OW pixel data is encoded in the host system's native endian format as defined by DICOM. *Figure 9* describes how 8-bit pixel data is encoded in an OW buffer for both big and little endian formats.

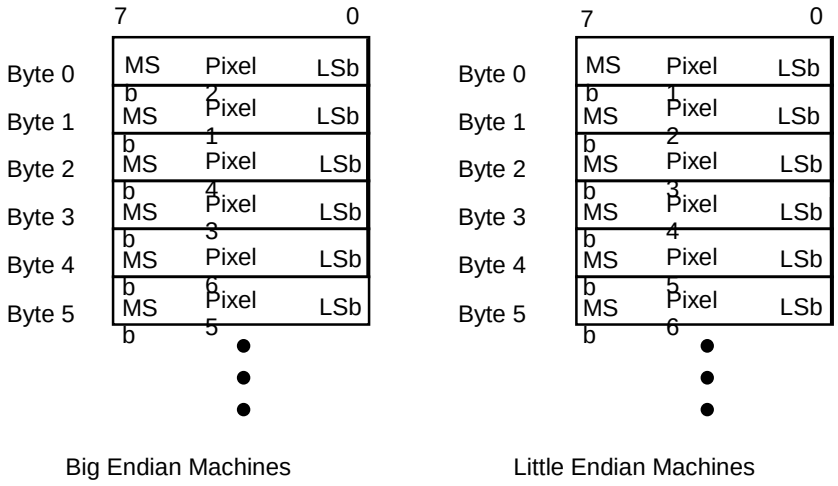

Figure 9: Sample Pixel Data Byte Streams for 8-bits Allocated, 8-bits Stored and a High bit of 7 (VR = OW)

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 toolkit 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

Merge DICOM 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 by 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()
MC_Get_Frame_To_Function()
```

Merge DICOM Toolkit will encode supplied data in an encapsulated format and generate the basic offset table. The data of basic offset table could be retrieved using call:

```
MC_Get_Offset_Table_To_Function()
```

Merge DICOM Toolkit 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 13*.

Table 13: Sample Encapsulated Pixel Data

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 04C6H	Compressed Fragment	(FFFE, E000)	0000 024AH	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

As specified by the DICOM standard, the various elements shown in *Table 13*, 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 10* contains the sample pixel data of *Table 13* in little endian format.

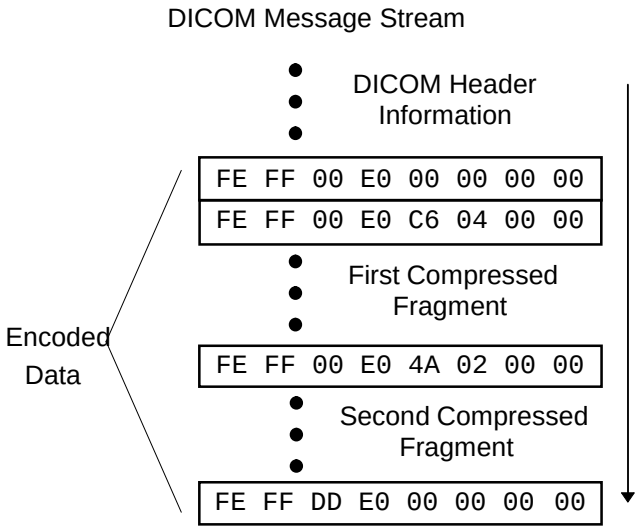

Figure 10: Sample Encoding of Encapsulated Pixel Data in Table 13

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 that `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 an 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 file. This can be done dynamically (at run-time) 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, Merge DICOM Toolkit supplies DICOM Toolkit 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 toolkit and will be discussed further in a later section.

`message.txt` can be very useful

Another file supplied with Merge DICOM Toolkit is the `message.txt` file. This file contains a listing of all the messages supported by the toolkit 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 Merge DICOM Toolkit 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_IR 144, ISO_IR 127, ISO_IR 126, ISO_IR 138, ISO_IR 148,
ISO_IR 166, ISO_IR 13, ISO 2022 IR 6, ISO 2022 IR 100,
ISO 2022 IR 101, ISO 2022 IR 109, ISO 2022 IR 110, ISO 2022 IR 144,
ISO 2022 IR 127, ISO 2022 IR 126, ISO 2022 IR 138, ISO 2022 IR 148,
ISO 2022 IR 149, ISO 2022 IR 166, ISO 2022 IR 13, ISO 2022 IR 87,
ISO 2022 IR 159, ISO_IR 192, GB18030
0008,1110    Referenced Study Sequence                     SQ     2
Item Name(s): REF_STUDY
0008,1125    Referenced Visit Sequence                     SQ     2
```

```
Item Name(s): REF_VISIT
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): CODE_SEQUENCE_MACRO
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    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 Merge DICOM Toolkit'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.
- 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).

and what validation cannot do for you.

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

- DICOM Part 3 specifies Information Object Definitions (IODs) as being composed of modules. Each module contains attributes. Only in the case of composite IODs may an attribute be specified in DICOM Part 3 as being contained in either a User Optional or Conditional Module. Merge DICOM Toolkit treats all such attributes as being value type 3 (optional).
- Also, only in the case of composite IODs (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. Merge DICOM Toolkit treats all Event Type IDs identically when performing message validation; namely it treats all attributes as type 3.

An example...

An example of the use of `MC_Validate_Message()` follows

```
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 example, 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 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 14*.

Table 14: Point of performance overhead associated with message validation

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()`

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 `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. Merge DICOM Toolkit 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 Merge DICOM Toolkit 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, Merge DICOM Toolkit 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.

Merge DICOM Toolkit also supplies function calls to the developer to generate and read DICOM message streams directly (see *Figure 11*). `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.

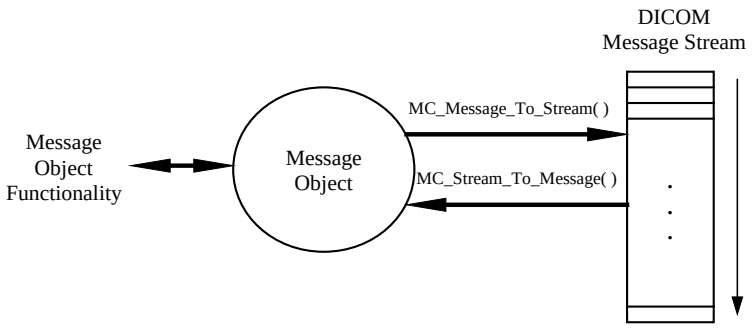

Figure 11: Relationship between a Message Object and a Message Stream

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 Merge DICOM Toolkit. 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 setting of `LARGE_DATA_STORE` 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 toolkit 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'll discuss how these DICOM messages are exchanged with other application entities over the network.

The exchange of DICOM messages between AEs 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 Merge DICOM Toolkit Function calls, are pictured in *Figure 12*.

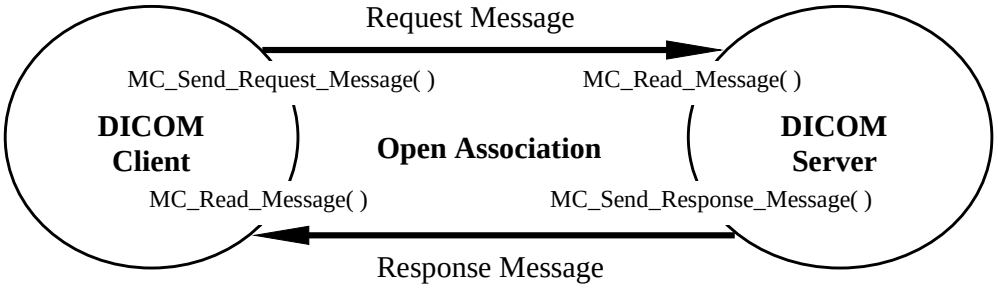

Figure 12: Message Exchange in Merge DICOM Toolkit Applications

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 `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 15*. Response codes for other DICOM commands are described in Part 4 of the DICOM Standard.

Table 15: Valid Response Message Status Codes for an N-GET Command

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

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 message's `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.

Message Exchange with Callbacks

Functions MC_Send_Request, MC_Send_Response and MC_Read_To_Stream provide the same functionality as their message analogs, but allow to utilize the *Callback* mechanism for message exchange. This might be especially useful for large data transfers, as the data are not loaded into application memory, but instead are sent or received by multiple blocks through the callback calls. In addition to decreasing of application memory footprint this mechanism avoids the internal message parsing, which leads to better performance.

Message exchange functions of the Merge DICOM Toolkit are listed in Table 16.

Table 16: Merge DICOM Toolkit message exchange functions

Function
MC_Send_Request_Message()
MC_Send_Request()
MC_Send_Response_Message()
MC_Send_Response()
MC_Read_Message()
MC_Read_Message_To_Tag()
MC_Continue_Read_Message_To_Tag()
MC_Continue_Read_Message_To_Stream()
MC_Read_To_Stream()

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 Merge DICOM Toolkit'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 Merge DICOM Toolkit) 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 13* illustrates how an SCU waits on a response from the SCP while the SCP processes a message.

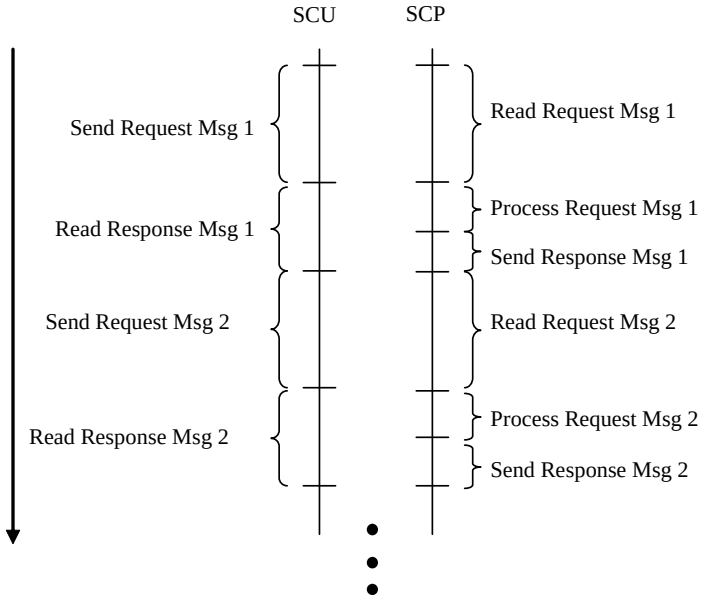

Figure 13: Timing of Message Exchange During Synchronous Transfers

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 Merge DICOM Toolkit 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 14 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.

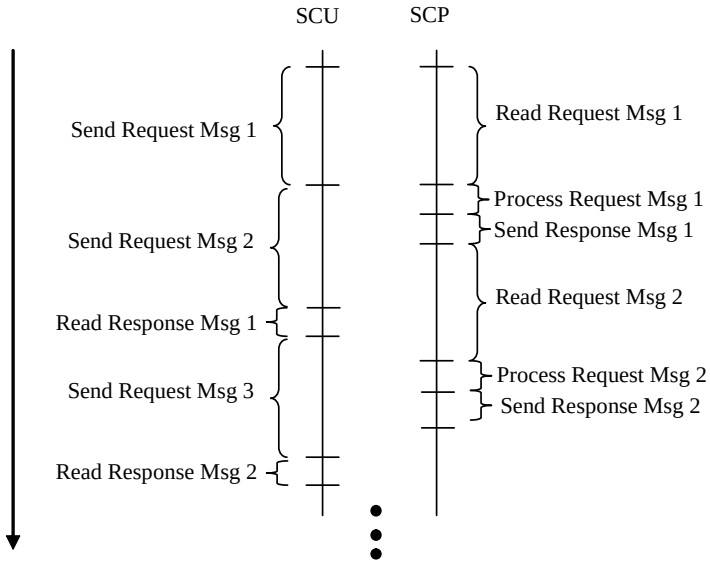

Figure 14: Timing of Message Exchange During Asynchronous Transfers

Message ID must be set for asvnchronous

Asynchronous operations are implemented utilizing the standard Merge DICOM Toolkit network functions, listed in Table 16. 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. Merge DICOM Toolkit 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, Merge DICOM Toolkit 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. Merge DICOM Toolkit 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

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 Merge DICOM Toolkit application must start reading the data. These options allow data to be queued by the OS in the background while a Merge DICOM Toolkit 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 call to `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 that need to be stored or transmitted across a network.

The callbacks, when registered, are utilized any time the functions in *Table 17* 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` or `RLE`.

Table 17: Callbacks Utilized by Functions that set and get Pixel Data

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
`MC_Get_Frame_To_Function()`	Decompressor
`MC_Get_Offset_Table_To_Function()`	Decompressor

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 one of the built in compressors and decompressors supplied by Merge DICOM Toolkit.

How to register a Compression Callback Function

`MC_Register_Compression_Callbacks()` can be used to register a Callback Function with the toolkit as follows:

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

This call registers `MyCompressionCallback()` and `MyDecompressionCallback()` with the DICOM Toolkit 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` allows 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. Merge DICOM Toolkit 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.

Merge DICOM Toolkit 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). Merge DICOM Toolkit 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

Merge DICOM Toolkit has two sets of built in compressors and decompressors. The `MC_RLE_Compressor` and `MC_RLE_Decompressor` can be used to compress or

decompress data encoded in the RLE transfer syntax. These routines support data with photometric interpretation of MONOCHROME1, MONOCHROME2, RGB, and YBR_FULL. The routines are supported on all Merge DICOM Toolkit platforms. These callbacks should be registered as follows:

```
status = MC_Register_Compression_Callbacks(msgID,
       MC_RLE_Compressor, MC_RLE_Decompressor);
```

The other Merge DICOM Toolkit built in compressor is MC_Standard_Compressor and the built in decompressor is MC_Standard_Decompressor. These routines utilize libraries from Accusoft (formerly Pegasus Imaging Corporation) (www.accusoft.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. These compressors are only available on platforms that Pegasus supports. These are 32/64-bit Windows on x86, 32-bit Solaris on SPARC, 32/64-bit Linux on x86 and 32-bit Android on ARM7.

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

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

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 Accusoft and configured in Merge DICOM Toolkit 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 18* 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 before decompressing YBR_FULL or YBR_FULL_422 data.

Table 18: JPEG Baseline Supported Photometric Interpretations and Bit Depths

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	3

NOTE: As of the present release of the toolkit, only the `JPEG_BASELINE` decompression is supported on the Android platform. The Pegasus library for `JPEG_BASELINE` compression (6420) is not available on the Android platform.

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 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` before decompressing `YBR_FULL` or `YBR_FULL_422` data.

Table 19: JPEG Extended Supported Photometric Interpretations and Bit Depths

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	3

NOTE: As of the present release of the toolkit, only the JPEG_EXTENDED_2_4 decompression is supported on the Android platform. The Pegasus library for JPEG_EXTENDED_2_4 compression (6420) is not available on the Android platform.

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 20* 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. Table 20: JPEG Lossless Supported Photometric Interpretations and Bit Depths

JPEG Lossless Non-Hierarchical Process 14		
Photometric Interpretation	MONOCHROME1 MONOCHROME2	RGB
Bits Stored	2 to 16	8
Bits Allocated	8 or 16	8
Samples Per Pixel	1	3

NOTE: As of the present release of the toolkit, only the JPEG_LOSSLESS_HIER_14 decompression is supported on the Android platform. The Pegasus library for JPEG_LOSSLESS_HIER_14 compression (6220) is not available on the Android platform.

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 21* details the photometric interpretation and bit depths supported by the standard compressor and decompressor for this transfer syntax.

Table 21: JPEG 2000 Lossy Supported Photometric Interpretations and Bit Depths

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

NOTE: As of the present release of the toolkit, only the JPEG_2000 decompression is supported on the Android platform. The Pegasus library for JPEG_2000 compression (6820) is not available on the Android platform.

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 22* details the photometric interpretation and bit depths supported by the standard compressor and decompressor for this transfer syntax.

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

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

NOTE: As of the present release of the toolkit, only the `JPEG_2000_LOSSLESS_ONLY` decompression is supported on the Android platform. The Pegasus library for `JPEG_2000_LOSSLESS_ONLY` compression (6820) is not available on the Android platform.

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` are used 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.

After using the standard compressor or decompressor the application should call `MC_Thread_Release()` to free resources allocated by Pegasus, before the thread that used the compressor or decompressor ends. This is needed to avoid memory leaks on Linux and UNIX platforms.

Merge DICOM Toolkit can update group 0x0028 for you

The `UPDATE_GROUP_0028_ON_DUPLICATE` configuration option can also be enabled so Merge DICOM Toolkit will update the Group 0x0028 tags for you. When this configuration option is enabled, the Photometric Interpretation will be updated for you as mentioned above. When decompressing an image, the photometric interpretation will also be updated. In addition, when lossy compression is done, the Lossy Image Compression, Lossy Image Compression Ratio, and Lossy Image Compression Method tags will be updated by Merge DICOM Toolkit.

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_Function()` and `MC_Set_Value_From_Function()` functions and the 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 Merge DICOM Toolkit 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 Merge DICOM Toolkit that pixel should be included in the message or file, Merge DICOM Toolkit 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 toolkit as follows:

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

This call registers `MyCallbackFunction()` with the DICOM Toolkit 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 transmission and reception of the data associated with the tag(s).

`MyCallbackFunction()` is prototyped as follows:

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 attribute value or supply/receive a 'chunk' of value data. `CBdataSizePtr`, `CBdataBufferPtr`, `CBisFirstPtr` and `CBisLastPtr` are used to manage the data flow. *Table 23* describes this behavior.

Table 23: Callback Function Behavior Based on Value of CBtype parameter

Value of CBtype	Required Callback Behavior
`REQUEST_FOR_DATA_LENGTH`	Callback is supplying OB/OW/OF data and the length of the entire attribute 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. Merge DICOM Toolkit 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).

Value of CBtype	Required Callback Behavior
`REQUEST_FOR_DATA_WITH_ OFFSET`	Callback is supplying OB/OW data and a 'chunk' of this data from a specific offset in the seekable stream is being requested. *CBdataSizePtr is the pointer to long which contains the offset of required data. After reading the data, the 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. Merge DICOM Toolkit 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 an 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. Merge DICOM Toolkit has set *CBdataSizePtr to the number of bytes of data it is providing at *CBdataBufferPtr. Merge DICOM Toolkit will set CBisFirstPtr 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

Value of CBtype	Required Callback Behavior
PROVIDING_DATA	Callback is receiving OB/OW/OF data and a 'chunk' of this data is being supplied. Merge DICOM Toolkit 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. Merge DICOM Toolkit 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). Merge DICOM Toolkit sets *CBisLastPtr to TRUE (not zero) the last time it will be providing data for this attribute's value (i.e., when it is providing the last block of data).
FREE_DATA	The memory associated with the enclosing message, file, or item is being freed, and the callback can free its memory associated with the OB/OW/OF data. Note that callbacks are only called with this callback type if the USE_FREE_DATA_CALLBACK configuration option is enabled in the system profile.

Because a single Callback Function's behavior and usage of parameters must vary based on the value of `CBtype` this may all seem a bit 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 containing 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 15 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 files. An attribute whose value is a sequence of items is simply an attribute that has a potentially large and complex value. Fortunately, Merge DICOM Toolkit 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.

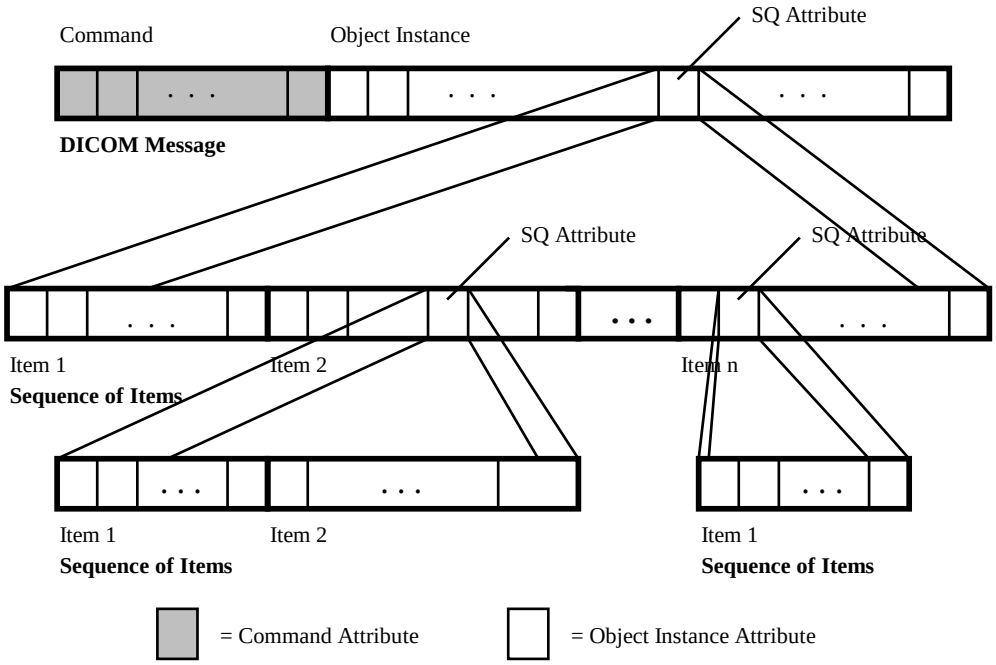

Figure 15: A DICOM Message containing doubly nested sequences of items.

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 Merge DICOM Toolkit 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.

`ItemNames` are strings used to identify each item and are listed in the `message.txt` file for attributes 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 Issuer of Accession Number Item, and the other the contents of that item.

```
######################################################################
CHEST_CAD_SR - C_STORE_RQ
######################################################################

0008,0005    Specific Character Set                          CS    1C
Condition: EXTENDED_OR_REPLACEMENT_CHARACTER_SET_USED
Defined Terms:     ISO_IR 100, ISO_IR 101, ISO_IR 109, ISO_IR 110,
ISO_IR 144, ISO_IR 127, ISO_IR 126, ISO_IR 138, ISO_IR 148,
ISO_IR 166, ISO_IR 13, ISO 2022 IR 6, ISO 2022 IR 100,
ISO 2022 IR 101, ISO 2022 IR 109, ISO 2022 IR 110, ISO 2022 IR 144,
ISO 2022 IR 127, ISO 2022 IR 126, ISO 2022 IR 138, ISO 2022 IR 148,
ISO 2022 IR 149, ISO 2022 IR 166, ISO 2022 IR 13, ISO 2022 IR 87,
ISO 2022 IR 159, ISO_IR 192, GB18030
0008,0012    Instance Creation Date                          DA    3
0008,0013    Instance Creation Time                          TM    3
0008,0014    Instance Creator UID                            UI    3
0008,0015    Instance Coercion DateTime                      DT    3
0008,0016    SOP Class UID                                   UI    1
0008,0018    SOP Instance UID                                UI    1
0008,001A    Related General SOP Class UID                   UI    3
0008,001B    Original Specialized SOP Class UID              UI    3
0008,0020    Study Date                                      DA    2
0008,0021    Series Date                                     DA    3
0008,0023    Content Date                                    DA    1
0008,0030    Study Time                                      TM    2
0008,0031    Series Time                                     TM    3
0008,0033    Content Time                                    TM    1
0008,0050    Accession Number                                SH    2
0008,0051    Issuer of Accession Number Sequence             SQ    3
Item Name(s): ISSUER_OF_ACCESSION_NUMBER
                                .
                                .
                                .

======================================================================
Item Name: ISSUER_OF_ACCESSION_NUMBER
======================================================================

0040,0031    Local Namespace Entity ID                       UT    1C
Condition: A00400032_NOT_PRESENT
0040,0032    Universal Entity ID                             UT    1C
Condition: A00400031_NOT_PRESENT
0040,0033    Universal Entity ID Type                        CS    1C
Condition: A00400032_PRESENT
Defined Terms:     DNS,EUI64,ISO,URI,UUID,X400,X500
```

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 Merge DICOM Toolkit. See that application's code for further examples 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 Merge DICOM Toolkit 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 16 summarizes the DICOM file access function calls described in this section.

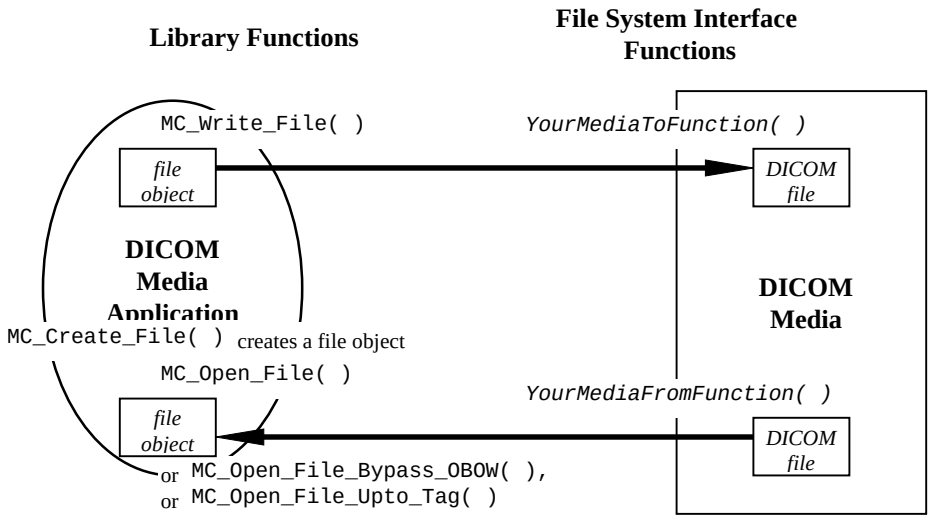

Figure 16: The DICOM File Access Functions in Merge DICOM Toolkit

File System Interface Functions

This may sound strange, but all the media interchange functionality of the DICOM 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 DICOM Toolkit 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 DICOM Toolkit 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 an `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 DICOM Toolkit 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 Toolkit 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 call back `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

Variants on `MC_Open_File()` are also provided by the toolkit:

- `MC_Open_File_Bypass_OBOW()` will read in an attribute's value that is of type OB or OW, but will not store it, 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 it in to improve the memory consumption or process it directly from the file using your own special filters or hardware.

- `MC_Open_File_Upto_Tag()` will stop reading the attributes from media into the file object when it reaches the first attribute greater than a specified tag. The offset in bytes from the beginning of the file to the beginning of the first attribute 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 using callback mechanism, or process it directly from the file using your own special filters or hardware.

- `MC_Open_File_Upto_Tag_Bypass_Value()` will read the attributes from the media into the file object including a specified tag, but will not read the tag attributes value. This function is most useful when the DICOM file contains large pixel data (7FE0, 0010) attribute. In this case the most effective way to handle it is to use the callback mechanism through MC_Register_Callback_Function so the attribute values could be retrieved from media upon later request. The user's application callback must then deal with reading data from seekable DICOM file stream from the given offset provided in the REQUEST_FOR_DATA_WITH_OFFSET command.

See the Reference Manual for a detailed description of the use of these 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 *`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()` works 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 a specialized application to examine and modify the 128 byte DICOM file preamble. The DICOM 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 24*.

Table 24: Point of performance overhead associated with file validation

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.

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 `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 `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 Merge DICOM Toolkit 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 `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.

Saving Raw (Unparsed) Messages as DICOM Files

Performance Tuning

A common usage of the *Merge DICOM Toolkit* is to save incoming (received from network) messages. When reading a DICOM message from the network, attributes in a message are parsed, validated before storing them in memory, and then later written out from memory objects to a DICOM file. With a message that has many levels of nested items, the parsing/creating of DICOM attributes in memory has a significant impact in performance. The Storage SCP application is commonly used to write out the received message content to a DICOM file without the need to modify the attributes of the message. When such a case is needed, it is best to save the raw streamed content as quickly and efficiently as possible. The following code snippet shows how to save an incoming message into a DICOM file without parsing: (For detail implementation, please refer to mc3apps\stor_scp.c in the distribution folder.)

```
/*
 * To read message from the association and save the raw content
  * without parsing the message's dataset, use
 * MC_Read_Message_To_Tag() to read only the "group 0" part of
 * the message instead of using MC_Read_Message() to read the
 * entire message content.
 */
mcStatus = MC_Read_Message_To_Tag( assocID, 30,
       0x00010000,  /*(0001,000) tag is just after group 0 */
       &msgID, &serviceName, &command);

/*
 * Get received message's transfer syntax
 */
mcStatus = MC_Get_Message_Transfer_Syntax( msgID,
       &messageSyntax );
/*
 * Convert transfer syntax enum to UID
 */
mcStatus = MC_Get_Transfer_Syntax_From_Enum(messageSyntax,
       syntaxUID, sizeof(syntaxUID));
/*
 * Set transfer syntax UID in (0002,0010) using same transfer
 * syntax from received message
 */
mcStatus = MC_Set_Value_From_String( msgID,
       MC_ATT_TRANSFER_SYNTAX_UID, syntaxUID);
/*
 * Set the rest of group 2 attributes in the message.
 * See AddGroup2ElementsFromGroup0() implementation in
 * mc3apps\stor_scp.c.
 */
sampStatus = AddGroup2ElementsFromGroup0( options, msgID );

/*
 * Convert the message to a file object with a filename ready
 * to be written out.
 */
mcStatus = MC_Message_To_File( msgID, filename );

/*
 * Stream out the DICOM part 10 file meta header (group 2) using
 * MC_Message_To_Stream().
 * See RawObjToFile() implementation in mc3apps\stor_scp.c.
 */
```

```
mcStatus = MC_Message_To_Stream ( msgID, 0x00020000, 0x0002FFFF,
       EXPLICIT_LITTLE_ENDIAN, &callbackInfo, RawObjToFile );

/*
 * Continue to read the data set part of message from network
 * and write to file directly using
 * MC_Continue_Read_Message_To_Stream().
 */
mcStatus = MC_Continue_Read_Message_To_Stream( assocID, msgID,
   &callbackInfo, RawObjToFile);

/*
 * Free the message after completion
 */
mcStatus = MC_Free_File(&msgID);
```

Note: Due to the raw saving technique, non DICOM compliant messages will be saved as is and no warning will be issued (due to no parsing of message).

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 usually 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 they exist) referenced by a directory record. DICOM File IDs 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 Merge DICOM Toolkit supplies functions that make DICOMDIR maintenance much simpler for your application.

The toolkit includes a basic, in memory, DICOMDIR API (`MC_Dir_` functions) and an enhanced, on-demand/incremental DICOMDIR API (`MC_DDH_` functions). The enhanced functions are now described.

Opening and Navigation

Opening a DICOMDIR file is simplified compared to other DICOM files because of the specifics of DICOMDIR. Opening an existing DICOMDIR file can be done using the `MC_DDH_Open()` function by specifying the location of the file on disk. This function reads in the DICOMDIR file attributes up to the record sequence attribute. Attributes for each record are read on demand, whenever the application requests a record that was not read before or a record that was released from memory.

A new DICOMDIR file can be created using the `MC_DDH_Create()` function, specifying the location of the file, an optional File Set ID and a template file containing group 2 and 4 attributes for the new DICOMDIR.

The identifier returned by either of these functions can be used with most value access function (e.g. `MC_Get_Value` functions) with certain limitations as described in the description of these functions.

When the application is done with the DICOMDIR object it must release the object as a regular file, using `MC_Free_File()`.

Navigating through a DICOMDIR

Once open, navigating a DICOMDIR usually involves calling `MC_DDH_Get_First_Lower_Record()` to get the first record of the root entity (e.g. the first patient record). From here `MC_DDH_Get_Next_Record()` is used to sequentially obtain further records of the root entity; while `MC_DDH_Get_First_Lower_Record()` is used to get lower level entity records (e.g. study records). `MC_DDH_Get_Parent_Record()` can be used to navigate up the record hierarchy.

The record identifiers obtained using the record navigation functions can be used with the `MC_Get_Value()` functions to obtain record attribute values. These identifiers are valid until the associated record object is freed. Record objects are

freed automatically when the DICOMDIR object is freed, or manually when the application calls `MC_DDH_Release_Record()` or `MC_DDH_Delete_Record()`.

Modifications to the content of existing records are not allowed.

The toolkit provides an easy and fast way for searching and collecting data from records through the `MC_DDH_Traverse_Records()` function. This function can be used for the traversal of the entire record hierarchy or just a branch of records by setting the proper starting point through the *RootID* argument. The toolkit will call the provided callback for each record encountered during traversal. The callback can dynamically control the traversal using various return values.

Here is an example on how to use `MC_DDH_Traverse_Records()` to find a specific series record:

```
/* Find Information Structure */
typedef struct RecordFind_Struct
{
   char*           matchValue;
   int             matchRecordID;
   char            buff[256];
} RecordFind;

/* Find callback */
MC_TRAVERSAL_STATUS FindSeriesCbk(int CrtRecID, void* CbkData)
{
   MC_STATUS status;
   MC_DIR_RECORD_TYPE recType = MC_REC_TYPE_UNKNOWN;
   RecordFind* findInfo = (RecordFind*)CbkData;

   /* Check for record type */
   MC_DDH_Get_Record_Type(CrtRecID, &recType);
   if(recType != MC_REC_TYPE_SERIES) return MC_TS_CONTINUE;

   /* Check for matching Series Instance UID */
   status = MC_Get_Value_To_String(CrtRecID,
      MC_ATT_SERIES_INSTANCE_UID, 256, findInfo->buff);

   if(status == MC_NORMAL_COMPLETION)
   {
      if(strcmp(findInfo->matchValue, findInfo->buff) == 0)
      {
          /* Found a match */
          findInfo->matchRecordID = CrtRecID;
          return MC_TS_STOP;
      }
   }

   /* Continue the traversal with the next series record instead
      of lower level instance records */
   return MC_TS_STOP_LOWER;
}

void FindSeries()
{
   RecordFind findInfo;

   findInfo.matchValue = "1.2.3.4.5.6.7.8.9";
```

```
    findInfo.matchRecordID = 0;

    MC_DDH_Traverse_Records(dirID, &findInfo, FindSeriesCbk);
    …
}
```

Adding and Deleting Records

The addition or deletion of directory records are handled through two simple calls: `MC_DDH_Add_Record()` and `MC_DDH_Delete_Record()`. These calls modify only the toolkit's view on the DICOMDIR record hierarchy; to apply the changes to the DICOMDIR file the application must call `MC_DDH_Update()`.

When adding a record using `MC_DDH_Add_Record()`, you pass in the identifier of the parent record or directory object (if adding a top level record) and optionally the record type to add. In most of the cases the new record's type can be inferred from the type of the parent record and you can pass a NULL for the record type. For instance level records the record type is determined by the SOP Class of the referenced instance and the toolkit accepts the SOP Class UID as the value of the record type.

If successful, `MC_DDH_Add_Record()` returns in the last parameter the identifier of the new toolkit record which can be used with `MC_Set_Value...` methods to set attribute values. An easier way to add attributes to the new record is using `MC_DDH_Copy_Values()` which can copy a specific set of attribute values from a message or file object into the new record.

Automatic deletion of referenced items

When deleting a record using `MC_DDH_Delete_Record()`, the only parameter required is the identifier of the record to delete. When a directory record is deleted, all lower level directory records are also deleted and freed and the associated identifiers become invalid.

Make sure you are committed!!

The Merge DICOM Toolkit updates and maintains all byte offsets that are part of the DICOMDIR structure automatically. But, one important note: All the changes to a DICOMDIR are made in memory and are not committed to the media until an `MC_DDH_Update()` call is made.

Storage of Directory Records

Depending on the type of media being used, the size of objects being stored, and how many tags are stored, the size of a DICOMDIR can grow quite large. For this reason, the toolkit delays the reading of the directory records until the application requests them. Once a directory record is read into memory the application can call `MC_DDH_Release_Record()` to release the memory associated with the record and all lower level records. The toolkit will reload any released record from the DICOMDIR file when required.

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:

- standard attributes in the Merge DICOM Toolkit API are referenced by `Tag`, while private attributes are referred to by `PrivateCode`, `Group`, and `ElementByte`.
- The `Group` number of a private attribute must always be odd, while for a standard attribute it is always even.
- 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 256 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.

Multi-threading Support

Check platform notes for multi-threading support

The Merge DICOM Toolkit library has been designed to be thread safe. Note, however this thread safety is not enabled on all platforms. Please check the platform notes to be sure that Merge DICOM Toolkit is thread safe on your platform.

There are some assumptions, however, concerning the thread safety of Merge DICOM Toolkit. In most cases, it is assumed that a Merge DICOM Toolkit object is only accessed from one thread at a time. This applies to Message objects, file objects, item objects, and association objects. Note, however, that different instances of an object can always be manipulated in different threads at the same time.

There are exceptions to this rule, however. The following is a summary of them:

The `MC_Abort_Association()` function call can be called when Merge DICOM Toolkit is working on an association in another thread. The `MC_Send_Request_Message`, `MC_Send_Request`, `MC_Send_Response_Message`, `MC_Send_Response`, `MC_Read_Message`, `MC_Read_To_Stream`, `MC_Close_Association` and `MC_Abort_Association` functions can all be used in one thread while another thread is calling `MC_Abort_Association` within another thread. This is useful when allowing a user to asynchronously cancel an in-progress association.

It is also possible to access tags within a message as it is being read from the network. It is possible to call the `MC_Set_Value_...`, `MC_Set_pValue_...`, `MC_Get_Value_...` and `MC_Get_pValue_...` routines from one thread while another thread is calling the `MC_Read_Message_To_Tag`, `MC_Continue_Read_Message`, `MC_Continue_Read_Message_to_Stream` routines, or when these routines are used in conjunction with a callback function registered with `MC_Register_Callback_Function` to manage pixel data.

Memory Management

Performance Tuning

Merge DICOM Toolkit 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 the toolkit. 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 the toolkit. They also allow Merge DICOM Toolkit 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. Merge DICOM Toolkit uses these routines to allocate buffers in sizes between 4 bytes and 28K. When an allocation is requested, the toolkit will take the

smallest buffer that will fit the bytes requested. These buffers will be kept in the toolkit's internal memory pool and never freed. For allocations larger than 28K, Merge DICOM Toolkit will simply use the 'C' functions `malloc()` and `free()`. Under most conditions, Merge DICOM Toolkit 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 Merge DICOM Toolkit typically 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 Merge DICOM Toolkit should stabilize and not increase in size. In applications using Merge DICOM Toolkit from multiple threads, this memory usage is not as consistent and depends on the timing of the threads using the toolkit. As a result of these routines, the first time an application performs a DICOM operation is typically slower than subsequent operations.

Merge DICOM Toolkit supplies the `MC_Report_Memory()` and `MC_Cleanup_Memory()` routines to allow some user control over this memory management. `MC_Report_Memory()` reports how much memory is currently allocated and in use by the toolkit. The `MC_Cleanup_Memory()` routine can be used to actually free memory allocated by Merge DICOM Toolkit. The routine looks for blocks of memory that are no longer in use by the toolkit and frees them with the operating system. This can be useful when a Merge DICOM Toolkit application reads a large DICOM object (such as a large DICOMDIR or a large multi-frame image) and the user would like to free some of the memory associated with the object.

When developing a DICOM application with Merge DICOM Toolkit, the most memory intensive operation is dealing with image data. The following sections discuss various Merge DICOM Toolkit functions. A description is given of how these functions manage memory in conjunction with various toolkit 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 Merge DICOM Toolkit 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, Merge DICOM Toolkit'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 Merge DICOM Toolkit 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 the buffer passed to `MC_Set_Value_From_Function()` does not have an effect on memory usage.

Reading Messages from the Network

Merge DICOM Toolkit 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 Merge DICOM Toolkit's standard memory management routines, the method for storing the image data can be influenced.

MC_Read_Message

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, Merge DICOM Toolkit will store the buffers in memory allocated with the 'C' function `malloc`(). This can be used to reduce the toolkit'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`, `MC_Open_File_Upto_Tag` and `MC_Open_File_Upto_Tag_Bypass_Value()` 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 of the 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()`, `MC_Open_File_Upto_Tag()` or `MC_Open_File_Upto_Tag_Bypass_Value()` 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

Merge DICOM Toolkit 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, OD or OF. When encountered, the attribute's data is passed to the registered callback function instead of being stored within Merge DICOM Toolkit. 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 Merge DICOM Toolkit'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 Merge DICOM Toolkit's Memory Management Functions For Pixel Data

When using `MC_Register_Callback_Function` to replace Merge DICOM Toolkit'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, Merge DICOM Toolkit 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 Merge DICOM Toolkit in any length as required by the user's application. The data is typically read from disk at this point and directly passed to Merge DICOM Toolkit. 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 Merge DICOM Toolkit. The complete image file can be reassembled later using Merge DICOM Toolkit.

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 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 be 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.

DICOM Structured Reporting

The Merge DICOM Toolkit provides high-level functionality to handle DICOM Structured Report (SR) Documents. This functionality provides a simple way for encoding and decoding SR Document content by manipulating content items and their attributes instead of tags and values.

Structured Report Structure and Modules

The DICOM standard Part 3 defines the following generic types of SR Information Object Definitions (IODs):

- Basic Text SR Information Object Definition — The Basic Text Structured Report (SR) IOD is intended for the representation of reports with minimal usage of coded entries (typically used in Document Title and headings) and a hierarchical tree of headings under which may appear text and subheadings. Reference to SOP Instances (e.g. images or waveforms or other SR Documents) is restricted to appear at the level of the leaves of this primarily textual tree. This structure simplifies the encoding of conventional textual reports as SR Documents, as well as their rendering.

- Enhanced SR Information Object Definition — The Enhanced Structured Report (SR) IOD is a superset of the Basic Text SR IOD. It is also intended for the representation of reports with minimal usage of coded entries (typically Document Title and headings) and a hierarchical tree of headings under which may appear text and subheadings. In addition, it supports the use of numeric measurements with coded measurement names and units. Reference to SOP Instances (e.g. images or waveforms or SR Documents) is restricted to appear at the level of the leaves of this primarily textual tree. It enhances references to SOP Instances with spatial regions of interest (points, lines, circle, ellipse, etc.) and temporal regions of interest.

- Comprehensive SR Information Object Definition — The Comprehensive SR IOD is a superset of the Basic Text SR IOD and the Enhanced SR IOD, which specifies a class of documents, the content of which may include textual and a variety of coded information, numeric measurement values, references to the SOP Instances and spatial or temporal regions of interest within such SOP Instances. Relationships by-reference are enabled between Content Items.

There are more specific SR IODs defined in the DICOM, like ***Key Object Selection Document*** and ***Mammography CAD SR***. Those IODs use the same way to encode data and the difference is in the constrains on the Content Item Types and their relationships. *Figure 17* illustrates the typical SR Document structure. As you can see, the top level header is very similar to the DICOM image IODs and consists of the same Patient, Study and Series modules. The main difference from other IODs is the ***SR Document Content Module***. The attributes in this Module convey the content of an SR Document.

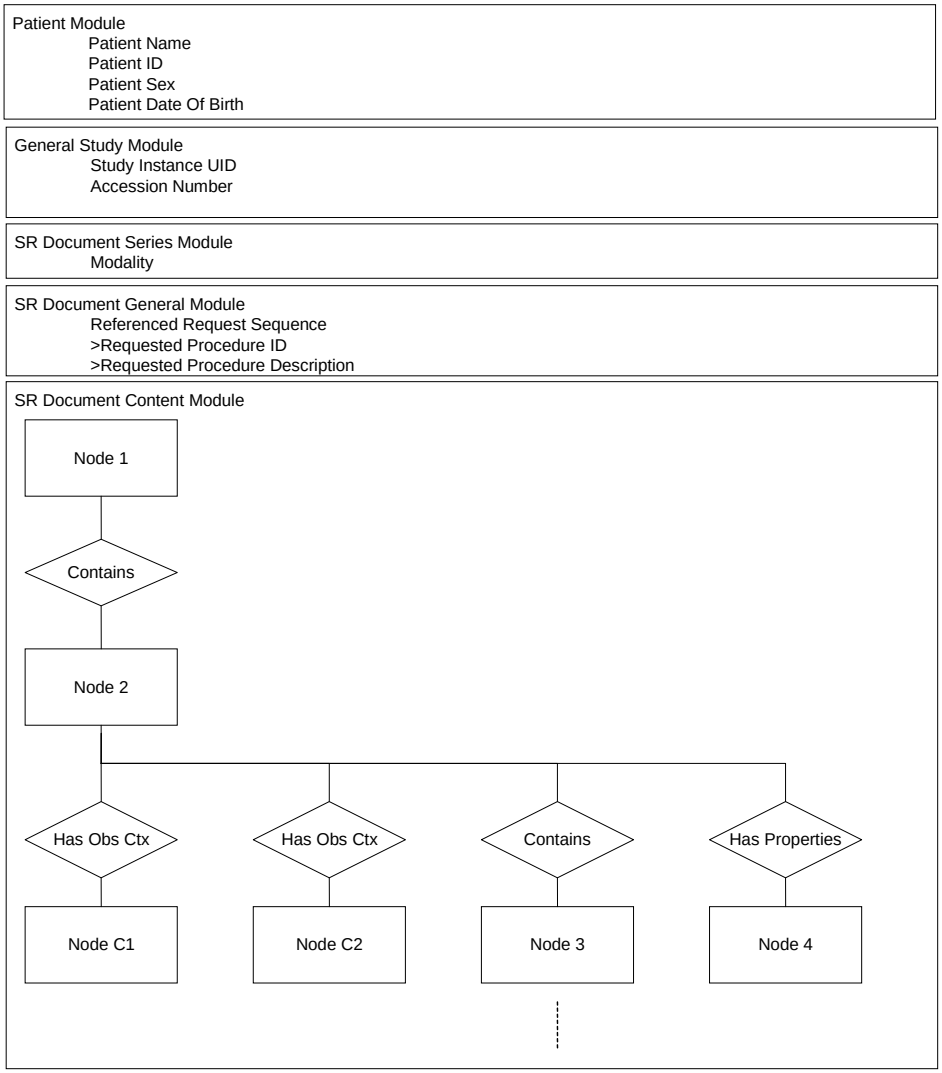

Figure 17: SR Document Structure example

The SR Document hierarchy

The Document Content Module has a tree structure and consists of a single root Content Item (Node 1) that is the root of the SR Document tree. The root Content Item conveys either directly or indirectly all of the other nested Content Items in the document. The hierarchical structuring of the Content Tree provided by recursively nesting Content Items. A parent (or source) Content Item has an explicit relationship to each child (or target) Content Item, conveyed by the Relationship Type. *Figure 18* depicts the relationship of SR Documents to Content Items and the relationships of Content Items to other Content Items and to Observation Context.

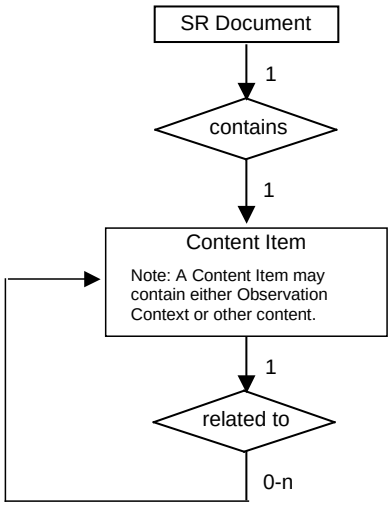

Figure 18: SR Information Model

Each Content Item contains the following:

- A name/value pair, consisting of:

o a single Concept Name Code that is the name of a name/value pair or a heading; and
o a value (text, numeric, code, etc.).

- References to images, waveforms or other composite objects, with or without coordinates.
- Relationships to other Items, either by-value through nested Content Sequences, or by-reference.

Note: Some Content Item Types can have multiple values.

Content Item Types

Table 25 defines all possible Content Item Types that can be used in the SR Document Content Module. The choice of which may be constrained by the IOD in which this Module is contained. Merge DICOM Toolkit Definition column specifies the enumerated value used in the Toolkit to identify the Content Item Type.

Table 25: SR Content Item Types

Item Type	Merge DICOM Toolkit Definition	Concept Name	Description
TEXT	SR_NODE_TEXT	Type of text, e.g. "Findings", or name of identifier, e.g. "Lesion ID"	Free text, narrative description of unlimited length. May also be used to provide a label or identifier value.
NUM	SR_NODE_NUM	Type of numeric value or measurement, e.g. "BPD"	Numeric value fully qualified by coded representation of the measurement name and unit of measurement.
CODE	SR_NODE_CODE	Type of code, e.g. "Findings"	Categorical coded value. Representation of nominal or non-numeric ordinal values.
DATETIME	SR_NODE_DATETIME	Type of DateTime, e.g. "Date/Time of onset"	Date and time of occurrence of the type of event denoted by the Concept Name.
DATE	SR_NODE_DATE	Type of Date, e.g. "Birth Date"	Date of occurrence of the type of event denoted by the Concept Name.
TIME	SR_NODE_TIME	Type of Time, e.g. "Start Time"	Time of occurrence of the type of event denoted by the Concept Name.
UIDREF	SR_NODE_UIDREF	Type of UID, e.g. "Study Instance UID"	Unique Identifier (UID) of the entity identified by the Concept Name.
PNAME	SR_NODE_PNAME	Role of person, e.g., "Recording Observer"	Person name of the person whose role is described by the Concept Name.
COMPOSITE	SR_NODE_COMPOSITE	Purpose of Reference	A reference to one Composite SOP Instance which is not an Image or Waveform.
IMAGE	SR_NODE_IMAGE	Purpose of Reference	A reference to one Image. IMAGE Content Item may convey a reference to a Softcopy Presentation State associated with the Image.
WAVEFORM	SR_NODE_WAVEFORM	Purpose of Reference	A reference to one Waveform.

Item Type	Merge DICOM Toolkit Definition	Concept Name	Description
SCOORD	SR_NODE_SCOORD	Purpose of Reference	Spatial coordinates of a geometric region of interest in the DICOM image coordinate system. The IMAGE Content Item from which spatial coordinates are selected is denoted by a SELECTED FROM relationship.
TCOORD	SR_NODE_TCOORD	Purpose of Reference	Temporal Coordinates (i.e. time or event based coordinates) of a region of interest in the DICOM waveform coordinate system. The WAVEFORM or IMAGE or SCOORD Content Item from which Temporal Coordinates are selected is denoted by a SELECTED FROM relationship.
CONTAINER	SR_NODE_CONTAINER	Document Title or document section heading. Concept Name conveys the Document Title (if the CONTAINER is the Document Root Content Item) or the category of observation.	CONTAINER groups Content Items and defines the heading or category of observation that applies to that content. The heading describes the content of the CONTAINER Content Item and may map to a document section heading in a printed or displayed document.

Relationship Types between Content Items

Table 26 describes the Relationship Types between Source Content Items and the Target Content Items. The choice of which may be constrained by the IOD in which this Module is contained. Merge DICOM Toolkit Definition column specifies the enumerated value used in the Toolkit to identify the Content Item Relationship.

Table 26: SR Relationship Types

Relationship Type	Merge DICOM Toolkit Definition	Description
CONTAINS	SR_REL_CONTAINS	Source Item contains Target Content Item. E.g.: CONTAINER "History" {CONTAINS: TEXT: "mother had breast cancer"; CONTAINS IMAGE 36}
HAS OBS CONTEXT	SR_REL_HAS_OBS_CONTEXT	Has Observation Context. Target Content Items shall convey any specialization of Observation Context needed for unambiguous documentation of the Source Content Item. E.g.: CONTAINER: "Report" {HAS OBS CONTEXT: PNAME: "Recording Observer" = "Smith^John^^Dr^"}
HAS CONCEPT MOD	SR_REL_HAS_CONCEPT_MOD	Has Concept Modifier. Used to qualify or describe the Concept Name of the Source Content item, such as to create a post-coordinated description of a concept, or to further describe a concept. E.g. CODE "Chest X-Ray" {HAS CONCEPT MOD: CODE "View = PA and Lateral"} E.g. CODE "Breast" {HAS CONCEPT MOD: TEXT "French Translation" = "Sein"} E.g. CODE "2VCXRPALAT" {HAS CONCEPT MOD: TEXT "Further Explanation" = "Chest X-Ray, Two Views, Posteroanterior and Lateral"}
HAS PROPERTIES	SR_REL_HAS_PROPERTIES	Description of properties of the Source Content Item. E.g.: CODE "Mass" {HAS PROPERTIES: CODE "anatomic location", HAS PROPERTIES: CODE "diameter", HAS PROPERTIES: CODE "margin", ...}.

Relationship Type	Merge DICOM Toolkit Definition	Description
HAS ACQ CONTEXT	SR_REL_HAS_ACQ_CONTEXT	Has Acquisition Context. The Target Content Item describes the conditions present during data acquisition of the Source Content Item. E.g.: IMAGE 36 {HAS ACQ CONTEXT: CODE "contrast agent", HAS ACQ CONTEXT: CODE "position of imaging subject", ...}.
INFERRED FROM	SR_REL_INFERRED_FROM	Source Content Item conveys a measurement or other inference made from the Target Content Items. Denotes the supporting evidence for a measurement or judgment. E.g.: CODE "Malignancy" {INFERRED FROM: CODE "Mass", INFERRED FROM: CODE "Lymphadenopathy",...}. E.g.: NUM: "BPD = 5mm" {INFERRED FROM: SCOORD}.
SELECTED FROM	SR_REL_SELECTED_FROM	Source Content Item conveys spatial or temporal coordinates selected from the Target Content Item(s). E.g.: SCOORD: "CLOSED 1,1 5,10" {SELECTED FROM: IMAGE 36}. E.g.: TCOORD: "SEGMENT 60-200mS" {SELECTED FROM: WAVEFORM}.

Content Item Identifier

Content Items are identified by their position in the Content Item tree. They have an implicit order as defined by the order of the Sequence Items. When a Content Item is the target of a by-reference relationship, its position is specified as the Referenced Content Item Identifier in the source Content Item. *Figure 19* illustrates an SR content tree and identifiers associated with each Content Item:

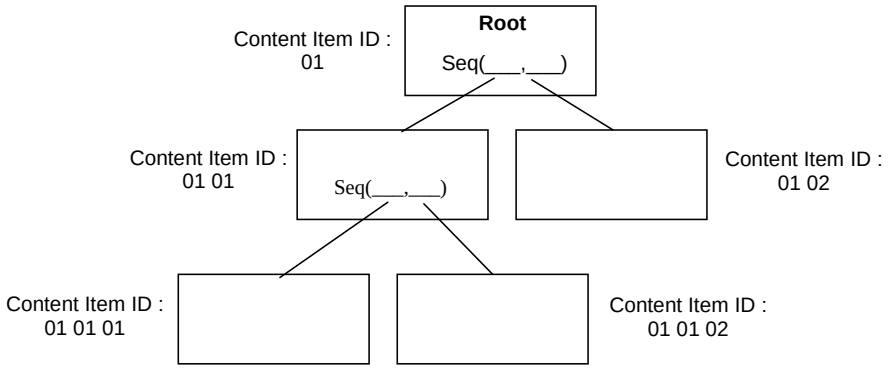

Figure 19: SR Item Identifier

Observation Context

Observation Context describes who or what is performing the interpretation, whether the examination of evidence is direct or quoted, what procedure generated the evidence that is being interpreted, and who or what is the subject of the evidence that is being interpreted.

Initial Observation Context is defined outside the SR Document Content tree by other modules in the SR IOD (i.e., Patient Module, Specimen Identification, General Study, Patient Study, SR Document Series, Frame of Reference, Synchronization, General Equipment and SR Document General modules). Observation Context defined by attributes in these modules applies to all Content Items in the SR Document Content tree and need not be explicitly coded in the tree. The initial Observation Context from outside the tree can be explicitly replaced.

If a Content Item in the SR Document Content tree has Observation Context different from the context already encoded elsewhere in the IOD, the context information applying to that Content Item shall be encoded as child nodes of the Content Item in the tree using the HAS OBS CONTEXT relationship. That is, Observation Context is a property of its parent Content Item.

The context information specified in the Observation Context child nodes (i.e. target of the HAS OBS CONTEXT relationship) adds to the Observation Context of their parent node Content item, and shall apply to all the by-value descendant nodes of that parent node regardless of the relationship type between the parent and the descendant nodes. Observation Context is encoded in the same manner as any other Content Item. Observation Context shall not be inherited across by-reference relationships.

Observation DateTime is not included as part of the HAS OBS CONTEXT relationship, and therefore is not inherited along with other Observation Context. The

Observation DateTime Attribute is included in each Content Item which allows different observation dates and times to be attached to different Content Items.

The IOD may specify restrictions on Content Items and Relationship Types that also constrain the flexibility with which Observation Context may be described.

The IOD may specify Templates that offer or restrict patterns and content in Observation Context.

Structured Reporting Templates

Templates are patterns that specify the Concept Names, Requirements, Conditions, Value Types, Value Multiplicity, Value Set restrictions, Relationship Types and other attributes of Content Items for a particular application. SR Document templates are defined in the Part 16 of the DICOM Standard. Part 17 of the DICOM also has some explanatory information on encoding SR Documents. The Merge DICOM Toolkit SR Functions follow DICOM Templates structures and allow straightforward encoding based on template tables.

SR Templates are described using tables of the form shown on *Table 27*.

Table 27: SR Template Definition

	NL	Rel with Parent	VT	Concept Name	VM	Req Type	Condition	Value Set Constraint
1								
2								
3								

Row Number

Each row of a Template Table is denoted by a row number. The first row is numbered 1 and subsequent rows are numbered in ascending order with increments of 1. This number denotes a row for convenient description as well as reference in conditions. The Row Number of a Content Item in a Template may or may not be the same as the ordinal position of the corresponding node in the encoded document. The Merge DICOM Toolkit does not use this number in any way.

Nesting Level (NL)

The nesting level of Content Items is denoted by ">" symbols, one per level of nesting below the initial Source Content Item (of the Template) in a manner similar to the depiction of nested Sequences of Items in Module Tables in Part 3 of the DICOM standard. When it is necessary to specify the Target Content Item(s) of a relationship, they are specified in the row(s) immediately following the corresponding Source Content Item. The Merge DICOM Toolkit provides functions to add nested (child) Content Items to the parent Content Item node. The following function pattern shall be used to add a child node with the specific type and relationship:

```
MC_SRH_Add_Type_Child((int AsrNodeID, SR_RELATIONSHIP
       Arelationship, …)
```

where *Type* is a child Content Item Type, for example "TEXT".

Relationship with Source Content Item (Parent)

Relationship Type and Mode are specified for each row that specifies a target content item. The Relationship Types are enumerated in *Table 26*.

Relationship Type and Mode may also be specified when another Template is included, either "top-down" or "bottom-up" or both (i.e., in the "INCLUDE Template" row of the calling Template or in all rows of the included Template or in both places). There shall be no conflict between the Relationship Type and Mode of a row that includes another Template and the Relationship Type and Mode of the rows of the included Template.

When the relationship is defined in a form as R-RTYPE, it means that Relationship Mode is "By-reference "and Relationship Type is "RTYPE". For example, "R-INFERRED FROM". Merge DICOM Toolkit provides the following functions to encode/decode references:

```
MC_STATUS EXP_FUNC MC_SRH_Add_Reference(
   int             AsrNodeID,
   SR_RELATIONSHIP Arelationship,
   int             AsrRefNodeID)

MC_STATUS EXP_FUNC MC_SRH_Get_Reference(
   int             AsrNodeID,
   int             *AsrRefNodeID)
```

Value Type (VT)

The Value Type field specifies the SR Value Type of the Content Item or conveys the word "INCLUDE" to indicate that another Template is to be included (substituted for the row). The Merge DICOM Toolkit uses explicit function call for each Content Item Type as is described above.

Concept Name

Any constraints on Concept Name are specified in this field as defined or enumerated coded entries or as baseline or defined context groups. Alternatively, when the VT field is "INCLUDE", the Concept Name field specifies the template to be included. The Merge DICOM Toolkit uses following arguments to specify the Concept Name:

```
const char          *AconceptNameValue,
const char          *AconceptNameScheme,
const char          *AconceptNameMeaning,
```

You will find that some of the functions do not include Concept Name arguments, because they are optional for those Content Item Types. In that case, a separate function can be used to set the Concept Name values as follows:

```
MC_STATUS EXP_FUNC MC_SRH_Set_Concept_Name(
int                 AsrNodeID,
const char          *AconceptNameValue,
const char          *AconceptNameScheme,
const char          *AconceptNameMeaning)
```

Templates defining References to coded concepts take the following form:

```
EV or DT (ConceptNameValue, ConceptNameSheme,
      "ConceptNameMeaning")
```

For example, EV (T-04000, SNM3, "Breast") would mean that hardcoded values shall be used for that Concept Name. Some template items don't have DT or EV abbreviation and just specify the hardcoded values.

Abbreviations used in templates

The following abbreviations are used in template definitions:

- **EV** Enumerated Value —values for are provided in the brackets.
- **DT** Defined Term —values are provided in the brackets.
- **BCID** Baseline Context Group ID — identifier that specifies the suggested Context Group. The suggested values can be found in DICOM Part 16 and identified by a Context ID provided in the brackets.
- **DCID** Defined Context Group ID — identifier that specifies the Context Group for a Coded Value that shall be used. The values can be found in DICOM Part 16 and identified by a Context ID provided in the brackets.
- **BTID** Baseline Template ID — identifier that specifies a template suggested to be used in the creation of a set of Content Items. The referenced template can be found in DICOM Part 16 and identified by a Template ID provided in the brackets.
- **DTID** Defined Template ID — identifier that specifies a template that shall be used in the creation of a set of Content Items. The referenced template can be found in DICOM Part 16 and identified by a Template ID provided in the brackets.

Value Multiplicity (VM)

The VM field indicates the number of times that either a Content Item of the specified pattern or an included Template may appear in this position. *Table 28* specifies the values that are permitted in this field.

Table 28: Permitted Values for VM

Expression	Definition
i (where 'i' represents an integer)	Exactly i occurrences, where i >= 1. E.g. when i == 1 there shall be one occurrence of the Content Item in this position.
i-j	From i to j occurrences, where i and j are >= 1 and j > i.
1-n	One or more occurrences

Requirement Type

The Requirement Type field specifies the requirements on the presence or absence of the Content Item or included Template. The following symbols are used:

- **M** — Mandatory. Shall be present.
- **MC** — Mandatory Conditional. Shall be present if the specified condition is satisfied.
- **U** — User Option. May or may not be present.
- **UC** — User Option Conditional. May not be present. May be present according to the specified condition.

Condition

The Condition field specifies any conditions upon which the presence or absence of the Content Item or its values depends. This field specifies any Concept Name(s) or Values upon which there are dependencies.

References may also be made to row numbers (e.g. to specify exclusive OR conditions that span multiple rows of a Template table).

The following abbreviations are used:

- **XOR** — Exclusive OR. One and only one row shall be selected from mutually exclusive options.

Note: For example, if one of rows 1, 2, 3 or 4 may be included, then for row 2, the abbreviation "XOR rows 1,3,4" is specified for the condition.

- **IF** — Shall be present if the condition is TRUE; may be present otherwise.
- **IFF** — If and only if . Shall be present if the condition is TRUE; shall not be present otherwise.
- **CV** — Code Value
- **CSD** — Coding Scheme Designator
- **CM** — Code Meaning
- **CSV** — Coding Scheme Version

Value Set Constraint

Value Set Constraints, if any, are specified in this field as defined or enumerated coded entries, or as baseline or defined context groups.

The Value Set Constraint column may specify a default value for the Content Item if the Content Item is not present, either as a fixed value, or by reference to another Content Item, or by reference to an Attribute from the dataset other than within the Content Sequence (0040,A730).

Inclusion of Templates

A Template may include another Template by specifying "INCLUDE" in the Value Type field and the identifier of the included Template in the Concept Name field. All of the rows of the specified Template are included in the invoking Template, effectively substituting the specified template for the row where the inclusion is invoked. Whether or not the inclusion is user optional, mandatory or conditional is specified in the Requirement and Condition fields. The number of times the included Template may be repeated is specified in the VM field.

We recommend that you implement templates as a subroutine or function call. In that case, the inclusion of the template will be implemented as a call to that template with passing parameters. Some of the templates defined in DICOM Part 16 already have predefined parameters and they are indicated by a name beginning with the character "$".

Memory Management

The Structured Reporting API is designed in such way that you only deal with types of objects:

- Message objects which are messages and message items.
- Structured Report objects which are the root SR Content Item and child Content Items.

You can convert back and forth between these objects and work with one object type at a time. However, it is imported to know how these objects are managed internally.

Structured Report Content Items are represented as a special SR objects in the memory. Each object is represented by the integer object ID that is also called a Node ID. The ID value for SR Objects is the same as the item ID or Message ID of the underlying message object. SR objects are always mapped to the message objects and use them as attribute storage. This allows you to use SR object ID in the basic toolkit functions like `MC_Set_Value`. Deletion of the SR Object does not delete the underlying item object. *Figure 20* shows the relationship between the SR tree and the message object:

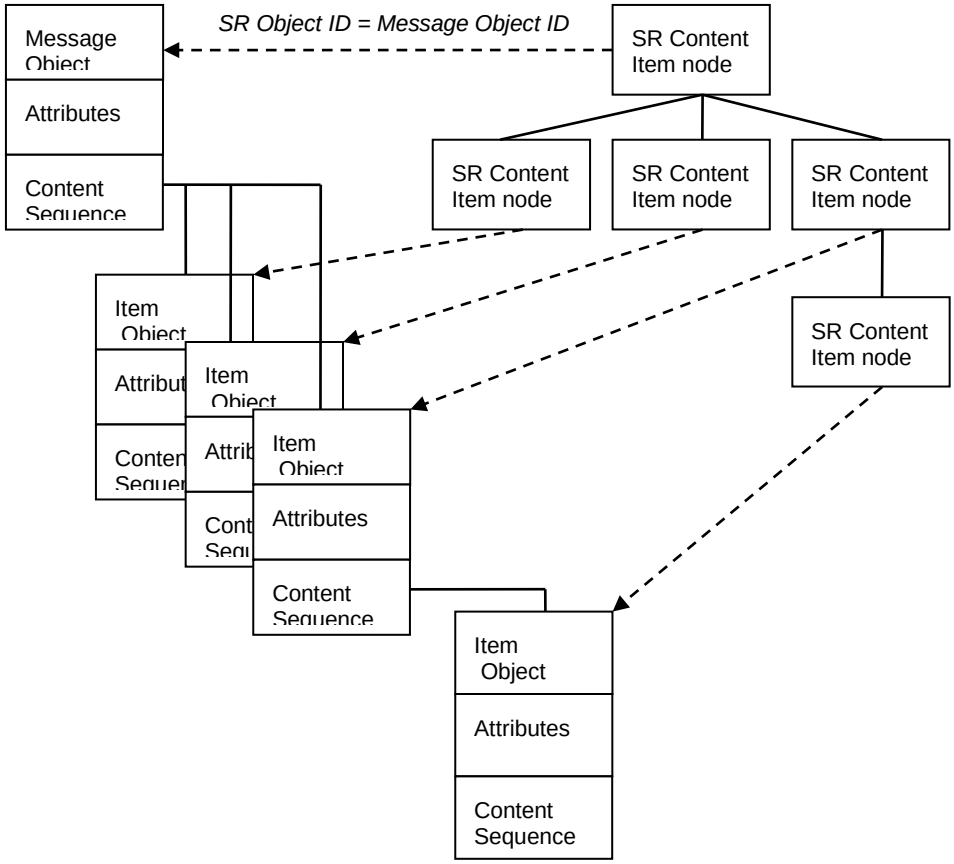

Figure 20: Relationship between SR object and the message object

When a message or item object is mapped with the SR object, it is marked with a special flag that prohibits some of the operations that can break the structured report hierarchy. For example, `MC_Duplicate_Message` will not work on the messages mapped with the SR object.

Overview of the Merge DICOM Toolkit SR functions

The Merge DICOM Toolkit has several types of functions that can be used for reading/writing Structured Report IODs.

Low-level attribute access functions — These are the same functions that are used to work with attributes in message objects. Every SR Content Item ID is mapped to the Message Item ID and can be used to set or get additional attributes that are not covered by the High Level API.

Low-level navigation and conversion functions — These functions provide mapping and conversion between message objects and SR objects (Content Items). Following functions are included:

- `MC_SR_Add_Root`. Creates a new SR root object and maps it to the existing message object.
- `MC_Message_To_SR`. Creates SR tree structure from the message and maps each SR Content Item with the corresponding message item.
- `MC_SR_Add_Child`. Creates a new SR Content Item and maps it to the existing message item.
- `MC_SR_Get_First_Child`. Retrieves the first child Content Item object ID.
- `MC_SR_Get_Next_Child`. Retrieves the next child Content Item object ID.
- `MC_SR_Delete_Child`. Deletes a child Content Item.
- `MC_SR_To_Message`. Releases memory allocated by SR tree objects and rebuilds underlying message object.

High-Level functions for encoding SR — These functions can be used to build an entire SR document tree with minimum coding. The following functions are included:

- `MC_SRH_Create_SR`. Creates an empty SR root object
- `MC_SRH_Add_`*`Type`*`_Child`. Adds a new child node, where *Type* is a child Content Item Type, for example "TEXT".
- `MC_SRH_Set_`*`Type_Data`*. Sets additional attributes for the Content Item node.
- `MC_SRH_Add_Reference`. Creates a reference to another Content Item node "by-reference".
- `MC_SRH_Free_SR`. Releases all memory associated with the SR object including the underlying message object. Can be used for cleanup in case of failure.

High-Level functions for reading SR — These functions quickly navigate the SR content tree and access Content Item's attributes. The following functions are included:

- `MC_SRH_Get_First_Child`. Retrieves the first child Content Item, relationship and node type.
- `MC_SRH_Get_Next_Child`. Retrieves the next child Content Item, relationship and node type.
- `MC_SRH_Get_Reference`. Retrieves the Content Item referenced "by-reference".
- `MC_SRH_Get_`*`Type`*`_Data`. Retrieves attributes of the Content Item, where *Type* is a child Content Item Type, for example "DATE".

High-Level utility functions — These functions are used internally by the other high-level functions and exposed to provide greater flexibility to the user. The ollowing functions are included in that group:

- `MC_SRH_Create_`*`Type`*`_Node`. Creates a new Content Item node specified by *Type*. The node is created as a standalone Message Item, without SR object.
- `MC_SRH_Add_Child`. Adds newly created Content Item node to an existing Content Item node as a child.

Encoding SR Documents

The creation of the SR document involves the following steps:

1. Create a new SR object as a root node.
2. Add Content Items (nodes) to the tree based on the templates definition.
3. Convert SR object to a message object.
4. Add Patient/Study/Series and other attributes required by the IOD definition,
5. Save the result message object to a file.

To create a new SR, you need to know the IOD type you are creating and the templates that will be used to generate the SR Document Content.

Key Object Selection Example

The Key Object Selection document is constrained by a single template. The following template is taken from DICOM Part 16.

TID 2010
KEY OBJECT SELECTION
Type: Non-Extensible

	NL	Rel with Parent	VT	Concept Name	VM	Req Type	Condition	Value Set Constraint
1			CONTAINER	DCID(7010) Key Object Selection Document Titles	1	M		Root node
2	>	HAS CONCEPT MOD	CODE	EV (113011, DCM, "Document Title Modifier")	1-n	U		
3	>	HAS CONCEPT MOD	CODE	EV (113011, DCM, "Document Title Modifier")	1	UC	IF Row 1 Concept Name = (113001, DCM, "Rejected for Quality Reasons") or (113010, DCM," Quality Issue")	DCID (7011)
4	>	HAS CONCEPT MOD	CODE	EV (113011, DCM, "Document Title Modifier")	1	MC	IF Row 1 Concept Name = (113013, DCM, "Best In Set")	DCID (7012)
5	>	HAS CONCEPT MOD	INCLUDE	DTID(1204) Language of Content Item and Descendants	1	U		
6	>	HAS OBS CONTEXT	INCLUDE	DTID(1002) Observer Context	1-n	U		
7	>	CONTAINS	TEXT	EV(113012, DCM, "Key Object Description")	1	U		
8	>	CONTAINS	IMAGE	Purpose of Reference shall not be present	1-n	MC	At least one of Rows 8, 9 and 10 shall be present	
9	>	CONTAINS	WAVEFORM	Purpose of Reference shall not be present	1-n	MC	At least one of Rows 8, 9 and 10 shall be present	
10	>	CONTAINS	COMPOSITE	Purpose of Reference shall not be present	1-n	MC	At least one of Rows 8, 9 and 10 shall be present	

The code below generates a valid DICOM KO object and illustrates how the template is encoded using the Merge DICOM Toolkit functions.

```
static char* fileName = "KO_demo.dcm";
MC_STATUS       status;
int srID, childID, tempId, item1, item2, item3;
unsigned short miVer = 0x0100;

/*
*  Create SR Document as well as the root CONTAINER.
*  The template ID is 2010 and we used the "Best in Set"
*  context ID from the CID 7010.
*/
status = MC_SRH_Create_SR("KEY_OBJECT_SELECTION_DOC", "2010",
        SR_CC_SEPARATE,"113013","DCM", "Best In Set", &srID );
if(status != MC_NORMAL_COMPLETION)
{
  /* process error here */return;
}
/*
*  Skipping Row 2 and 3 of the template and encoding Row 4.
*  The code is taken from the CID 7012.
*/
status = MC_SRH_Add_CODE_Child(srID, SR_REL_HAS_CONCEPT_MOD, "113011", "DCM",
        "Document Title Modifier", "113015", "DCM", "Series", &childID);
if(status != MC_NORMAL_COMPLETION)
{
  /* process error here */ return;
}
/*
* Skipping Row 5 and 6 of the template and encoding Row 7.
* The code is taken from the CID 7012.
* The text value shall describe the image selection.
*/
status = MC_SRH_Add_TEXT_Child(srID, SR_REL_CONTAINS,
  "113012", "DCM", "Key Object Description", "Doctor's comments on selection",
        &childID);
if(status != MC_NORMAL_COMPLETION)
{
  /* process error here */ return;
}
/*
* Adding an IMAGE from Row 8.
* The values "1.2.3.4.1", "1.2.3.4.5.1" suppose to be an image SOP Class
* and SOP Instance.
*/
status = MC_SRH_Add_IMAGE_Child(srID, SR_REL_CONTAINS,
  "1.2.3.4.1", "1.2.3.4.5.1", &childID);
if(status != MC_NORMAL_COMPLETION)
{
  /* process error here */ return;
}
/*
* Convert SR to the message
*/
status = MC_SR_To_Message(srID);
if(status != MC_NORMAL_COMPLETION)
{
  /* process error here */ return;
}
```

```
/*
* Add other root lvel attributes
* The error handling is skipped here in order to make a compact code
* Please add eeror handling if you use it.
*/

/** Set file meta information **/
MC_Set_Value_From_Buffer(srID, MEDIA_FILE_META_INFO_VER, &miVer, sizeof(miVer));
MC_Set_Value_From_String(srID, MEDIA_TRANSFER_SYNTAX_UID,
        "1.2.840.10008.1.2.1");
MC_Set_Value_From_String(srID, MEDIA_IMPLEMENTATION_CLASS_UID,
        "2.16.840.1.113669.2.931128");
MC_Set_Value_From_String(srID, MEDIA_IMPLEMENTATION_VER_NAME,
        "MergeCOM3_1.0");
MC_Set_Value_From_String(srID, MEDIA_STORAGE_SOP_CLASS_UID,
        "1.2.840.10008.5.1.4.1.1.88.59");
MC_Set_Value_From_String(srID, MEDIA_STORAGE_SOP_INSTANCE_UID,
        "1.2.3.4.5.6.7.300");

/** Set other required attributes for the KO IOD ***/
MC_Set_Value_From_String(srID, MC_ATT_SOP_CLASS_UID,
        "1.2.840.10008.5.1.4.1.1.88.59");
MC_Set_Value_From_String(srID, MC_ATT_SOP_INSTANCE_UID, "1.2.3.4.5.6.7.300");
MC_Set_Value_From_String(srID, MC_ATT_STUDY_DATE, "19991029");
MC_Set_Value_From_String(srID, MC_ATT_CONTENT_DATE, "19991029");
MC_Set_Value_From_String(srID, MC_ATT_STUDY_TIME, "154500");
MC_Set_Value_From_String(srID, MC_ATT_CONTENT_TIME, "154510");
MC_Set_Value_From_String(srID, MC_ATT_ACCESSION_NUMBER, "123456");
MC_Set_Value_From_String(srID, MC_ATT_MODALITY, "KO");
MC_Set_Value_From_String(srID, MC_ATT_MANUFACTURER, "MERGE" );
MC_Set_Value_From_String(srID, MC_ATT_REFERRING_PHYSICIANS_NAME,
        "Luke^Will^^Dr.^M.D." );
MC_Set_Value_To_NULL(srID,
        MC_ATT_REFERENCED_PERFORMED_PROCEDURE_STEP_SEQUENCE);
MC_Set_Value_From_String(srID, MC_ATT_PATIENTS_NAME, "Jane^Doo" );
MC_Set_Value_From_String(srID, MC_ATT_PATIENT_ID, "234567" );
MC_Set_Value_From_String(srID, MC_ATT_PATIENTS_BIRTH_DATE, "19991109" );
MC_Set_Value_From_String(srID, MC_ATT_PATIENTS_SEX, "F");
MC_Set_Value_From_String(srID, MC_ATT_STUDY_INSTANCE_UID,
        "1.2.3.4.5.6.7.100");
MC_Set_Value_From_String(srID, MC_ATT_SERIES_INSTANCE_UID,
        "1.2.3.4.5.6.7.200");
MC_Set_Value_From_String(srID, MC_ATT_STUDY_ID,  "345678");
MC_Set_Value_From_String(srID, MC_ATT_SERIES_NUMBER,  "1");
MC_Set_Value_From_String(srID, MC_ATT_INSTANCE_NUMBER,  "1");
MC_Set_Value_To_NULL(srID, MC_ATT_PERFORMED_PROCEDURE_CODE_SEQUENCE);
MC_Open_Item(&item1, "HIERARCHICAL_SOP_INST_REF_MACRO");
MC_Set_Value_From_String(item1, MC_ATT_STUDY_INSTANCE_UID,
        "1.2.3.4.5.6.7.100");
MC_Open_Item(&item2, "HIERARCHICAL_SERIES_REF_MACRO");
MC_Set_Value_From_String(item2, MC_ATT_SERIES_INSTANCE_UID,
        "1.2.3.4.5.6.7.200");
MC_Open_Item(&item3, "REF_SOP" );
/** following UIDs are the same as used in the Row 8 item ****/
MC_Set_Value_From_String(item3, MC_ATT_REFERENCED_SOP_CLASS_UID, "1.2.3.4.1");
MC_Set_Value_From_String(item3, MC_ATT_REFERENCED_SOP_INSTANCE_UID,
        "1.2.3.4.5.1");
MC_Set_Value_From_Int(item2, MC_ATT_REFERENCED_SOP_SEQUENCE, item3);
MC_Set_Value_From_Int(item1, MC_ATT_REFERENCED_SERIES_SEQUENCE, item2);
MC_Set_Value_From_Int(srID,
        MC_ATT_CURRENT_REQUESTED_PROCEDURE_EVIDENCE_SEQUENCE, item1);
```

```
/** Convert message to file **/
MC_Message_To_File(srID, fileName);
```

Reading SR Documents

Reading SR Documents is done in a similar way as encoding, but in reverse sequence:

- Read a File or receive a message object.
- Read root level attributes.
- Convert message object into SR object.
- Traverse SR content tree and extract Content Node attributes,

The following code demonstrates a reading sequence for the Key Object Document generated above.

```
MC_STATUS       status;
int srID, childID, tempId;
SR_CONTENT_TYPE    nodeType;
char    conceptNameValue[512];
char    conceptNameScheme[512];
char    conceptNameMeaning[512];
SR_RELATIONSHIP relationship;
int isLast;
char    sopClassUid[512];
char    sopInstanceUid[512];
int     refFramesCount;

/* convert file->message->SR objects */
status = MC_File_To_Message(srID);
if(status != MC_NORMAL_COMPLETION)
{
    printf("MC_File_To_Message Eror code %d", status);
    return;
}

/*
* Here you can read root level attributes from the message
*/
status = MC_Message_To_SR(srID);
if(status != MC_NORMAL_COMPLETION)
{
    printf("MC_Message_To_SR Eror code %d", status);
    return;
}

/* Get a concept name from the first CONTAINER node */
status = MC_SRH_Get_Concept_Name(srID, conceptNameValue,
        sizeof(conceptNameValue), conceptNameScheme, sizeof(conceptNameScheme),
        conceptNameMeaning, sizeof(conceptNameMeaning));
if(status != MC_NORMAL_COMPLETION)
{
    printf("MC_SRH_Get_Concept_Name Eror code %d", status);
    return;
```

```
}
printf("Document Title: %s\n", conceptNameMeaning);

/* Reading information about the first child node.
*  The following code assumes the Cntent Items type and their sequence.
*  To make it more generic you would need to create a recursive loop and
*  process each child based on the returned Content Item node type.
*/
status = MC_SRH_Get_First_Child(srID, &childID, &relationship, &nodeType,
       &isLast);
if(status != MC_NORMAL_COMPLETION)
{
    printf("MC_SRH_Get_First_Child Eror code %d", status);
    return;
}

/* First child expected to be the CODE item */
if(nodeType == SR_NODE_CODE)
{
    /* Print concept code + code value */
    status = MC_SRH_Get_Concept_Name(childID, conceptNameValue,
        sizeof(conceptNameValue), conceptNameScheme, sizeof(conceptNameScheme),
        conceptNameMeaning, sizeof(conceptNameMeaning));
    if(status != MC_NORMAL_COMPLETION)
    {
        printf("MC_SRH_Get_Concept_Name Eror code %d", status);
        return;
    }
    printf("%s: ", conceptNameMeaning);

    status = MC_SRH_Get_CODE_Data(childID, conceptNameValue,
        sizeof(conceptNameValue), conceptNameScheme, sizeof(conceptNameScheme),
        conceptNameMeaning, sizeof(conceptNameMeaning));
    if(status != MC_NORMAL_COMPLETION)
    {
        printf("MC_SRH_Get_CODE_Data Eror code %d", status);
        return;
    }
    printf("%s\n", conceptNameMeaning);
}

/* Reading information about the next child node. */
status = MC_SRH_Get_Next_Child(srID, &childID, &relationship, &nodeType,
       &isLast);
if(status != MC_NORMAL_COMPLETION)
{
    printf("MC_SRH_Get_Next_Child Eror code %d", status);
    return;
}

/* Second child expected to be the TEXT item */
if(nodeType == SR_NODE_TEXT)
{
    /* Print concept code + text value */
    status = MC_SRH_Get_Concept_Name(childID, conceptNameValue,
        sizeof(conceptNameValue),conceptNameScheme, sizeof(conceptNameScheme),
        conceptNameMeaning, sizeof(conceptNameMeaning));
    if(status != MC_NORMAL_COMPLETION)
    {
        printf("MC_SRH_Get_Concept_Name Eror code %d", status);
```

```
        return;
    }
    printf("%s: ", conceptNameMeaning);

    status = MC_SRH_Get_TEXT_Data(childID, strBuffer, sizeof(strBuffer));
    if(status != MC_NORMAL_COMPLETION)
    {
        printf("MC_SRH_Get_TEXT_Data Eror code %d", status);
        return;
    }
    printf("%s\n", strBuffer);
}

/* Reading information about the next child node. */
status = MC_SRH_Get_Next_Child(srID, &childID, &relationship, &nodeType,
        &isLast);
if(status != MC_NORMAL_COMPLETION)
{
    printf("MC_SRH_Get_Next_Child Eror code %d", status);
    return;
}

/* Next child expected to be the IMAGE item */
if(nodeType == SR_NODE_IMAGE)
{
    status = MC_SRH_Get_IMAGE_Data(childID, sopClassUid, sizeof(sopClassUid),
        sopInstanceUid, sizeof(sopInstanceUid), &refFramesCount);
    if(status != MC_NORMAL_COMPLETION)
    {
        printf("MC_SRH_Get_IMAGE_Data Eror code %d", status);
        return;
    }
    printf("Image SOP Class: %s, Image SOP Instance: %s\n", sopClassUid,
        sopInstanceUid);
}
```

Unicode Support

Check platform notes for Unicode support

Merge DICOM Toolkit supports DICOM character sets to Unicode conversion using a public domain library – ICU4C. The Unicode conversion libraries are optional and users that are not planning to use Unicode conversion don't need to deploy the extra two shared objects included in the distribution package.

The original copyright notice of the ICU4C software is below:

> ICU License - ICU 1.8.1 and later
>
> COPYRIGHT AND PERMISSION NOTICE
>
> Copyright (c) 1995-2012 International Business Machines Corporation and others
>
> All rights reserved.
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

APIs to perform Unicode conversion are listed below:

- `MC_Enable_Unicode_Conversion()`
- `MC_Byte_To_Unicode()`
- `MC_Unicode_To_Byte()`
- `MC_Get_Value_To_UnicodeString()`
- `MC_Get_Next_Value_To_UnicodeString()`
- `MC_Set_Value_From_UnicodeString()`
- `MC_Set_Next_Value_From_UnicodeString()`

A new type `MC_UChar` is introduced to represent a Unicode character storage unit. This storage unit, inherited from the ICU4C library, is implicitly defined as "`unsigned short`" on all supported platforms. This unit of storage sufficiently covers the ranges of all DICOM specified character sets. A Unicode string is treated as an array of `MC_UChar` with terminator (U+0000).

To use the Unicode conversion functions, `MC_Enable_Unicode_Conversion`() must be called first with a non-zero argument to initialize the conversion library. This involves loading the two supplied shared objects (ICU4C) in the distribution (refer to Platform Notes). The loading process also involves loading the dependency files of the ICU4C libraries. Users must ensure all dependency files are accessible at runtime.

All conversion functions require an output buffer and its size to be passed down as arguments. This means that the user must pre-allocate a big enough buffer with sufficient space to receive the output. If insufficient output space is detected, the functions will return `MC_BUFFER_TOO_SMALL`.

> As a rule of thumb, when converting from Unicode to a DICOM character set, create a buffer size equals to the number of Unicode characters times 16. The overestimation is to ensure that any escape sequence and terminator byte (NULL) can be accommodated. When converting from DICOM character set to Unicode, create a buffer size equals to ((number of input bytes + 1) * `sizeof(MC_Uchar)`). The assumption is that the DICOM character set can only produced one 16-bit Unicode character per input byte and one extra terminator byte (U+0000) at the end of the string.

There are two utility functions, `MC_Byte_To_Unicode()` and `MC_Unicode_To_Byte()` that convert DICOM character set to Unicode and back

respectively. These two functions perform conversion without requiring any message handle.

There are four functions, `MC_Get_Value_To_UnicodeString()`, `MC_Get_Next_Value_To_UnicodeString()`, `MC_Set_Value_From_UnicodeString()` and `MC_Set_Next_Value_From_UnicodeString()` dealing with getting and setting an attribute value with Unicode character string. The operations are similar to the regular `MC_Get_(Next_)Value_To_String()` and `MC_Set_(Next_)Value_From_String()` counterparts.

There are no equivalent functions to get/set private attribute with Unicode strings. To work around, the user can call `MC_Unicode_To_Byte()` and then `MC_Set_pValue...()` to set an attribute value or `MC_Get_pValue...()` and then `MC_Byte_To_Unicode()` to get an attribute value.

Finally, if the user prefers unloading the Unicode conversion library, calling `MC_Enable_Unicode_Conversion()` with zero as argument will unload the ICU4C libraries. Note that if the user calls `MC_Release_Library`(), the ICU4C library will NOT be unloaded automatically.

Below is an example usage of the utililty functions:

```
/* example to convert Unicode to Japanese character sets */
MC_STATUS status;
char *charsets[3];
MC_UChar *uChars = input; /* input is defined externally (a Unicode
    string) */
int numberOfUnicodeCharacter = -1; /* set to -1 only if Unicode string is
    U+0000 terminated, or set to exact number of Unicode characters */
int outputBufferSize;
char bytes; /* output */
int outLen; /* to receive the exact number of bytes produced (excluding
    terminator) */

/* enable Unicode conversion library FIRST, only need to do once in your
    application */
status = MC_Enable_Unicode_Conversion(1);
if(status != MC_NORMAL_COMPLETION)
    printf("failed to initialize Unicode library\n");
}

/* equivalent to DICOM attribute (0008,0005) but in array form */
charsets[0] = "ISO 2022 IR 13";
charsets[1] = "ISO 2022 IR 87";
charsets[2] = "ISO 2022 IR 159";
 /* sufficent buffer to hold output, each Unicode character wouldn't
    produce more than 16 bytes */
outputBufferSize = numberOfUnicodeCharacter * 16;
bytes = malloc(outputBufferSize);

/* convert from Unicode to DICOM character set */
status = MC_Unicode_To_Byte(charsets, 3, (const MC_UChar *)uChars,
    numberOfUnicodeCharacter, outputBufferSize, &outLen, bytes);
if(status == MC_BUFFER_TOO_SMALL)
{
    printf("fail because buffer too small, increase output buffer size\n");
}
```

```
else if(status != MC_NORMAL_COMPLETION)
{
    printf("failed to convert\n");
}


/* ------------------------------------------- */

/* example to convert Korean character set to Unicode */
MC_STATUS status;
char *firstCharset = ""; /* Korean dataset (0008,0005) is "\ISO 2022 IR
    149", always take the first charset which is "" */
char bytes = input; /* input is defined externally (a raw byte array, may
    contain escape sequence) */
int numberOfByte = -1; /* set to -1 if input is NULL terminated, or set it
    to exact number of input bytes */
MC_UChar *uChars; /* output */
int outputBufferSize;
int outLen; /* to receive the exact number of Unicode characters produced
    (excluding terminator) */

/* enable Unicode conversion library FIRST, only need to do once in your
    application */
status = MC_Enable_Unicode_Conversion(1);
if(status != MC_NORMAL_COMPLETION)
    printf("failed to initialize Unicode library\n");
}

 /* sufficent buffer to hold output, each input byte can produce one
    MC_UChar */
outputBufferSize = (numberOfByte+1) * sizeof(MC_UChar); /* add 1 for
    terminator U+0000 */
uChars = malloc(outputBufferSize);

/* convert from DICOM character set to Unicode */
status = MC_Byte_To_Unicode(firstCharset, bytes, numberOfByte,
    outputBufferSize, &outLen, uChars);
if(status == MC_BUFFER_TOO_SMALL)
{
    printf("fail because buffer too small, increase output buffer size\n");
}
else if(status != MC_NORMAL_COMPLETION)
{
    printf("failed to convert\n");
}
```

Below is an example usage of the Set and Get functions:

```
/* example of MC_Set_Value_From_UnicodeString */
int msgID; /* assigned elsewhere when message was created */
MC_UChar *inputStr; /* assigned elsewhere as the input Unicode source */
int inputLen = -1; /* if input is U+0000 terminated, or set to exact
    number of Unicode characters */

/* enable Unicode conversion library FIRST, only need to do once in your
    application */
status = MC_Enable_Unicode_Conversion(1);
if(status != MC_NORMAL_COMPLETION)
    printf("failed to initialize Unicode library\n");
}
```

```
status = MC_Set_Value_From_UnicodeString(msgID, MC_ATT_PATIENT_NAME,
    inputLen, inputStr);
if(status != MC_NORMAL_COMPLETION)
{
    printf("failed to set str\n");
}


/* ------------------------------------------- */

/* example of MC_Get_Value_To_UnicodeString */
int msgID; /* assigned elsewhere when message was created */
int rawLen;
MC_UChar *outputStr;
MC_size_t outputBufferSize;
int outLen; /* actual number of Unicode characters produced (excluding
    U+0000 terminator) */

/* enable Unicode conversion library FIRST, only need to do once in your
    application */
status = MC_Enable_Unicode_Conversion(1);
if(status != MC_NORMAL_COMPLETION)
    printf("failed to initialize Unicode library\n");
}

/* get raw attribute length */
status = MC_Get_Value_Length(msgID, MC_ATT_PATIENT_NAME, 1 /* value index
    starts from 1 */, &rawLen);
if(status != MC_NORMAL_COMPLETION)
{
    printf("failed to get length\n");
}
/* create sufficent output buffer */
outputBufferSize = (rawLen + 1) * sizeof(MC_UChar); /* add 1 for the
    U+0000 terminator */
outputStr = malloc(outputBufferSize);
status = MC_Get_Value_To_UnicodeString(msgID, MC_ATT_PATIENT_NAME,
    outputBufferSize, &outLen, outputStr );
if(status == MC_BUFFER_TOO_SMALL)
{
    printf("fail because buffer too small, increase output buffer size\n");
}
else if(status != MC_NORMAL_COMPLETION)
{
    printf("failed to get str\n");
}
```

Deploying Applications

There are several issues to consider when deploying a Merge DICOM Toolkit based application. These include deciding which Merge DICOM Toolkit 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.

Merge DICOM Toolkit Required Files

There are a limited number of files required by Merge DICOM Toolkit applications. These files are described in *Table 29*. Note that the use of some of these files can be avoided by using the `genconf` and `gendict` utilities. Each of these utilities generates a source file from the configuration files that can then be compiled and linked into your application.

Table 29: Files needed when deploying an application

File	Description and Use
merge.ini	Merge DICOM Toolkit initialization file. This file contains logging information and path names for the other configuration files. Use of this file can be avoided by using the `genconf` utility to link the file into the Merge DICOM Toolkit application.
mergecom.pro	Merge DICOM Toolkit 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 Merge DICOM Toolkit application.
mergecom.app	Merge DICOM Toolkit application profile. This file contains configuration information about the services supported by the Merge DICOM Toolkit application and information about remote DICOM applications. Use of this file can be avoided by using the `genconf` utility to link it into the Merge DICOM Toolkit application.
mergecom.srv	Merge DICOM Toolkit services file. This file contains information about the services supported by Merge DICOM Toolkit. Use of this file can be avoided by using the `genconf` utility to link it into the Merge DICOM Toolkit application.
mrgcom3.msg	Merge DICOM Toolkit 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 Merge DICOM Toolkit application.
mrgcom3.dct	Merge DICOM Toolkit 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 Merge DICOM Toolkit application.

Configuration Options

The majority of Merge DICOM Toolkit'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 Merge DICOM Toolkit application to help reduce potential problems. These options are listed in *Table 30* with descriptions of how they can be set.

Table 30: Configuration options to consider when deploying an application

Configuration Option	Description
ACCEPT_ANY_APPLICATION_TITLE	When set to NO, Merge DICOM Toolkit 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, Merge DICOM Toolkit 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 NO 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 NO 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 uniquely identify 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 Merge DICOM Toolkit application.

Configuring Remote Nodes for SCU Applications

Configuring remote nodes at run-time

Typical Merge DICOM Toolkit 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 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 Merge DICOM Toolkit 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 decides 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 Merge DICOM Toolkit'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 Merge DICOM Toolkit to not export attributes encoded as UN VR to the network. This option was added to release 2.3.0 of the Merge DICOM Toolkit.

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 Merge DICOM Toolkit.

Appendix A: Frequently Asked Questions

This appendix lists some frequently asked questions by Merge DICOM Toolkit users.

1. *I recently received a new version of Merge DICOM Toolkit and wonder what is required for me to upgrade to the new version?*

   There are several areas where changes typically occur between releases of the Merge DICOM Toolkit. The following are specific areas to look at when upgrading to a new version:

   - **Upgrade the header files** — The mc3msg.h, mergecom.h, mc3media.h, and mcstatus.h files typically change with each release. These files must be updated when moving to a new Merge DICOM Toolkit software release. There are several structures defined in these files which have been updated in the past to support new functionality. Not updating these files can cause problems with the library.

   - **Upgrading the library** — The Merge DICOM Toolkit library itself must be updated. It is recommended that your application be recompiled against the new version of the library instead of just replacing the library.

   - **Upgrading the data dictionary files** — The diction.h, mergecom.srv, mrgcom3.msg, and mrgcom3.dct files must all be updated when upgrading the Merge DICOM Toolkit data dictionary. Upgrading some, but not all of these files can cause subsequent problems.

   - **Upgrading the configuration files** — Upgrading the merge.ini, mergecom.pro, and mergecom.app configuration files is optional. Although new configuration options are often added to these files, Merge DICOM Toolkit will assume default values for these options if they are not included in a configuration file. These files do not have to be updated when moving to a new release. Note however that the descriptions of configuration options are often updated and it is useful to have the latest versions of these files.

2. *I am running the toolkit'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 toolkit 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 toolkit will be unable to find the files and produce this error. Changing these paths to absolute paths will fix the problem.

3. *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?*

Merge DICOM Toolkit 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 = $(MERGE_ROOT)\mc3apps\mergecom.pro
- In this example, MERGE_ROOT would be an environment variable set in a similar fashion as the MERGE_INI environment variable.
- A special macro "MC3INIDIR" is used to represent the directory where "merge.ini" is. It is used like the environment variable with the difference that it is automatically resolved and does not need to be set.
- If MERGECOM_3_PROFILE, MERGECOM_3_SERVICES or MERGECOM_3_APPLICATIONS contain relative paths with a prefix "$(MC3INIDIR)" or "%MC3INIDIR%", the toolkit considers the path relative to the location of the "merge.ini" file.

For example:

```
MERGECOM_3_PROFILE = $(MC3INIDIR)../config/mergecom.pro
```

The path of the profile file is "../config/mergecom.pro" relative to the location of the "merge.ini" file.

4. *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 hosts 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.

5. *What can be done to reduce the memory requirements of the Merge DICOM Toolkit?*

Performance Tuning

There are several methods for reducing the memory requirements of Merge DICOM Toolkit. 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 Merge DICOM Toolkit'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 Merge DICOM Toolkit 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.

- **DICOMDIR_STREAM_STORAGE** — This option can be used when reading DICOMDIR files to reduce the amount of memory required to store directory records within the DICOMDIR.

6. *What can be done to increase the performance of the Merge DICOM Toolkit?*

   There are several Merge DICOM Toolkit 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 toolkit that causes the toolkit 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 toolkit will receive. If during association negotiation the maximum sized PDU of the system negotiating with the toolkit application is larger than this value, the PDU size will be limited to this value.
     Setting this option so that a PDU fits within an even multiple of the default TCP/IP Maximum Segment Size (MSS) of 1460 bytes will increase performance. Note that 6 bytes for the PDU header must be added to the configured maximum PDU size when calculating a multiple of the MSS. Having the PDU Maximum length an even multiple of the MSS ensures that there are limited delays within TCP/IP stack when transferring. With the exception of the final TCP/IP packet for a message, all packets transferred should exactly fit within a TCP/IP packet.

   - **WORK_BUFFER_SIZE** — This option specifies how the toolkit 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 toolkit for server (SCP) applications. This value should also be slightly larger than the **PDU_MAXIMUM_LENGTH** to increase performance. Setting this value to an even multiple of the MSS (1460 bytes) will help increase performance on most platforms.

- **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 toolkit for client (SCU) applications. This value should also be slightly larger than the **PDU_MAXIMUM_LENGTH** to increase performance. Setting this value to an even multiple of the MSS (1460 bytes) will help increase performance on most platforms.

- **EXPORT_UNDEFINED_LENGTH_SQ** — This option determines how Merge DICOM Toolkit 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 Merge DICOM Toolkit to determine the length of sequences and increases performance.

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

- **EXPORT_GROUP_LENGTHS_TO_MEDIA** — This option determines if Merge DICOM Toolkit encodes group length attributes when writing to files. Setting this option to No increases Merge DICOM Toolkit performance. This eliminates the need for Merge DICOM Toolkit to determine the length of groups when writing to media.

- **EXPORT_UNDEFINED_LENGTH_SQ_IN_DICOMDIR** — This option determines how Merge DICOM Toolkit 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 Merge DICOM Toolkit no longer needs to calculate the length of sequence attributes in DICOMDIRs.

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

   - The **PDU_MAXIMUM_LENGTH**, **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 Merge DICOM Toolkit.

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

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

   The Merge DICOM Toolkit 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, we expect 8-bit data to be byte swapped on big endian machines. 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.

9. *I recently upgraded to a new release of the Merge DICOM Toolkit. 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 Merge DICOM Toolkit 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

10. *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 Merge DICOM Toolkit 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()` functions to create an object, Merge DICOM Toolkit 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`, Merge DICOM Toolkit will return `MC_INVALID_TAG` for any attribute that does not have a value defined.

11. *I am trying to assign the value to a DICOM attribute within a message, but Merge DICOM Toolkit 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.

12. *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` or `MC_INVALID_VALUE_FOR_VR` is returned, the value is still encoded. It is a common mistake in Merge DICOM Toolkit applications 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.

Sample UID Format

There are many methods that can be used to ensure the uniqueness of a UID. The following is one example encoding of a UID to ensure uniqueness:

```
<root>.<serial>.<process id>.<timestamp>.<count>
```

In this example, `<root>` is the assigned root UID for an organization. The `<serial>` component would be a unique serial number assigned to the product within the organization. It may also be an encoding of the MAC address assigned to an Ethernet card in the system. This field along with the root gives a base for the UIDs that is unique for a specific device. The remaining components ensure that the UID is unique within that device.

The `<process id>` field would be a process ID or thread ID for the process generating the UID. The `<timestamp>` field would be a timestamp generated when the process or thread is created. Finally, the `<count>` field would be a unique counter that is incremented for each UID created.

Some vendors also include fields within the UID to identify the type of UID. For example, the first component after the root within the UID may be a ".1" for Study Instance UIDs, a ".2" for Series Instance UIDs, and".3" for SOP Instance UIDs. Occasionally a product identifier will also be included within a UID. This may be a unique number assigned to a product within an organization. Finally, a number may also be added to signify the software revision number for a product that is generating the UID.

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

Appendix C: XML and JSON Structures

The Merge DICOM Toolkit provides an API to convert a DICOM message, file or item into an XML or JSON string.

The conversion to XML might be done based on Merge DICOM Model using `MC_Message_To_XML()`API or using Native DICOM Model with `MC_Message_To_XML_Native()`API. These functions are detailed in the Reference Manual.

The structure of the Merge DICOM XML string created from a DICOM message by the `MC_Message_To_XML()` API looks like the following:

XML structure with the Base64 encoding of bulks and attributes with VR UN:

```
<?xml version="1.0" encoding="utf-8"?>
<DcmFile>
  <FileMetaInfo Service="STANDARD_SEC_CAPTURE"
       Command="C_STORE_RQ">
    <Attribute Tag="00020001" VR="OB" Name="File Meta Information
       Version" Length="2">AAE=</Attribute>
    <Attribute Tag="00020002" VR="UI" Name="Media Storage SOP
       Class UID" Length="25">...</Attribute>
    <Attribute Tag="00020003" VR="UI" Name="Media Storage SOP
       Instance UID" Length="29">...</Attribute>
    <Attribute Tag="00020010" VR="UI" Name="Transfer Syntax UID"
       Length="19">1.2.840.10008.1.2.1</Attribute>
    .....
    <Attribute Tag="00020016" VR="AE" Name="Source Application
       Entity Title" Length="15">MERGE_STORE_SCP</Attribute>
  </FileMetaInfo>
  <DataSet Service="STANDARD_SEC_CAPTURE" Command="C_STORE_RQ"
       TransferSyntax="1.2.840.10008.1.2.1">
    <Attribute Tag="00080008" VR="CS" Name="Image Type"
       Length="24">ORIGINAL\SECONDARY\OTHER</Attribute>
    <Attribute Tag="00080016" VR="UI" Name="SOP Class UID"
       Length="25">1.2.840.10008.5.1.4.1.1.7</Attribute>
    .....
    <Attribute Tag="00080020" VR="DA" Name="Study Date"
       Length="8">20020717</Attribute>
    <Attribute Tag="00080030" VR="TM" Name="Study Time"
       Length="6">123429</Attribute>
    <Attribute Tag="00080060" VR="CS" Name="Modality"
       Length="2">OT</Attribute>
    .....
    <Attribute Tag="00081111" VR="SQ" Name="Referenced Performed
       Procedure Step Sequence" Length="1">
      <Item>
        <Attribute Tag="00081150" VR="UI" Name="Referenced SOP
       Class UID" Length="23">1.2.840.10008.3.1.2.3.3</Attribute>
        <Attribute Tag="00081155" VR="UI" Name="Referenced SOP
       Instance UID"
       Length="44">2.16.840.1.113669.4.960070.844.1026926027.44</A
       ttribute>
      </Item>
    </Attribute>
```

```
    <Attribute Tag="00090010" VR="LO" Name="Private Creator Code"
       PCode="PrivateCode" Length="11">SAMPLE PCODE</Attribute>
    <Attribute Tag="00091010" VR="LO" Name="Private" PCode="SAMPLE
       PCODE" Length="6">Value1</Attribute>
    <Attribute Tag="00091015" VR="UN" Name="Private" PCode="SAMPLE
       PCODE" Length="6">INAgNAEy</Attribute>
    .....
    <Attribute Tag="00100010" VR="PN" Name="Patient's Name"
       Length="28">Last^First</Attribute>
    .....
    <Attribute Tag="7FE00010" VR="OW" Name="Pixel Data"
       Encoding="Base64"
       Length="262144">HQAABgMAAAIHBAM.....</Attribute>
  </DataSet>
</DcmFile>
```

XML structure with the default encoding of bulks and attributes with VR UN:

```
<?xml version="1.0" encoding="utf-8"?>
<DcmFile>
  <FileMetaInfo Service="STANDARD_SEC_CAPTURE"
       Command="C_STORE_RQ">
    <Attribute Tag="00020001" VR="OB" Name="File Meta Information
       Version" Length="2">00 01</Attribute>
    .....
    <Attribute Tag="00020016" VR="AE" Name="Source Application
       Entity Title" Length="15">MERGE_STORE_SCP</Attribute>
  </FileMetaInfo>
  <DataSet Service="STANDARD_SEC_CAPTURE" Command="C_STORE_RQ"
       TransferSyntax="1.2.840.10008.1.2.1">
    <Attribute Tag="00080008" VR="CS" Name="Image Type"
       Length="24">ORIGINAL\SECONDARY\OTHER</Attribute>
    <Attribute Tag="00080016" VR="UI" Name="SOP Class UID"
       Length="25">1.2.840.10008.5.1.4.1.1.7</Attribute>
    .....
    <Attribute Tag="00081111" VR="SQ" Name="Referenced Performed
       Procedure Step Sequence" Length="1">
      <Item>
        <Attribute Tag="00081150" VR="UI" Name="Referenced SOP
       Class UID" Length="23">1.2.840.10008.3.1.2.3.3</Attribute>
        <Attribute Tag="00081155" VR="UI" Name="Referenced SOP
       Instance UID"
       Length="44">2.16.840.1.113669.4.960070.844.1026926027.44</A
       ttribute>
      </Item>
    </Attribute>
    <Attribute Tag="00090010" VR="LO" Name="Private Creator Code"
       PCode="PrivateCode" Length="11">SAMPLE PCODE</Attribute>
    <Attribute Tag="00091010" VR="LO" Name="Private" PCode="SAMPLE
       PCODE" Length="6">Value1</Attribute>
    <Attribute Tag="00091015" VR="UN" Name="Private" PCode="SAMPLE
       PCODE" Length="6">20 20 20 20 20 30y</Attribute>
    .....
    <Attribute Tag="7FE00010" VR="OW" Name="Pixel Data"
       Encoding="Base64" Length="262144">06 00 04 00 04 00 02 00
       03.....</Attribute>
  </DataSet>
</DcmFile>
```

The structure of the Native DICOM XML string created from a DICOM message by the `MC_Message_To_XML_Native()` API looks like the following:

XML structure with the Base64 encoding of bulks and attributes with VR UN:

```
<?xml version="1.0" encoding="utf-8"?>
<NativeDicomModel
       xsi:schemaLocation="http://dicom.nema.org/PS3.19/models/Nat
       iveDICOM" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
       instance"
       xmlns="http://dicom.nema.org/PS3.19/models/NativeDICOM">
 <DicomAttribute tag="00020001" vr="OB"
       keyword="FileMetaInformationVersion"><InlineBinary>AAE=
 </InlineBinary></DicomAttribute>
<DicomAttribute tag="00020010" vr="UI"
       keyword="TransferSyntaxUID">
  <Value number="1">1.2.840.10008.1.2.1</Value>
 </DicomAttribute>
 <DicomAttribute tag="00080005" vr="CS"
       keyword="SpecificCharacterSet">
  <Value number="1">ISO 2022 IR 13</Value>
  <Value number="2">ISO 2022 IR 87</Value>
 </DicomAttribute>
 <DicomAttribute tag="00080008" vr="CS" keyword="ImageType">
  <Value number="1">ORIGINAL</Value>
  <Value number="2">DERIVED</Value>
  <Value number="3">CAPTURE</Value>
 </DicomAttribute>
<DicomAttribute tag="00081050" vr="PN"
       keyword="PerformingPhysicianName">
  <PersonName number="1">
   <Alphabetic>
    <FamilyName>Family</FamilyName>
    <GivenName>Given</GivenName>
   </Alphabetic>
  </PersonName>
 </DicomAttribute>
 <DicomAttribute tag="00081111" vr="SQ"
       keyword="ReferencedPerformedProcedureStepSequence">
  <Item number="1">
   <DicomAttribute tag="00081150" vr="UI"
       keyword="ReferencedSOPClassUID">
    <Value number="1">1.2.840.10008.3.1.2.3.3</Value>
   </DicomAttribute>
   <DicomAttribute tag="00081155" vr="UI"
       keyword="ReferencedSOPInstanceUID">
    <Value
       number="1">1.2.392.200036.9116.6.14.36309661475.20080417.15
       3441.22178</Value>
   </DicomAttribute>
  </Item>
<DicomAttribute tag="0040A13A" vr="DT"
       keyword="ReferencedDateTime">
  <Value number="1">20081212115553</Value>
  <Value number="2">20081212115559</Value>
 </DicomAttribute>
 <DicomAttribute tag="7FDF1040" vr="SQ" keyword="Private"
       privateCreator="PRIVATE">
  <Item number="1">
```

```
    <DicomAttribute tag="00200011" vr="IS" keyword="SeriesNumber">
     <Value number="1">70</Value>
     <Value number="2">82</Value>
    </DicomAttribute>
    <DicomAttribute tag="7FDF0010" vr="LO" keyword="Private Creator
        Code" privateCreator="PRIVATE2">
     <Value number="1">PRIVATE2</Value>
    </DicomAttribute>
    <DicomAttribute tag="7FDF1050" vr="ST" keyword="Private"
        privateCreator="PRIVATE2">
     <Value number="1">100</Value>
    </DicomAttribute>
   </Item>
  </DicomAttribute>
  </NativeDicomModel>
```

The structure of the DICOM JSON string created from a DICOM message by the `MC_Message_To_Json()` API looks like the following:

```
  <?xml version="1.0" encoding="utf-8"?>
  {
    "00020001": {
      "vr": "OB",
      "InlineBinary": "AAE="
    },
    "00020010": {
      "vr": "UI",
      "Value": [
        "1.2.840.10008.1.2.1"
      ]
    },
    "00080005": {
      "vr": "CS",
      "Value": [
        "ISO 2022 IR 13",
        "ISO 2022 IR 87"
      ]
    },
    "00080008": {
      "vr": "CS",
      "Value": [
        "ORIGINAL",
        "DERIVED",
        "CAPTURE"
      ]
    },
    "00081050": {
      "vr": "PN",
      "Value": [
        {
          "Alphabetic": "Family^Given"
        }
      ]
    },
    "00081111": {
      "vr": "SQ",
      "Value": [
        {
          "00081150": {
```

```
            "vr": "UI",
            "Value": [
              "1.2.840.10008.3.1.2.3.3"
            ]
          },
          "00081155": {
            "vr": "UI",
            "Value": [

         "1.2.392.200036.9116.6.14.36309661475.20080417.153441.22178
         "
            ]
          }
        }
      ]
    },
    "0040A13A": {
      "vr": "DT",
      "Value": [
        "20081212115553",
        "20081212115559"
      ]
    },
    "7FDF1040": {
      "vr": "SQ",
      "Value": [
          "00200011": {
            "vr": "IS",
            "Value": [
              70,
              82
            ]
          },
          "7FDF0010": {
            "vr": "LO",
            "Value": [
              "PRIVATE2"
            ]
          },
          "7FDF1050": {
            "vr": "ST",
            "Value": [
              "100"
            ]
          }
      ]
    }
  }
```

Appendix D: XML License

The Merge DICOM Toolkit supports DICOM to XML and XML to DICOM conversions through the use of a public domain library: *libxml2*. Usage of the *libxml2* library is governed by the MIT License and Copyright notice. The original content of the MIT License and Copyright notice as shown below:

Appendix E: JSON License

The Merge DICOM Toolkit supports DICOM to JSON and JSON to DICOM conversions through the use of a public domain library: *jansson*. Usage of the *jansson* library is governed by the MIT License and Copyright notice. The original content of the MIT License and Copyright notice as shown below: