DICOM® Connectivity Framework (DCF™)

DCF Developers Guide

Released with DCF version 3.3.68c

LAUREL BRIDGE

Orchestrating Medical Imaging Workflow

Laurel Bridge Software, Inc.
302-453-0222
www.laurelbridge.com

Document Version: 2.55

Document Number: LBDC-000018-0255

Last Saved: 6/11/2020 10:32:00 PM

® DICOM is the registered trademark of the National Electrical Manufacturers Association for its standards publications relating to digital communications of medical information.

What's new in this version of the manual?

- Updated version.
- Updated supported platform configurations.

Table of Contents

1. Overview 1
1.1. System Architecture Overview 1
1.2. An Example DCF Application 2
1.3. Components 4
1.3.1. Component Classification 4
1.3.1.1. DCF Component Categories 4
1.3.1.2. DCF Component Physical Types 4
1.3.2. Selected DCF Components Organized by Category 5
1.3.2.1. DICOM Applications 5
1.3.2.2. Non DICOM Applications 6
1.3.2.3. Examples 7
1.3.2.4. Common Service Interface Libs 9
1.3.2.5. Common Service Implementation Libs 9
1.3.2.6. Application Support Libs 10
1.3.2.7. IDL Interface Libs 11
1.4. Platforms 12
1.5. Systems 13
2. Installing the DCF 15
2.1. Multi-user vs. Single-user Installation 15
2.2. DCF Shared Files 15
2.3. DCF Per-user Files 16
2.4. The DCF Remote Service Interface 19
2.4.1. Running the Apache Web Server 19
2.4.1.1. Alternate Web Servers 19
2.4.2. Connecting to the web server 20
2.4.3. The DCF Remote Service Interface 20
2.4.3.1. Start with … (Choose a configuration) 21
2.4.3.2. Shutdown DCF Processes 21
2.4.3.3. Clear Log Files 21
2.4.3.4. View Log Files 22
2.4.3.5. View DCF Real-Time Log 24
2.4.3.6. View/Edit Configuration Files 25
2.4.3.7. View DCF Online Documentation 26
2.4.3.8. Set Debug Flags 27
2.4.3.9. Configure [DCF system config name] 29
2.4.3.10. Edit Global Filter Sets 31
2.4.4. The DCF Command-line Operation 32
2.5. Using Multiple Versions of the DCF 33
2.5.1. UNIX 33
2.5.2. Windows 33
2.5.3. Testing 33
2.6. Windows x86 vs. x64 34
3. DICOM Programming Overview 35
3.1. Core DCF DICOM classes 35
3.1.1. Element related 35
3.1.2. Association Manager 35
3.2. Verification Service Class 36
3.2.1. Verification Client (SCU) 36
3.2.2. Verification Server (SCP) 36
3.3. Storage-related Service Classes 36
3.3.1. Store Client (SCU) 36
3.3.2. Store Server (SCP) 36
3.3.3. Storage Commitment Server (SCP) 37
3.3.4. Storage Commitment Client (SCU) 37
3.3.5. Storage Commitment Client Agent 37

3.4. Query/Retrieve (Q/R) Service Class 38
3.4.1. Q/R Client (SCU) 38
3.4.2. Q/R Server (SCP) 38
3.5. Modality Worklist Service Class 39
3.5.1. MWL SCU 39
3.5.2. MWL Server (SCP) 39
3.6. Modality Performed Procedure Step Service Class 39
3.6.1. MPPS Client (SCU) 39
3.6.2. MPPS Server (SCP) 39
3.7. Print Service Class 40
3.7.1. Print Client (SCU) 40
3.7.2. Print Server (SCP) – (C++ only) 40
3.8. DICOM File (Media Storage) Services 40
3.8.1. DICOM File Set Reader (FSR role) 40
3.8.2. DICOM File Set Creator and Updater (FSC and FSU roles) 40
4. C++ Programming Examples 41
4.1. Running Example Servers 41
4.1.1. Using the Web Service Interface 41
4.1.2. Using a DCF Command Prompt – w/Common Services 41
4.1.3. Using a DCF Command Prompt – w/Minimal Resources 42
4.2. DICOM Programming Examples 42
4.2.1. Reading a DICOM file and extracting an element from the header 42
4.2.2. Creating a DICOM file that contains image data and patient demographics 44
4.2.3. Using the C++ StoreClient 45
4.2.3.1. Creating a job from DicomPersistentObjectDescriptors 45
4.2.3.2. Using C++ StoreClient to C-Store DicomDataSets in memory 45
4.2.4. Using the C++ PrintClient 47
4.2.5. Media Storage Application Profiles – DICOMDIR files 49
4.2.5.1. Example – Creating a DICOMDIR 49
4.2.5.2. Example – Adding to a DICOMDIR 50
4.2.5.3. Example – Reading a DICOMDIR 51
4.3. Deploying a Simple Standalone DCF C++ Application 51
4.4. Common Services Programming Examples 53
4.4.1. C++ “hello world” Example Application Using the DCF 53
4.4.2. Using the LOG interface – Logging from C++ programs 56
4.4.3. Using the CDS interface 57
4.4.4. Using the APC interface 57
4.5. Advanced DICOM Programming Examples 57
4.5.1. Writing a customized storage SCP 57
4.5.2. Writing a customized query retrieve SCP 61
4.6. Using the C++ Modality Worklist examples 64
4.7. DICOM compression transfer syntax support for C++ 64
5. Java Programming Examples 66
5.1. Running Example Servers 66
5.2. Using the DCF with Java IDE tools 66
5.2.1. Using the DCF with Eclipse for Java 66
5.3. DICOM Programming Examples 67
5.3.1. Using Java Print Element Value Program 67
5.3.1.1. Example – ex_jprint_element_value 68
5.3.2. Using Java modify DICOM image data program 69
5.3.2.1. Example – ex_jModPixelData 69
5.3.3. Using Java DICOM Verification (Echo) Client Classes 71
5.3.3.1. Example – ex_jecho_scu 71
5.3.4. Using Java Modality Worklist Query SCU Classes 73
5.3.4.1. Example – ex_jmwl_client 73
5.3.5. Using Java Print Client Classes 76
5.3.5.1. Example – ex_jprint_client 77
5.3.6. Using Java Store Client Classes 80

5.3.6.1. Example – ex_jstore_client 80
5.3.7. Using Java Q/R Client Classes 82
5.3.7.1. Example – ex_jqr_scu 82
5.3.8. Handling alternate character sets with DCF (Java) 84
5.4. Deploying a Simple Standalone DCF Java Application 84
5.5. Common Services Programming Examples 86
5.5.1. Java "Hello World" Example Using DCF Common Services 86
5.5.2. Using the LOG interface – Logging from Java programs 91
5.5.3. Using the CDS interface 92
5.5.4. Using the APC interface 92
5.6. Advanced DICOM Programming Examples 92
5.6.1. Using StorageCommitmentSCU 92
5.6.1.1. Example – Send store commit requests, and receive StoreCommitClientListener notifications 92
5.6.2. Using Java MWLClient Classes 94
5.6.2.1. Example – ex_jmwl_scu 94
5.6.3. Using Java MPPSClient Classes 94
5.6.3.1. Example – ex_jmpps_scu 94
5.6.4. Using Java Store, Q/R, and MWL Server-Related Classes 95
5.6.4.1. Example – ex_jstore_scp 95
5.6.4.2. Example – ex_jqr_scp 99
5.6.4.3. Example – ex_jmwl_scp 100
5.6.4.4. Using the MWL Server as an MPPS Server 100
5.6.4.5. Example – Server Objects 100
5.6.4.6. Example – StoreSCP: Implementing a custom storeObject method 100
5.6.4.7. Additional coding examples: 102
5.6.4.8. Writing a Custom DICOM SCP 102
5.7. DICOM Compression Transfer Syntax Support for Java 102
5.7.1. Example – ex_jdcf_filter: Uncompressed to Compressed 103
5.7.2. Example – ex_jdcf_filter: Compressed to Uncompressed 103
5.8. JAI – DCF integration for Java 103
5.8.1. Example – ex_jdcf_dcm2jai: Convert a DICOM file to a JAI type 104
5.8.2. Example – ex_jdcf_jai2dcm: Convert JAI types to a DICOM file 105
5.8.3. Example – ex_jdcf_dcmview: View a DICOM Image file 105
5.8.4. Example – Writing a JPEG Image with Java 105

6. C# Programming Examples 107
6.1. Running Example Servers 107
6.2. Using the DCF with MS Visual Studio .NET 2003/2005 for C# 108
6.2.1. Opening an Existing C# Example Project 108
6.2.2. Quick Start - Using `create_cs_comp.pl` to generate VS project files and source code 109
6.2.3. Using `dcfmake.pl` to generate a `.csproj` file 109
6.2.4. Manually Creating C# Projects from MS Visual Studio 2003/2005 IDE 109
6.3. DICOM Programming Examples 110
6.3.1. DICOM File Access 110
6.3.1.1. Example – Open a DICOM file 110
6.3.1.2. Example – Create a DICOM file 110
6.3.1.3. Example – Create a DICOM file from Config Group Data 111
6.3.1.4. Example – Open, Modify, and Save a DICOM file 112
6.3.2. Using VerificationClient 113
6.3.2.1. Example – Connect to a Verification SCP 113
6.3.3. Using StoreClient 114
6.3.3.1. Example – Create and submit a store job from files on disk 114
6.3.3.2. Example – Create and submit a store job – Handling Events 115
6.3.4. Using Query/Retrieve classes 115
6.3.4.1. Example – Using Query/Retrieve 117
6.3.5. Using PrintClient 117
6.3.5.1. Example – Create and submit a print job, handling status events 119
6.3.5.2. Example – Create and submit a print job where ImageBox data comes from DICOM disk files. 119
6.3.6. Creating and Populating DICOM Sequences 120
6.3.6.1. Example – Explicitly creating DICOM Sequence elements 120

6.3.6.2. Example – Setting DICOM Sequence elements using a config group 121
6.3.7. Handling alternate character sets with DCF (C#) 122
6.4. Deploying a Simple C# Standalone Application 123
6.4.1. Deploying a Simple C# Standalone DCF Application 123
6.4.2. Deploying a Simple C# Standalone OEM Application 125
6.5. Common Services Programming Examples 127
6.5.1. C# "hello world" Example Application Using the DCF 127
6.5.2. Using the LOG interface – Logging from C# programs 131
6.5.3. Avoiding or Embracing use of the Common Services 131
6.5.4. Using the CDS interface 133
6.5.5. Using the APC interface 133
6.6. Advanced DICOM Programming Examples 133
6.6.1. Using StorageCommitmentSCU 133
6.6.1.1. Example – Send store commit requests and receive StoreCommitClientListener notifications 134
6.6.2. Using MWLClient 135
6.6.2.1. Example – Send Worklist Query and wait for all responses before continuing 135
6.6.2.2. Example – Send Worklist Query and handle responses as they arrive 135
6.6.3. Using MPPSClient 135
6.6.3.1. Example – MPPSClient Console Application 135
6.6.3.2. Example – Send DIMSE N-CREATE or N-SET messages to a MPPS Server 135
6.6.4. C# Store, Q/R, and MWL Server-Related Examples 136
6.6.4.1. Using the MWL Server as an MPPS Server 137
6.6.4.2. Example – Implementing a custom storeObject() method 138
6.6.4.3. Example – How DicomDataService (DDS) gets called: 139
6.6.4.4. Example – Adding OEM specific data to DicomSessionSettings: 140
6.6.4.5. Example – Receiving or Logging Retired SOP classes: 140
6.6.4.6. Writing a Custom DICOM SCP 141
6.7. DICOM compression transfer syntax support for C# 141

7. Using DCF System Manager to control processes 143
7.1. Installing and Starting the System Manager 143
7.1.1. Installing and starting as a service on Windows 143
7.1.2. Installing and starting as a normal server process on Windows 143
7.1.3. Installing and starting on Unix 144
7.2. System Manager Related Interfaces 144
7.3. System Manager Configuration 145
7.3.1. System Manager Application Configuration 145
7.3.2. System Manager System Configuration 146
7.4. System startup for a DCF server application 149
7.5. System shutdown for a DCF server application 150

8. The DCF Development Environment 151
8.1. Using the dcfmake.pl utility 151
8.1.1. Command line options for dcfmake.pl 151
8.1.2. The cinfo.cfg file 152
8.1.3. Generated files for various component types 155
8.2. Example: Creating a DCF library component 158
8.3. Example: Creating a DCF application component 159
8.4. Using iodgen to create DICOM dataset wrappers to represent an IOD 160

9. Configuring DCF Applications 163
9.1. Configuration Files and the CDS interface 163
9.1.1. Using cds_client to access data in the configuration database 164
9.1.2. Receiving notifications of updated data 164
9.2. Application and Process Configurations 165
9.2.1. Application Configuration Settings 165
9.2.1.1. Structure of an application or process configuration 166
9.3. Process Configuration Settings 166
9.3.1. Process configuration with AppControl setup 166
9.3.1.1. Monitoring the Process Configuration 167
9.3.2. Process configuration without AppControl setup 167

9.3.3. Creating a custom application configuration 167
9.4. Log/Debug tracing control using “`debug_flags`” 169
9.4.1. Example – Setting Debug Flags for an Example App 170
9.4.2. Defined Debug Flags 170
9.5. C#-related Configuration Notes 171
9.5.1. Description of DCF setup code 171
9.5.2. Common services setup description: 171
10. Configuring DICOM features 173
10.1. Java and C# DICOM configuration 173
10.1.1. Example Session settings 173
10.2. C++ DICOM configuration 177
10.3. Customizing DicomDataDictionary 178
11. DICOM Image Compression 180
11.1. Lossy Compression Quality Issues & Concerns 180
11.2. JPEG Encoding Notes 180
11.2.1. JPEG Lossless (.57, .70) Encoding Notes 180
11.2.2. JPEG Lossy (.50, .51) encoding notes 182
11.2.3. JPEG 2000 Lossless (.90) encoding notes 183
11.2.4. JPEG 2000 Lossy (.91) encoding notes 183
11.3. JPEG Decoding Notes 185
11.3.1. Photometric Interpretation Problems 185
11.3.2. JPEG Lossless (.57, .70) Decoding Notes 185
11.3.3. JPEG Lossy (.50, .51) Decoding Notes 185
11.3.4. JPEG 2000 Lossless (.90) Decoding Notes 186
11.3.5. JPEG 2000 Lossy (.91) Decoding Notes 186
11.4. Using Aware, Inc JPEG Compression libraries 186
11.4.1. C# configuration 186
11.4.2. Java Configuration 186
11.4.3. C++ Configuration 187
12. I/O Statistics for Java and C# 188
13. Deploying a DCF-based application 189
13.1. Deployment Guidelines 189
13.2. License Key for a Deployed Application 189
13.2.1. OEM applications that ship with an installed license 189
13.2.2. OEM applications that do not ship with an installed license 189
13.2.3. OEM applications that use an activatable license 190
13.3. Limiting the DCF libraries required for Deployment 190
13.3.1. Find Application Dependencies - Windows 192
13.3.2. Find Application Dependencies - Linux 193
13.3.3. Find Application Dependencies - Java 193
13.4. Deploying standalone applications containing DCF code 194
13.4.1. Language Specific Standalone Installation 194
13.4.2. An Example Application Installer 194
13.5. DICOM Ports 196
Appendix A: Glossary of Terms 198
Appendix B: Bibliography 203
1. The DICOM Standard 203
2. Integrating the Healthcare Enterprise (IHE) 203
3. Sources for Compression Related Information 203
Appendix C: Storing Images from Print SCP 205
1. Collecting image and association info from Print_SCP 205
1.1. Preparation and Configuration Steps 205
1.2. Example Batch Script 206
1.3. Viewing Print SCP output via a web browser 206
Appendix D: Using DCF Dicom Filters 208
1. Fixing or working-around protocol problems 208

1.1. An application is sending an incorrect field in a DICOM print request 208
1.2. Modifying the DIMSE messages sent by a Java application 211
2. Using DicomTestFilter to support automated testing 212
3. Logging/Debugging DICOM Filter Effects 213
4. Using DCF DICOM Filters Overview 213
5. DCF DICOM Filter Configuration Overview 214
5.1. Sample configuration for the DicomElementFilter class. 215
5.2. Other Filter Types 216
5.3. Filter Configuration Files 216
5.3.1. Specifying a Sequence in a Configuration File 222
5.3.2. Using Macros to Specify Data 223
5.4. Example Filters 224
5.4.1. Example 1: Replacing a Value 224
5.4.2. Example 2: Removing an Element 224
5.4.3. Example 3: Modifying an Element’s Value with Regular Expressions 225
5.4.4. Example 4: Padding an Element’s Value 225
6. Developing Custom Filters 226
6.1. Custom Filters in Java 227
6.2. Custom Filters in C# 228
Appendix E: DCF MakeUID Function 229
1. Description of the DCF makeUID function 229
2. The function getUIDPrefix() returns the uid_prefix. 229
3. The function getUIDSuffix() returns a new UID suffix each time it is called. 230
Appendix F: Using Nunit tests with DCF .NET Applications 231
1. Example NUnit test class 231
2. Some Background on NUnit: 232
Appendix G: Using Perl with the DCF 233
Appendix H: Customizing the DCF Remote Service Interface 234
1. Shortcuts for Setting Debug Flags 234
1.1. Debug Shortcuts 234
1.2. Viewing the current settings 235
2. Example Debug Shortcut File 236
3. CDS Configuration Shortcuts 238
3.1. Shortcut Attributes 238
3.1.1. Types of shortcuts 239
3.1.1.1. title 239
3.1.1.2. html 239
3.1.1.3. display_only 239
3.1.1.4. integer or string 240
3.1.1.5. boolean 240
3.1.2. Optional attributes: 240
3.1.3. Synchronizing attributes: 241
3.2. Automatic Shortcut Generation 242
3.3. Example shortcut configuration file: 243
4. Authenticating Access to DCF Web Pages 245
5. Java Applet Issues 247
6. Other Applet Issues 249
6.1. Enabling Java 249
6.2. Firewall Issues 250
6.3. JavaScript Issues 251
Appendix I: Editing the Extended Data Dictionary 254
Appendix J: Product Licensing and Activation 258
1. Installing a new SDK License 258

2. Activating an SDK License 259
   2.1. SDK Network Activation Mode 260
   2.2. SDK Manual Activation Mode 261

3. Deploying OEM Applications with an Activatable License 264
   3.1. Developer Perspective 264
      3.1.1. Files to Distribute 265
   3.2. User Perspective 266
      3.2.1. Install the un-activated template license 266
      3.2.2. Create and Reserve a license 266
      3.2.3. Activate a license 267
         3.2.3.1. Network Activation Mode 268
         3.2.3.2. Manual Activation Mode 269
   3.3. Administrative Issues 273
      3.3.1. Reactivating a License 273
      3.3.2. Transferring a License 273
      3.3.3. Networking Issues Related to Network Activation 273

Table of Figures

Figure 1: Typical DCF-based DICOM server application....2
Figure 2: DCF Remote Service Interface Screen Example....20
Figure 3: Example – List of System Configurations to Start....21
Figure 4: Example – Partial List of Log Files....23
Figure 5: DCF Real-Time Log Screen Example....24
Figure 6: DCF Configuration Viewer Screen Example....25
Figure 7: DCF Documentation Screen Example....26
Figure 8: DCF Set Debug Flags Screen Example....27
Figure 9:Set Debug Flags Example – List of Running Processes....28
Figure 10: Set Debug Flags example – List of Components in Store SCP....28
Figure 11: Set Debug Flags Example – List of Debug Flags in the DCS Component....29
Figure 12: Modifying the Server Configuration....30
Figure 13: DCF Global Filter Set Editor Screen Example....31
Figure 14: Creating a DCF Library Component....158
Figure 15: Creating a DCF Application Component....159
Figure 16: Using a custom filter editor class....227
Figure 17: Displayed Debug shortcuts.....237
Figure 18: After Clicking the Update button.....238
Figure 19: Display results of applying a shortcut configuration file.....245
Figure 20: Example security warning from Firefox for a signed JAR file....248
Figure 21: Example security warning from Internet Explorer for a signed JAR file....248
Figure 22: Enabling Java in Firefox....249
Figure 23: Enabling Java in Internet Explorer....250
Figure 24: Setting firewall exceptions....251
Figure 25: Example JavaScript warning....251
Figure 26: Enabling JavaScript in Firefox....252
Figure 27: Enabling JavaScript in Internet Explorer....252
Figure 28: Adding your computer to the list of trusted sites....253
Figure 29: Editing the Extended Data Dictionary in GVIM....254
Figure 30: Editing the Extended Data Dictionary....255
Figure 31: Editing the Extended Data Dictionary as text....256
Figure 32: Select the “elements” group to add to the data dictionary, and the click “Add”.....256
Figure 33: Enter the new values for the data dictionary....256
Figure 34: Installing a new license....258
Figure 35: Warning to activate the license....259
Figure 36: Network mode for activating a license....260
Figure 37: Activation succeeded via Internet....260
Figure 38: Manual mode for activating a license....261
Figure 39: License activation web page....262
Figure 40: Web page showing license activation code....263
Figure 41: Successfully activated the license manually....263
Figure 42: Licensing Web Site Options....265
Figure 43: Reserving a license key....267
Figure 44: Network activation of a license....268
Figure 45: Manual activation of a license....269
Figure 46: License activation web page....270
Figure 47: Activation Code Display....271
Figure 48: Manual activation succeeded....272

Conventions

The following font conventions are followed throughout this document:

Bold

is used for definition of terms.

`Constant Width Italics`

is used for filenames, directory names, URLs, commands to runs, and developer actions.

Italics

is used for emphasis.

`Constant Width Large`

is used for DCF class names, component names, interfaces, or method names when they appear within this document's body text. Method names are shown with the usual () notation.

`Constant Width Small`

is used for code blocks or specific non-code file contents.

1. Overview

The DICOM Connectivity Framework – or **DCF** – is a software product that enables a medical imaging system – printer, scanner, modality, archive device, workstation, etc. – to communicate with other devices over a network, using the DICOM version 3 protocol. By using the DCF, an OEM can provide DICOM connectivity for their device with a minimum of effort. The DCF provides much more than DICOM libraries – Common Services Components allow an OEM to produce commercial-quality applications without having to reinvent such facilities as logging, configuration data management, and process control. The DCF development environment allows developers to focus on adding value to their products and getting to market quickly.

The DICOM Connectivity Framework supports modular programming and mixed language development environments. In most large-scale software development efforts, there are multiple languages or technologies involved, possibly multiple target platforms, and multiple developers working on different subsystems. Managing these processes and technologies is at best time-consuming and tedious – at worst it can be a nightmare. The DCF provides technology to simplify these complex software development tasks.

The DCF can be used in two distinct ways. First, where local conventions, file system layout, build processes, and such are well established, DCF applications and libraries can be used like any other software tool product: point your makefiles and/or IDE settings and various environment variables to the DCF library, include, and binary directories, and go. The second approach uses the DCF to help organize the structure of applications and automate much of the process. The DCF will generate makefiles, Windows Visual Studio project files, debug/trace instrumentation for your code, configuration files, and API documentation. The DCF encourages the re-use of both software AND the processes used to create and manage that software. The DCF performs all of these tasks using Laurel Bridge Software proprietary technologies, along with best-of-breed commercial or open source software products.

The DCF can be used to create DICOM solutions in the Microsoft .NET environment, using C# or other .NET programming languages.

1.1. System Architecture Overview

The DCF is a modular system, made up of various software components. These software components are combined with the OEM's software on a particular computer platform to form a system. DCF components are tested by Laurel Bridge in the context of various systems. Obviously, an OEM will be defining his own system configuration. The OEM system may resemble one of the DCF sample systems, or it may be something altogether different.

A **component** is a reusable, independently built, unit of code.

A **platform** is the computing environment on which DCF components run. This includes hardware, operating system, and any third-party software that must be present on the system.

A **system** is the combination of one platform and some number of software components.

1.2. An Example DCF Application

The following diagram shows the structure of a typical DCF application (a DICOM storage service class provider).

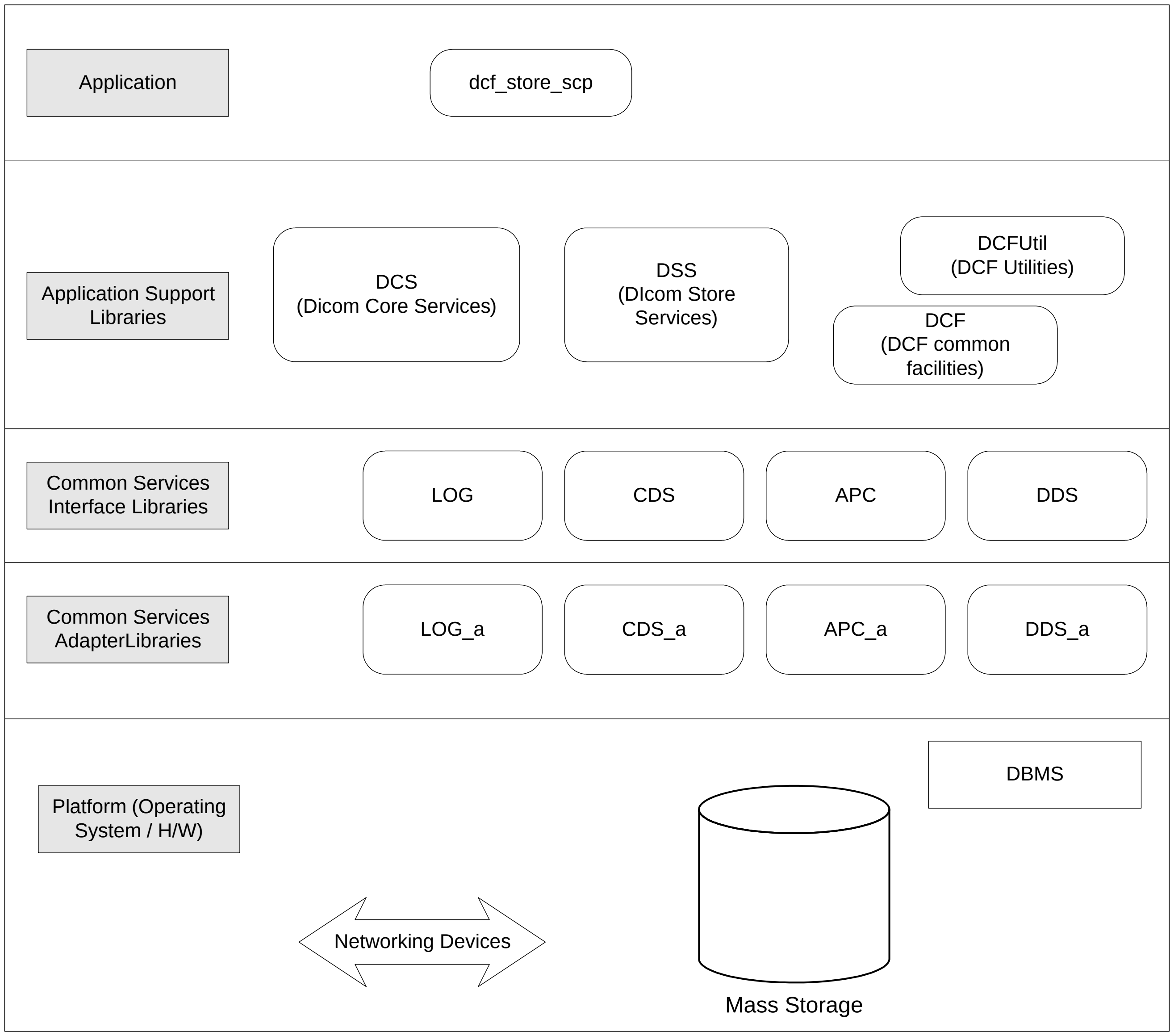

Application
dcf_store_scp
Application Support Libraries
DCS (Dicom Core Services)
DSS (DIcom Store Services)
DCFUtil (DCF Utilities)
DCF (DCF common facilities)
Common Services Interface Libraries
LOG
CDS
APC
DDS
Common Services AdapterLibraries
LOG_a
CDS_a
APC_a
DDS_a
DBMS
Platform (Operating System / H/W)
Networking Devices
Mass Storage

Figure 1: Typical DCF-based DICOM server application

The diagram shows the division of DCF libraries into layers. The top layer “application” contains the “dcf_store_scp” application component. This is the entry point for the application. During initialization, dcf_store_scp installs the selected Common Services Adapters, and then uses objects in the DCS and DSS libraries to perform the real work of implementing the Storage SCP.

The libraries in the “application support” layer perform the real DICOM work. The DCS library provides the basic classes for DICOM attributes or elements, data sets, PDUs, DIMSE messages, as well as higher level classes such as `AssociationManager`, `DicomAssociation`, for easily managing multiple concurrent associations, `DicomDataDictionary`, numerous classes for network and file I/O and Data set/Image filtering.

The DSS library provides the `StoreSCP` class which registers as the handler for all Storage presentation contexts. C-Store-Request messages are dispatched to `StoreSCP`.

The “common services interface” layer provides abstract interfaces for services such as logging, application configuration, process control, and data storage/retrieval. `StoreSCP` invokes the `storeObject()` method on the DDS (DICOM Data Service) interface to locally store the dataset contained in the C-Store-Request message.

The “common services adapter” layer provides concrete implementations of common services. The LOG adapter (LOG_a) library writes log messages to either the file system or the LOG Server, via the DLOG (Distributed LOG) CORBA interface. The Configuration Data Service adapter (CDS_a) accesses configuration data from either the file system or the DCDS_Server, using the DCDS (Distributed CDS) CORBA interface. The DICOM Data Service adapter might store image or SOP instance data to the file system, and may store header data in an SQL database system. An OEM that wishes to customize the behavior of the store_scp for example to use a locally defined schema needs only implement one function: “`storeObject()`”. The developer can opt to hide all handling of concurrent network associations, DIMSE messages, and other complex DICOM details. See the section “Writing a Customized Storage SCP” (Section 4.5.1) for an example of extending the `DicomDataService` implementation.

1.3. Components

The DCF is designed as a collection of components. Components in the DCF are rather large-grained, along the lines of "package" in the UML terminology, or "subsystem" in certain other models. This is as opposed to some systems where "component" is synonymous with "class" or "object" which would lead to a more fine-grained component architecture.

A DCF Component that is used directly by Microsoft.NET clients is delivered as a binary "Assembly", along with associated configuration data.

1.3.1. Component Classification

Each component or part of a DCF based software system is one of the supported component physical types (see 1.2.1.2). All components of a given type share a common structure. Once you have determined what type of component you need, there is no need to waste time deciding how to manage the runtime or build configuration information, how to instrument the component, how to arrange the source files for that component, etc. API documentation for components can be automatically generated. Functional or technical specifications, test plans, and other important documents can be organized along component boundaries, providing additional structure and consistency to your development processes. DCF components may also be classified in terms of their category or functional type.

1.3.1.1. DCF Component Categories

Type	**Description**
DICOM Applications	Executable programs which function as a DICOM SCU or SCP, plus various other DICOM utilities
Applications	DCF support applications and other executable programs
Common Services Interface Libraries	Interfaces to horizontal services
Common Services Adapter Libraries	Implementations of horizontal services
Application Support Libraries	Interfaces and implementations of vertical services
IDL Interface Libraries	CORBA interfaces (Java and C++ only)
Miscellaneous	Scripts and other items
Examples	Library or Application components used to demonstrate use of the DCF
Tests	Components used to test other DCF components

1.3.1.2. DCF Component Physical Types

Type	**Description**
cpp_lib	C++ shared library Unix: a .so file, and accompanying header (.h) files Windows: a .dll file, .lib file and accompanying .h files
cpp_lib_src	C++ shared library source Source code and header files that are logically separate, but compiled (and in the case of Windows DLL, linked) together with other cpp_lib_src components to form a cpp_lib_pkg. See cpp_lib_pkg.
cpp_lib_pkg	C++ shared library package

	Unix: a .so file, and accompanying header (.h) files Windows: a .dll file, .lib file and accompanying .h files A cpp_lib_pkg combines multiple cpp_lib_src components into a single buildable component. Components that have circular dependencies can be defined separately (cpp_lib_src) but built as one (cpp_lib_pkg). This becomes an issue only when building a Windows dll that must "link" with other libraries, resolving any external symbol references.
java_lib	Java shared library A DCF Java shared library corresponds to a Java package. The primary deliverable is a directory of .class files, or a .jar
cs_lib	C# library A DCF C# library corresponds to a .NET assembly. The primary deliverable is a dll file. DCF .Net assembly dll's are registered in the global assembly cache by the installation program. You can register assemblies manually using the "gacutil" application.
cpp_app	C++ application An application or binary program. Every application has a default configuration file (which will ultimately be represented as a CFGGroup object). This application configuration file contains a copy of the component configuration file from each C++ library component used by the application. Application names must be unique, i.e., a C++ app and a Java app cannot have the same name.
cpp_jni_lib	C++ JNI library - Java native invocation lib
cpp_com_lib	C++ dll that implements one or more COM interfaces. Similar to VS ATL dll project
cpp_ipc_app	C++ application that implements a COM interface in a server proc., and also supports CORBA communications.
java_app	Java application A Java executable is like a Java shared library, except that one of the classes has a "main()" function, and that there is an application configuration file, like a C++ application.
cs_app	C# application
cs_win_app	C# Windows GUI application
idl_lib	IDL interface An IDL interface corresponds to an IDL module. The result of building an IDL interface is C++ shared library, and associated header files, and a Java shared library.

1.3.2. Selected DCF Components Organized by Category

1.3.2.1. DICOM Applications

Name	Component Type	Description
dcf_analyze	cpp_app	Display DICOM file information - creates pixel data value histogram, and other statistics
dcf_create_fileset	cpp_app	Create a DICOMDIR index file from a collection of DICOM files
dcf_dump	cpp_app	Print the contents of a DICOM file to the console - automatically detects file encoding
dcf_echo_scp	cpp_app	DICOM Verification Service Class Provider - used primarily for testing
dcf_echo_scu	cpp_app	DICOM Verification Service Class User - used to ping a DICOM server
dcf_filter	cpp_app	Apply filtering operations to a DICOM file, creating a modified file
dcf_mod	cpp_app	Perform modifications to a DICOM file, creating a new DICOM file

Name	Component Type	Description
dcf_mpps_scu	cpp_app	DICOM Modality Performed Procedure Step Service Class User - used to send an MPPS object to a server
dcf_mwl_scp	cpp_app	DICOM Modality Worklist/Performed Procedure Step Service Class Provider
dcf_mwl_scu	cpp_app	DICOM Modality Worklist Service Class User - used to retrieve a list of worklist items from a server
dcf_pg	cpp_app	DICOM Test Pattern Generator - creates images or other DICOM file types
dcf_print_scp	cpp_app	DICOM Print Service Class Provider
dcf_print_scu	cpp_app	DICOM Print Service Class User - used to send sheets of images to a server
dcf_qr_scp	cpp_app	DICOM Query Retrieve Service Class Provider
dcf_qr_scu	cpp_app	DICOM Query Retrieve Service Class User - used to query or move sets of objects from a server
dcf_store_scp	cpp_app	DICOM Storage Service Class Provider
dcf_store_scu	cpp_app	DICOM Storage Service Class User - used to send objects to server for storage
dcf_storecommit_scu	cpp_app	DICOM Storage Commitment Service Class User - used to request commitment for long term storage of objects
dcm2jpeg	cpp_app	Convert DICOM images to JPEG or JPEG2000
jdcf_ImageViewer	java_app	Java app for viewing DICOM images

1.3.2.2. Non DICOM Applications

Name	Component Type	Description
apc_client	cpp_app	Command line client for Application Control
cds_cgi	cpp_app	CGI program to set attributes in CDS via shortcuts
cds_client	cpp_app	Command line client for Configuration Data Service
dcf_info	cpp_app	Display DCF version and system information
hex_dump	cpp_app	Formatted hexadecimal display for files
sdfcgi	cpp_app	Set debug flags WWW cgi-bin program
sdfgroup_cgi	cpp_app	Set debug flags as groups - WWW cgi-bin program for DCF
dcf_sysmgr	cpp_ipc_app	DCF System Manager
NDCDS_Server	cs_app	C# Configuration DataBase Server
CfgEdit	java_app	Configuration program for Java CDS/CDS_a components
DCDS_Server	java_app	Distributed Config Data Service - Java server for CDS data storage/retrieval
DLOG_Server	java_app	Java server implementation of DLOG interface, for centralized logging facilities
FilterEditor	java_app	Editor for filters used by the DCF
FilterSetEditor	java_app	Editor for filter sets used by the DCF
Log_viewer	java_app	Java viewer of DLOG_Server output
PrinterServer	java_app	Java Printer Server application
Std_applets	java_app	Standard configuration information for all applets to use

1.3.2.3. Examples

Name	Component Type	Description
ex_cds	cpp_app	Example program that uses DCF common services - CDS in particular
ex_combo	cpp_app	Example program that uses a combination of DCF API's for DICOM image creation, storage, and printing
ex_create_file	cpp_app	Example program of using DCS component for creating a DICOM file
ex_dump_dicom_dir	cpp_app	Example program that dumps the contents of DicomDir.
ex_file_access_1	cpp_app	Example program use of API's to access data in a DICOM file
ex_hello_world	cpp_app	Example program that uses DCF common services to implement the classic first application
ex_iod	cpp_app	Example program that uses IOD library component to access Information Object Descriptions
ex_notify	cpp_app	Example program that uses CDS services to create a simple distributed alarm delivery system.
ex_oemlog	cpp_app	Unit test for DCF LOG interface and OEMLOG_a custom adapter example implementation
ex_print_element_value	cpp_app	Example program showing very basic use of API's to access data in a DICOM file
ex_server	cpp_app	Example server program that uses DCF common services
ex_verification_scu	cpp_app	Example program that uses the DCS::VerificationSCU class
OEMLOG_a	cpp_lib	OEM LOG adapter - example custom implementation of LOG interface
ex_ndcf_create_fileset	cs_app	Create a DICOMDIR index file from a collection of DICOM files
ex_ndcf_dump	cs_app	C# example app that reads a DICOM file and displays to console
ex_ndcf_filter	cs_app	Example using DCF filter classes to modify a DICOM file
ex_ndcf_HelloWorld	cs_app	C# example app logs.
ex_ndcf_ModPixelData	cs_app	C# example app that reads a DICOM file, modifies the pixel data, and creates a new file.
ex_ndcf_mwl_scu	cs_app	C# DICOM MWL Client example.
ex_ndcf_simple_query_scu	cs_app	C# DICOM Query Retrieve or MWL Client example.
ex_ndump_dicom_dir	cs_app	C# example app that reads a DICOMDIR file and displays to console
ex_necho_scu	cs_app	CS client app that demonstrates use of VerificationClient and DicomSCU in DCS lib
ex_nmpps_scu	cs_app	MPPS Example with source code.
ex_nmwl_scp	cs_app	CS server app that demonstrates use of QRServer and DicomSCP in DCS lib
ex_nmwl_scu	cs_app	MWL Example with source code.
ex_nprint_client	cs_app	C# client app that demonstrates use of PrintClient and DPS lib
ex_nprint_element_value	cs_app	C# example app that reads a DICOM file and displays one element
ex_nprint_scu	cs_app	CS client app that demonstrates use of PrintClient and DPS lib

Name	**Component Type**	**Description**
ex_nqr_scp	cs_app	CS server app that demonstrates use of QRServer and DicomSCP in DCS lib
ex_nqr_scu	cs_app	Query Example with source code.
ex_nstorecommit_scp	cs_app	CS server app that demonstrates use of StoreServer and DicomSCP in DCS lib
ex_nstorecommit_scu	cs_app	cs client for Storage Commitment SOP class
ex_nstore_client	cs_app	cs client for StoreClient and StoreSCU classes
ex_nstore_scp	cs_app	CS server app that demonstrates use of StoreServer and DicomSCP in DCS lib
ncds_client	cs_app	Command line client for Configuration Data Service
OEMLOG_a	cs_lib	LOG Client Adapter - C# example implementation of DCF.LOGClient interface
ex_ndcf_CFGObserver	cs_win_app	CDS observer example application.
ex_ndcf_echo_scu	cs_win_app	C# DICOM Echo Client example.
ex_ndcf_ImageViewer	cs_win_app	C# example image viewer uses DCS library
ex_ndcf_query_scu	cs_win_app	C# DICOM Query Retrieve or MWL Client example.
ex_jdcf_create_fileset	java_app	Create a DICOMDIR index file from a collection of DICOM files
ex_jdcf_dcm2jai	java_app	Java example app that converts DICOM files to one of the supported JAI file types
ex_jdcf_dcmview	java_app	Java example app that displays a DICOM image using the JAI API’s
ex_jdcf_dump	java_app	Java example app that prints the contents of a DICOM file to the console
ex_jdcf_filter	java_app	Java app that reads a DICOM file, applies filters and writes a new file
ex_jdcf_HelloWorld	java_app	Java test application for DCF common services
ex_jdcf_jai2dcm	java_app	Java example app that one of the supported JAI file types to DICOM
ex_jdump_dicom_dir	java_app	Java example app that prints the contents of a DICOMDIR file to the console.
ex_jecho_scu	java_app	Java client app that demonstrates use of VerificationClient and DicomSCU in DCS lib
ex_jecho_scu_gui	java_app	Java client app that demonstrates use of VerificationSCU in DCS lib
ex_jModPixelData	java_app	Java example app that reads a DICOM file, modifies the pixel data, and creates a new file.
ex_jmpps_scu	java_app	MPPS Example with source code.
ex_jmwl_client	java_app	Example program showing use of DCS API’s to implement MWL client
ex_jmwl_scp	java_app	Java server app that demonstrates use of QRServer and DicomSCP in DCS lib
ex_jmwl_scu	java_app	MWL Example with source code.
ex_jnotify	java_app	Java test application for DCF common services
ex_jprint_client	java_app	Java client app that demonstrates use of PrintClient and DPS lib

Name	Component Type	Description
ex_jprint_element_value	java_app	Example program showing very basic use of API's to access data in a DICOM file
ex_jprint_scu	java_app	Java client app that demonstrates use of PrintClient and DPS lib
ex_jqr_scp	java_app	Java server app that demonstrates use of QRServer and DicomSCP in DCS lib
ex_jqr_scu	java_app	Query Example with source code.
ex_jquery_scu	java_app	Java DICOM Query Retrieve or MWL Client example
ex_jstorecommit_scp	java_app	Java server app that demonstrates use of StoreCommitServer and StoreCommitSCP
ex_jstorecommit_scu	java_app	java example SCU for Storage Commitment SOP class
ex_jstore_client	java_app	Java client app that demonstrates use of StoreClient and DicomSCU in DCS lib
ex_jstore_scp	java_app	Java server app that demonstrates use of StoreServer and StoreSCP
ex_jstore_scu	java_app	Java client app that demonstrates use of StoreClient and DicomSCU in DCS lib
jcds_client	java_app	Command line client for Configuration Data Service
OEMLOG_a	java_lib	LOG Adapter - java example implementation of LOG interface

1.3.2.4. Common Service Interface Libs

Name	Component Type	Description
APC	cpp_lib_src	Application Control - C++ interface to startup, shutdown, event handling, and application configuration services - APC,LOG,CDS combine into DCF_DCFCore library
CDS	cpp_lib_src	Configuration Data Service - C++ interface to hierarchical database for application settings - APC,LOG,CDS combine into DCF_DCFCore library
DDS	cpp_lib	DICOM Data Service - C++ interface to store/retrieve/search mass storage services for DICOM objects
LOG	cpp_lib_src	LOG - C++ interface to logging services - APC,LOG,CDS combine into DCF_DCFCore library
DCF	cs_lib	DICOM Connectivity Framework Common Services (also contains CDS, APC, and LOG interfaces)
APC	java_lib	Application Control - java interface to startup, shutdown, event handling, and application configuration services
CDS	java_lib	Configuration Data Service - java interface to hierarchical database for application settings

1.3.2.5. Common Service Implementation Libs

Name	Component Type	Description
APC_a	cpp_lib	Application Control Adapter - C++ reference implementation of APC interface
CDS_a	cpp_lib	Configuration Data Service Adapter - C++ reference implementation of CDS interface
DDS_a	cpp_lib	DICOM Data Service Adapter - C++ reference implementation of DDS interface

LOG_a	cpp_lib	LOG Adapter - C++ reference implementation of LOG interface
APC_a	cs_lib	Application Control Adapter - C# reference implementation of DCF.AppControl interface
CDS_a	cs_lib	Configuration Data Service Adapter - cs reference implementation of CDS interface
DDS_a	cs_lib	DICOM Data Service cs library - C# interface to store/retrieve/search mass store services for DICOM objects
LOG_a	cs_lib	LOG Client Adapter - C# reference implementation of DCF.LOGClient interface
APC_a	java_lib	Application Control Adapter - java reference implementation of APC interface
CDS_a	java_lib	Configuration Data Service Adapter - java reference implementation of CDS interface
DDS_a	java_lib	DICOM Data Service Adapter - Java reference implementation of DDS interface
LOG_a	java_lib	LOG Adapter - java reference implementation of LOG interface

1.3.2.6. Application Support Libs

Name	**Component Type**	**Description**
com_lbs_LOG_a_SyslogOutput_a	cpp_jni_lib	JNI (Java Native Interface) implementation of Unix syslog LOGOutput.
com_lbs_DCS_DicomTSCWCodec	cpp_jni_lib	JNI implementation for accessing JPEG compression libraries
boost_regex	cpp_lib	copy of boost regular expression source
DCFUtil	cpp_lib	DICOM Connectivity Framework - Utilities
DCF_gui	cpp_lib	Common look and feel elements for C++ CGIs
DCS	cpp_lib	DICOM Core Services - classes for accessing DICOM data and communicating with DICOM devices
DIS	cpp_lib	DICOM Information Services - classes for HIS/RIS integration - Modality Worklist, MPPS, etc.
DPS	cpp_lib	DICOM Print Services - classes for DICOM Print clients and servers
DSS	cpp_lib	DICOM Store Services - classes for DICOM Store, Query Retrieve, Storage Commit, DICOMDIR processing
IOD	cpp_lib	Information Object Description - auto-generated DICOM data set wrapper classes
ljpeg12	cpp_lib	IJG JPEG lib with lossless patch applied (12 bit build)
ljpeg16	cpp_lib	IJG JPEG lib with lossless patch applied (16 bit build)
ljpeg8	cpp_lib	IJG JPEG lib with lossless patch applied (8 bit build)
TSCW	cpp_lib	Transfer Syntax Codec Wrapper
TSCWAware	cpp_lib	Plugin to Aware, Inc. JPEG codec
TSCWIJG	cpp_lib	Plugin for IJG JPEG codec
TSCWJasper	cpp_lib	Plugin for JasPer JPEG codec
DCFCore	cpp_lib_pkg	DICOM Connectivity Framework Core interfaces library package - contains DCF, LOG, CDS, and APC library components
DCF	cpp_lib_src	DICOM Connectivity Framework - common modules
DCS	cs_lib	DICOM Core Services C# class library
DCS	cs_lib	DICOM Core Services C# class library

Name	Component Type	Description
DDS	cs_lib	DICOM Data Service library - C# interface to store/retrieve/search mass store services for DICOM objects
DIS	cs_lib	DICOM Information Services C# class library
DPS	cs_lib	DICOM Print Services C# class library
DSS	cs_lib	DICOM Store Services C# class library
IJGCodec	cs_lib	DICOM transfer syntax codec using IJG JPEG library
IOD	cs_lib	Generated IOD wrappers.
NDCDS	cs_lib	Configuration Data Service Adapter - cs reference implementation of CDS interface
DCF	java_lib	DCF commonly used classes
DCS	java_lib	DICOM Core Services Java library
DCS	java_lib	DICOM Core Services Java library
DDS	java_lib	DICOM Data Service Java library - Java interface to store/retrieve/search mass store services for DICOM objects
DIS	java_lib	DICOM Information Services Java class library
DPS	java_lib	DICOM Print Services Java class library
DSS	java_lib	DICOM Storage Services Java class library
GUI_helper	java_lib	Java GUI helper classes
IOD	java_lib	Information Object Description - auto-generated DICOM data set wrapper classes
OEMPrinter	java_lib	OEM Printer - reference implementation of OEM Java printer code

1.3.2.7. IDL Interface Libs

Name	Component Type	Description
DAPC	idl_lib	Distributed Application Control CORBA interface
DCDS	idl_lib	Distributed Configuration Data Service CORBA interface
DDCS	idl_lib	Distributed DICOM Core Services CORBA interface
DDPS	idl_lib	Distributed DICOM Print Services CORBA interface
DLOG	idl_lib	Distributed Log CORBA interface

1.4. Platforms

The DCF defines a platform as the computer hardware, operating system, and any third-party tools or programs that have been added to that system.

The currently supported platforms for the DCF and a brief description of each are shown in the following table.

DCF code is designed for portability. Contact Laurel Bridge Software for information regarding additional platform support.

Platform Name	**Operating System(s)**	**Minimal OS Version(s)**	**Processor**	**Description**
Linux_13_x86_gcc_493	Suse	13.x	x86	Intel X86 PC compatible Suse Linux 13.x GNU C++ 4.9.3 Java SE JDK
Linux_13_x64_gcc_493	Suse	13.x	x64	AMD 64 PC compatible Suse Linux 13.x GNU C++ 4.9.3 Java SE JDK
Centos_65_x64_gcc_482	Centos	6.5	x64	AMD 64 PC compatible Centos Linux 6.x GNU C++ 4.8.2 Java SE JDK
Windows_NT_5_x86_VisualStudio12.x	Windows 10 Server 2008+	Windows 10	x86	Intel X86 PC compatible Visual Studio 2013 Microsoft.NET2.0+ Java SE JDK
Windows_NT_5_x64_VisualStudio12.x	Windows 10 Server 2008+	Windows 10	AMD64	AMD 64 PC compatible Visual Studio 2013 Microsoft.NET2.0+ Java SE JDK
Windows_x86_VisualStudio16.x	Windows 10 Server 2008+	Windows 10	x86	Intel X86 PC compatible Visual Studio 2019 Microsoft.NET2.0+ Java SE JDK
Windows_x64_VisualStudio16.x	Windows 10 Server 2008+	Windows 10	AMD64	AMD 64 PC compatible Visual Studio 2019 Microsoft.NET2.0+ Java SE JDK

Note that the DCF uses Java 6 Update 32 (sometimes known as “1.6.0_32”) or newer updates. Java versions older than Java 6 are not supported, while versions newer than Java 6 (e.g., Java 7, Java 8) are not routinely tested but have been used in production environments. The Java SDK may be downloaded from http://www.java.com/. Windows 10 and Server configurations assume current updates applied.

1.5. Systems

In practice, it is the OEM or system integrator who decides what a system is. For the purposes of DCF development, we define a system as a specific platform running a specific set of application components with their associated configuration data.

Giving a name to this particular combination of hardware, off-the-shelf software, DCF, and OEM software provides a structure, which can be useful for defining tests. This structure helps during the debugging, testing, and potentially field support phases, providing a more concise way of answering common questions like: “What did the computer that *xyz* happened on look like?” or “What else was running at the time that *abc* application failed?” or “Did we ever try the combination of applications *x* and *y* with operating system *z*?”

The following table lists examples of systems that are defined for internal DCF testing at Laurel Bridge:

Name	**Description**
all_servers_unix	All DICOM servers running on Unix platform
all_servers_win32	All DICOM servers running on Win32 platform
dcds_server_unix	Simple DCDS Server only configuration running on Unix platform
dcds_server_win32	Simple DCDS Server only configuration running on Win32 platform.
dcf_switch_unix	DICOM Switch configuration with echo and store SCP’s running on Unix platform
dcf_switch_win32	DICOM Switch configuration with echo and store SCP’s running on Win32 platform
jmwl_server_unix	Java Modality Worklist server configuration running on Unix platform
jmwl_server_win32	Java Modality Worklist server configuration running on Win32 platform
jqr_server_unix	Java QR server configuration running on Unix platform
jqr_server_win32	Java QR server configuration running on Win32 platform
jstorecommit_scu_agent_unix	Java Storage Commitment SCU Agent configuration running on UNIX platform
jstorecommit_scu_agent_win32	Java Storage Commitment SCU Agent configuration running on Win32 platform
jstorecommit_server_unix	Java Storage Commitment Server configuration running on Unix platform
jstorecommit_server_win32	Java Storage Commitment server configuration running on Win32 platform
jstore_server_unix	Java Store server configuration running on Unix platform
jstore_server_win32	Java Store server configuration running on Win32 platform
mwl_server_unix	Modality Worklist server configuration running on Unix platform
mwl_server_win32	Modality Worklist server configuration running on Win32 platform
ndcds_server_win32	Simple DCDS Server only configuration running on Win32 platform.
nmwl_server_win32	C# Modality Worklist server configuration running on Win32 platform
nqr_server_win32	C# QR server configuration running on Win32 platform
nstorecommit_scu_agent_win32	C# Storage Commitment SCU Agent configuration running on Win32 platform
nstorecommit_server_win32	C# Storage Commitment Server with C# Storage Commitment SCU agent configuration running on Win32 platform
nstore_server_win32	C# Store server configuration running on Win32 platform
only_jmwl_server_unix	Java Modality Worklist server only configuration running on Unix platform
only_jmwl_server_win32	Java Modality Worklist server-only configuration running on Win32 platform
only_jqr_server_unix	Java QR server only configuration running on Unix platform
only_jqr_server_win32	Java QR server-only configuration running on Win32 platform

Name	Description
only_jstore_server_unix	Java Store server only configuration running on Unix platform
only_jstore_server_win32	Java Store server-only configuration running on Win32 platform
only_nstore_server_win32	C# Store server only configuration running on Win32 platform
print_server_unix	Print server configuration running on Unix platform
print_server_win32	Print server configuration running on Win32 platform
qr_server_unix	Query Retrieve server configuration running on Unix platform
qr_server_win32	Query Retrieve server configuration running on Win32 platform
store_server_unix	Store server configuration running on Unix platform
store_server_win32	Store server configuration running on Win32 platform

2. Installing the DCF

Follow the installation instructions that are located in the release notes file included on the distribution CD; choose the appropriate instructions for your platform.

Note: If you are installing on Vista, Windows 7 or Windows 10, you may need to disable the UAC (User Access Controls) in order for the DCF to be installed correctly.

2.1. Multi-user vs. Single-user Installation

The DCF is designed to allow multiple developers to share a single server or host computer. Sharing a host is not particularly difficult for users of some applications such as compilers or client-side programs. For developers trying to perform the entire cycle of develop, deploy, execute/test, and debug for an entire suite of client and server applications, sharing a host can be a serious challenge. Files are installed into either shared or per-user directories. The shared files are installed once, under the directory given by the DCF_ROOT environment variable.

The DCF provides each developer a sandbox or virtual server environment that is isolated from all other developers. The primary resources that must be isolated are files and TCP ports. Files that are private to a developer are located under the directory given by the DCF_USER_ROOT environment variable.

By setting environment variables appropriately all files can be installed under a single directory for a simpler, single-user configuration. For Windows developers this single-user installation is much more common.

TCP port numbers for one DICOM server and the Apache web server are automatically created for each user. On UNIX hosts, the Apache web server port number is 1000 + user id. The DICOM port is 2000 + user id.

For example, if your UNIX user id is 1004 then your private web server will listen on port 2004. The first DICOM server (SCP) application that you run will listen on port 3004.

Of course, these port numbers are all specified in configuration files and can be manually set if needed. If you need to run more than one DICOM SCP process concurrently, you can use the `$DCF_FUNC{euid_plus, n }` macro to automatically generate additional port numbers based on the current user's effective id.

2.2. DCF Shared Files

The shared directory (DCF_ROOT), e.g., */opt/DCF-X.X.X*, or a similar location contains the following subdirectories:

Installed Directory	Description
./bin	Pre-built applications and scripts
./classes	Pre-built Java class files
./lib	Pre-built C++ shared libraries and .NET assembly dll's
./lib/perl5	Perl modules used by various scripts
./doc	
./doc/applications	Documentation for applications
./doc/components	

Installed Directory	Description
./doc/components/java	Generated documentation for Java components
./doc/components/cc	Generated documentation for C++ components
./doc/components/cs	Generated documentation for C#.NET components
./doc/components/idl	Generated documentation for IDL components (CORBA interfaces)
./doc/userguides	Miscellaneous technical documents
./test	Top of directory used for system test scripts and data
./test/cds	Scripts and data files used to test CDS
./test/db	Scripts and data files used to configure a sample radiology database using postgreSQL.
./test/print	Scripts and data files used to test DICOM print SCP and SCU
./test/qr	Scripts and data files used to test DICOM query retrieve SCP and SCU
./test/store	Scripts and data files used to test DICOM store SCP and SCU
./test/worklist	Scripts and data files used to test Modality Worklist, and Modality Performed Procedure Step SCP and SCU
./include	C++ include files. Each C++ library component listed above has an include directory of the same name.
./oem	Files that are extracted during per-user install
./devel	Files used during development
./devel/lib	Miscellaneous files to support development tools
./devel/lib/common	Common templates for makefiles and other files generated by dcfmake.pl

2.3. DCF Per-user Files

A per-user directory (DCF_USER_ROOT), e.g., `/home/demo/DCF`, or a similar location, contains the following subdirectories:

User Directory	Description
./devel	Top of development tree
./devel/cfgsrc	Configuration source – these are configuration files that are not generated
./devel/cfggen	Configuration files that are generated by dcfmake.pl or update_cds.pl
./devel/cfggen/apps	Application configuration files

User Directory	Description
./devel/cfggen/apps/defaults	Default application config files, generated by dcfmake.pl when C++ or Java applications are built
./devel/cfggen/components	Component configuration files – this is information about library or other components that does not vary depending on the application that uses the component. There is a subdirectory below for each type of component.
./devel/lib	Miscellaneous files to support development tools
./devel/csrc	C++ source code
./devel/cssrc	C#.Net source code
./devel/jsrc	Java source code
./devel/isrc	Idl source code
./bin	Contains binaries that are built by the user
./cfg	Configuration data that is used at runtime. This data is accessed by applications through the CDS interface. This directory is populated initially by the update_cds.pl utility, which combines files from devel/cfgsrc and devel/cfggen.
./cfg/apps	Application configurations
./cfg/components	Component configuration – static information about components – i.e., not dependent on application context
./cfg/dicom	Miscellaneous DICOM-related configuration files
./cfg/procs	Processes – This directory contains files each of which represents an active process or DCF application. When an application starts, a working copy of the application configuration (e.g., ./cfg/apps/defaults/store_server) is made. This copy is called the “application instance configuration”. (For process id 123, for example, this is: `./cfg/procs/store_server.123`.) Changes can be made to data in this object, which may immediately affect the associated process. This is how debug-flags are updated on a running application. Applications are not necessarily observing all data in their instance configuration. Applications can observe data anywhere in the CDS repository, not just /procs. (see notify_example)
./cfg/systems	Definitions of system configurations - used to start up different combinations of processes for testing
./cfg/test	Miscellaneous data used in testing
./cfg/tmp	Miscellaneous temporary data
./tmp	Temporary files created/used by DCF

User Directory	Description
./tmp/log	Default location of log files
./tmp/scp_images	Images sent to print_server and store_server go here by default.
./tmp/job_images	Images referenced by print jobs go here by default
./classes	Contains Java classes that are built by the user
./lib	Contains C++ libraries (.so files) that are built by the user
./include	Contains include files used or created by the user.
./httpd	(may be in another location – e.g., /home/demo/httpd) The per-user Apache web server uses this directory.

2.4. The DCF Remote Service Interface

The DCF provides a web browser interface for service and testing tools. You can use this interface in your development or target environment, as is, or customize it to integrate with your own product's web service interface. All of the DCF web-based utilities are optional and *are not required* to implement DICOM SCU or SCP applications. These web utilities can provide an easy way for developers or field service engineers to view or modify your application's configuration and operation and to aid them in diagnosing problems that may occur – all that is needed is to run a web server and provide the web utility scripts.

2.4.1. Running the Apache Web Server

If your system is not running a web server or if you have not opted to connect the DCF web pages and cgi-bin executables into your system's default web server environment, you should run the Apache web server that has been configured for use with the DCF.

When the DCF was installed an Apache web server configuration was created. These files are in the directory given by the environment variable DCF_HTTPD_ROOT. Each user on a multi-user development host can have an independent web server. Start this server by typing the command

```
perl –S run_apache.pl
```

(See Appendix G: Using Perl with the DCF for information about invoking the Perl interpreter.)

2.4.1.1. Alternate Web Servers

It is possible to use the DCF with web servers other than Apache 2.2.16, such as Microsoft IIS or Tomcat, but the DCF does not automatically come configured to use different web servers. If you want to use a different web server, you have to configure your web server of choice to serve up the DCF's web pages and CGI scripts, and you may need to port the CGI scripts (which are written in Perl) to a scripting language that your web server supports.

The configurations for the various web servers are too different to be described here, but the basic steps are as follow:

1. Set the web server's document root to point to the DCF's web pages.
2. Configure the web server to parse the DCF's CGI scripts. To use the existing Perl scripts, you should configure the script interpreter to allow it to run Perl scripts. Alternatively, you can translate the scripts into the scripting language of your choice and let your web server parse and serve them up that way.
3. You may need to configure your server so it can run the CGI scripts in the DCF's CGI-BIN directory; for example, Apache uses the "ScriptAlias" directive to indicate the directory where scripts are located.
4. You may wish to modify the Perl scripts "`run_apache.pl`" and "`kill_apache.pl`" to start and stop your web server, if you wish to use those scripts in your operation of the DCF.

The primary files of interest are as follows:

- The static HTML pages for the DCF are located in the directory $DCF_ROOT/httpd/html, for example, "`C:\Program Files\DCF\httpd\html`".
- The CGI scripts are located in $DCF_ROOT/httpd/cgi-bin, e.g., "`C:\Program Files\DCF\httpd\cgi-bin`".
- You may wish to examine the web configuration file that comes with the DCF to understand how Apache is configured; the file is "`$DCF_ROOT/httpd/conf/httpd.conf`".

2.4.2. Connecting to the web server

Point the web browser of your choice to the address for the per-user web server. For example:

`http://yourhostname:2004` (where 2004 is replaced with the appropriate port)

or

`http://yourhostname:8080` (typical for Windows installations)

If your Apache port is 2004, you should see the DCF Remote Service Interface. (Note: the typical Apache port for Windows installations is 8080.)

2.4.3. The DCF Remote Service Interface

The DCF provides a web-based user interface (see below) for accessing various system functions. The DCF Remote Service Interface is the top level or home page for DCF service facilities.

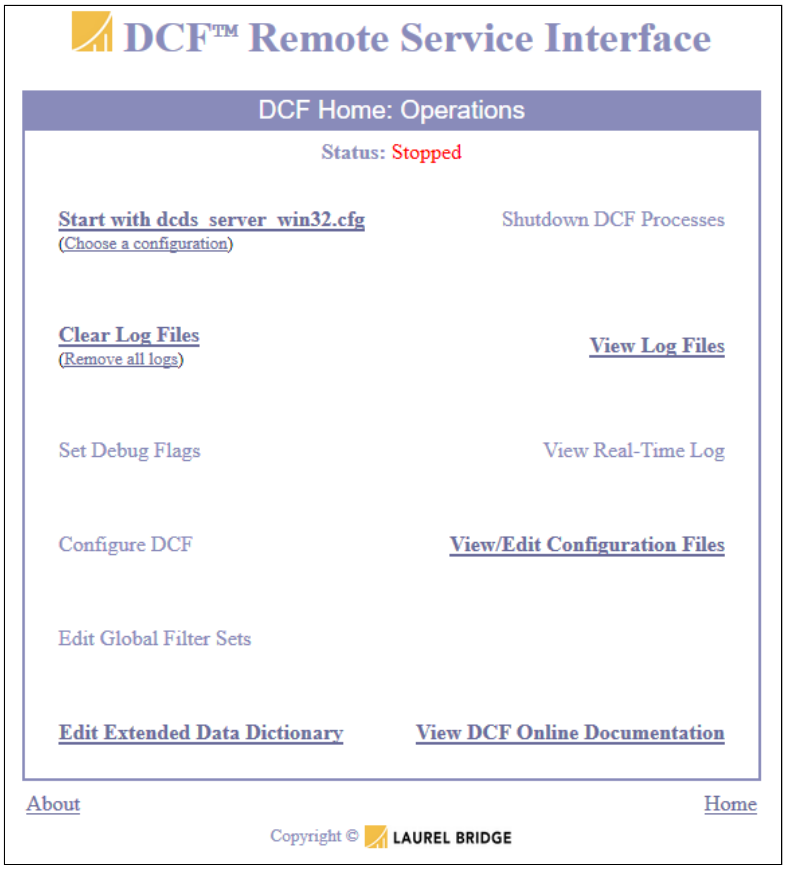


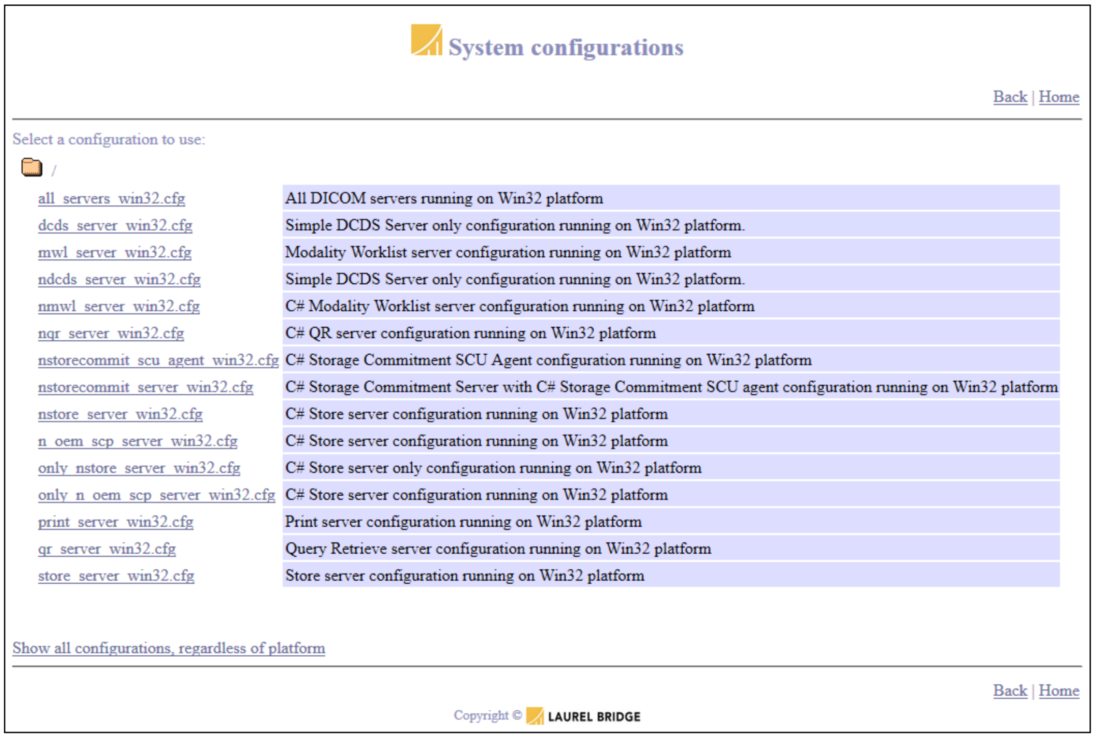
Figure 2: DCF Remote Service Interface Screen Example

The DCF Remote Service Interface typically contains the following links:

- Start with [DCF system config name] (Choose a configuration)
- Clear Log Files (Remove all logs)
- Set Debug Flags
- Configure [DCF system config name]
- Edit Global Filter Sets
- Edit Extended Data Dictionary
- Shutdown DCF Processes
- View Log Files
- View Real-Time Log
- View/Edit Configuration Files
- View DCF Online Documentation

2.4.3.1. Start with … (Choose a configuration)

A collection of programs defined by a DCF system configuration file is started. There are several system configurations defined which are used for testing. See the directory *`$DCF_USER_ROOT/cfg/systems`* for files which define various system configurations.



Select a configuration to use:

/

all_servers_win32.cfg	All DICOM servers running on Win32 platform
dcds_server_win32.cfg	Simple DCDS Server only configuration running on Win32 platform.
mwl_server_win32.cfg	Modality Worklist server configuration running on Win32 platform
ndcds_server_win32.cfg	Simple DCDS Server only configuration running on Win32 platform.
nmwl_server_win32.cfg	C# Modality Worklist server configuration running on Win32 platform
nqr_server_win32.cfg	C# QR server configuration running on Win32 platform
nstorecommit_scu_agent_win32.cfg	C# Storage Commitment SCU Agent configuration running on Win32 platform
nstorecommit_server_win32.cfg	C# Storage Commitment Server with C# Storage Commitment SCU agent configuration running on Win32 platform
nstore_server_win32.cfg	C# Store server configuration running on Win32 platform
n_oem_scp_server_win32.cfg	C# Store server configuration running on Win32 platform
only_nstore_server_win32.cfg	C# Store server only configuration running on Win32 platform
only_n_oem_scp_server_win32.cfg	C# Store server configuration running on Win32 platform
print_server_win32.cfg	Print server configuration running on Win32 platform
qr_server_win32.cfg	Query Retrieve server configuration running on Win32 platform
store_server_win32.cfg	Store server configuration running on Win32 platform

Show all configurations, regardless of platform

Figure 3: Example – List of System Configurations to Start

The OEM can choose whether or not to start applications this way.

The program *dcfstart.pl* can be run on the command line to perform the same function (see Section 2.4.4); alternately an OEM could start individual DCF or other processes in their own initialization code.

For more information about starting and stopping systems see Chapter 7 (Using DCF System Manager to control processes).

2.4.3.2. Shutdown DCF Processes

This will request that all programs started using the “Start with [config]” link are shut down.

The program `dcfstop.pl` can be run on the command line to perform the same function.

For more information about starting and stopping systems see Chapter 7 (Using DCF System Manager to control processes).

2.4.3.3. Clear Log Files

Delete log files for processes that are not running, and truncate log files for processes that are running, or for which the state of the writing process is not known (the .out.log files for example, since they do not have a process id in their name).

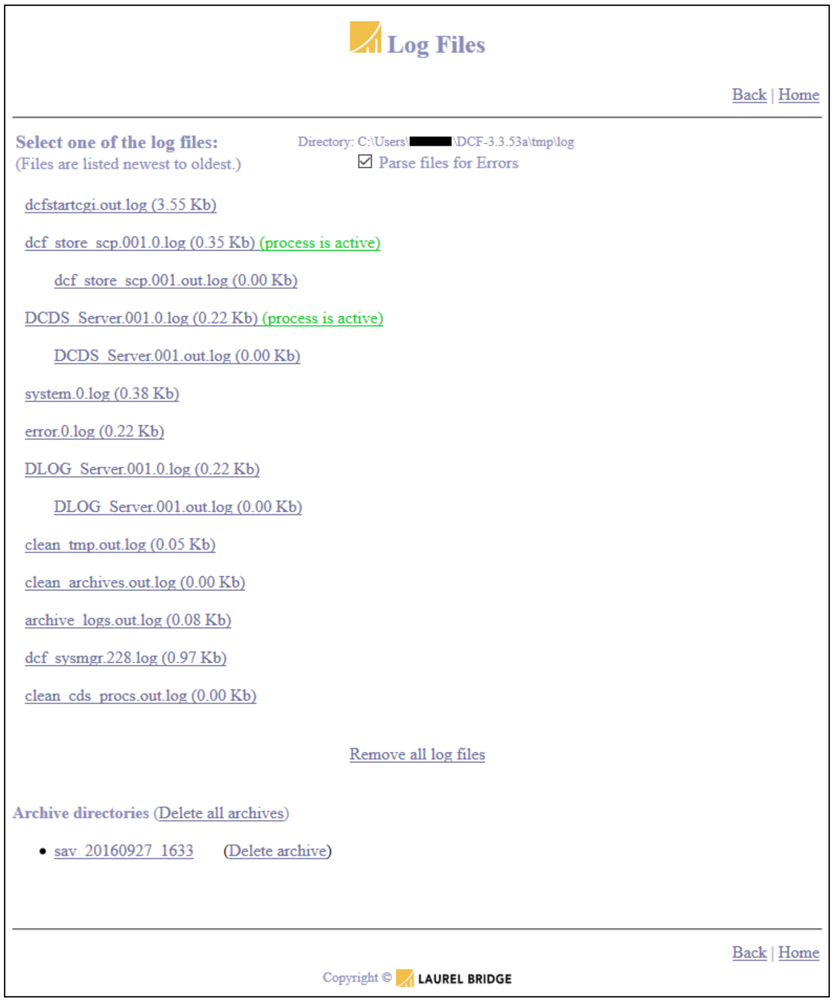
2.4.3.4. View Log Files

The files in the directory *`$DCF_USER_ROOT/tmp/log`* will be listed. (See the example in Figure 4 below.) If the process associated with a log file is currently alive, that will be noted. If the log file contains any errors, that fact will also be noted – since parsing the files for errors may take a long time if the files are large, you can click the checkbox at the top of the page to disable the parsing of the files and speed up the operation. Clicking on a file will allow a single log file to be viewed in the browser.

If you are using the standard DCF logging components, there will be three types of log files that are created in the log directory. All of these files have the same format but are produced in different manners. Each process started by *`dcfstart`* has a log file with the extension ".out.log". This file contains the standard output of the program. Only output that was explicitly written to the standard output will appear here. Typically, this file will be empty or nearly empty. The files with the extension ".<pid>.log" contain the logger output for the program with the given process id. The file system.log contains the output of the log server process, which may receive messages from multiple processes.

The default or reference implementation LOG adapter, as well as the dlog_server process, write log files as plain text. Only when a file is selected for viewing is it formatted as html. Several features are available while viewing log files. Many of the fields in the log header messages are displayed as html hyperlinks. Clicking on this field will create a filtered display of the log file. For example, clicking on the thread-id field in a message header will redisplay the log showing only messages from that thread. Clicking on the component name will redisplay the log showing only messages from that component. A link is available to return to the unfiltered log file.

If a process is running, it may be adding to its log file while you are viewing it. If you think there may be more text available, click the refresh- or reload-page button on your browser.



Select one of the log files:
(Files are listed newest to oldest.)

Directory: C:\Users\[redacted]\DCF-3.3.53a\tmp\log

☑ Parse files for Errors

dcfstartcgi.out.log (3.55 Kb)

dcf_store_scp.001.0.log (0.35 Kb) (process is active)

    dcf_store_scp.001.out.log (0.00 Kb)

DCDS_Server.001.0.log (0.22 Kb) (process is active)

    DCDS_Server.001.out.log (0.00 Kb)

system.0.log (0.38 Kb)

error.0.log (0.22 Kb)

DLOG_Server.001.0.log (0.22 Kb)

    DLOG_Server.001.out.log (0.00 Kb)

clean_tmp.out.log (0.05 Kb)

clean_archives.out.log (0.00 Kb)

archive_logs.out.log (0.08 Kb)

dcf_sysmgr.228.log (0.97 Kb)

clean_cds_procs.out.log (0.00 Kb)

Remove all log files

Archive directories (Delete all archives)

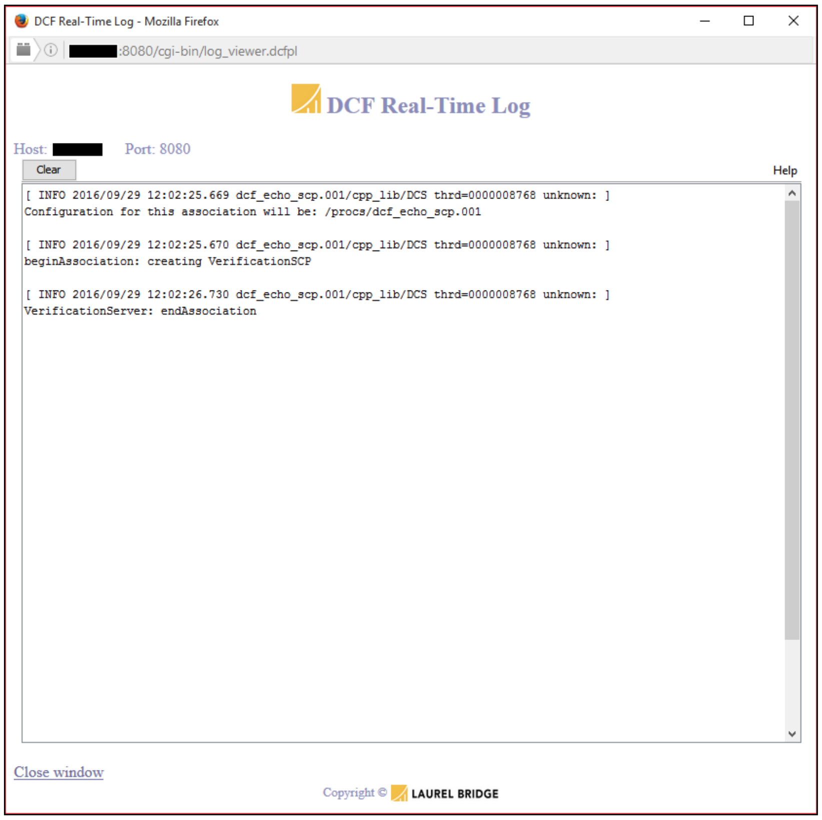
- sav_20160927_1633 (Delete archive)

Figure 4: Example – Partial List of Log Files

2.4.3.5. View DCF Real-Time Log

Start the real-time log viewer applet. This is a Java applet that runs on the browser and receives text from the log server process. All messages sent to the log server are forwarded to the log viewer.

Note: you may have to add a security exception for "`http://<hostname>:8080`" via the Java console Security tab to allow this applet to run on the system.


DCF Real-Time Log - Mozilla Firefox
[redacted]:8080/cgi-bin/log_viewer.dcfpl
DCF Real-Time Log
Host: [redacted]
Port: 8080
Clear
Help
Close window
Copyright © LAUREL BRIDGE

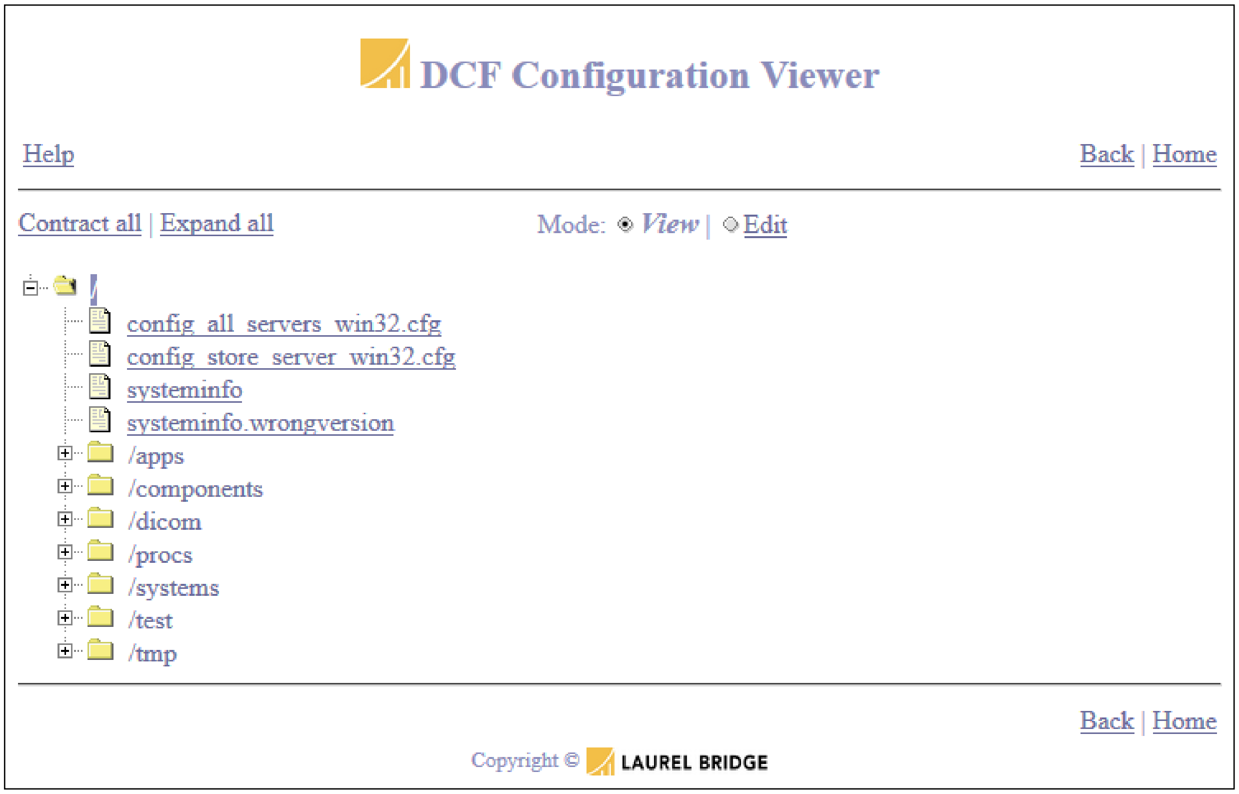
```
[ INFO 2016/09/29 12:02:25.669 dcf_echo_scp.001/cpp_lib/DCS thrd=0000008768 unknown: ]
Configuration for this association will be: /procs/dcf_echo_scp.001

[ INFO 2016/09/29 12:02:25.670 dcf_echo_scp.001/cpp_lib/DCS thrd=0000008768 unknown: ]
beginAssociation: creating VerificationSCP

[ INFO 2016/09/29 12:02:26.730 dcf_echo_scp.001/cpp_lib/DCS thrd=0000008768 unknown: ]
VerificationServer: endAssociation
```

Figure 5: DCF Real-Time Log Screen Example

Note: The Real-Time Log is a Java applet. You will need to have Java installed to run the applet. See Appendix H: Section 5, Java Applet Issues for information on possible applet issues.

2.4.3.6. View/Edit Configuration Files

View or edit objects in the CDS (Configuration Data Services) repository. The CDS is the facility in DCF for managing configuration data. CDS configuration data is stored in the file system under the directory: `$DCF_USER_ROOT/cfg` (a.k.a. $DCF_CFG). This interface allows you access to all the configuration files used by the DCF and its components. Use of this interface is recommended primarily for advanced users who are familiar with the DCF and its components, as there are *many* files and each file has many configuration attributes in it. (For information on presenting a simplified view of the configuration attributes, see Appendix H: Customizing the DCF Remote Service Interface.)


DCF Configuration Viewer

Help

Back | Home

Contract all | Expand all

Mode: ● View | ○ Edit

/
- config_all_servers_win32.cfg
- config_store_server_win32.cfg
- systeminfo
- systeminfo.wrongversion
- /apps
- /components
- /dicom
- /procs
- /systems
- /test
- /tmp

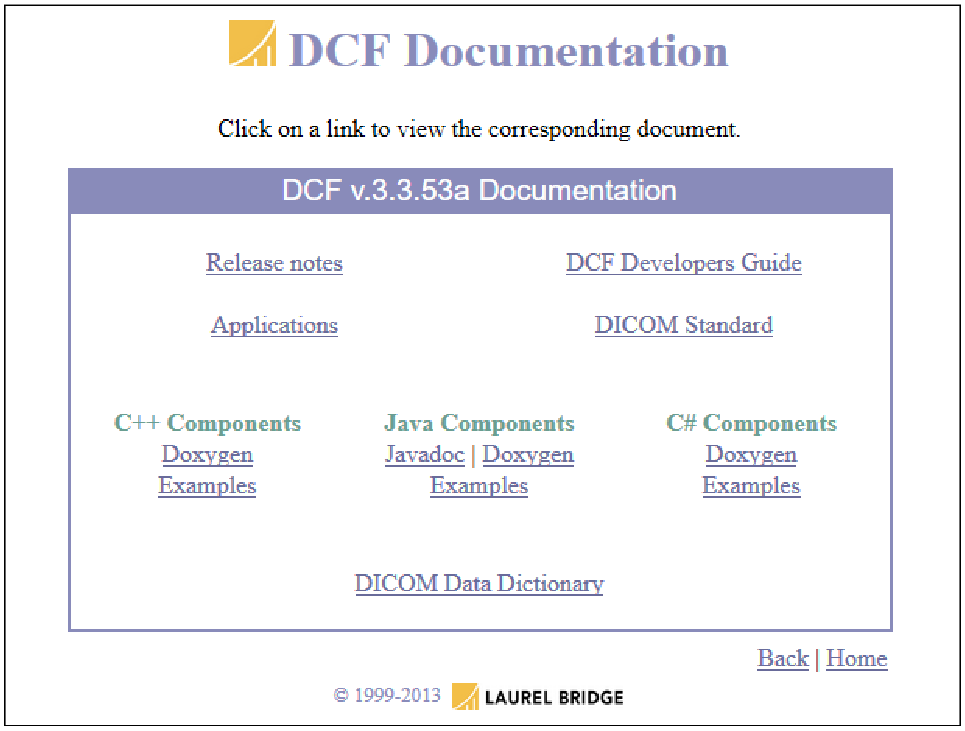
Back | Home

Copyright © LAUREL BRIDGE

Figure 6: DCF Configuration Viewer Screen Example

2.4.3.7. View DCF Online Documentation

This link leads to the online documentation page.



DCF Documentation

Click on a link to view the corresponding document.

DCF v.3.3.53a Documentation

Release notes
DCF Developers Guide
Applications
DICOM Standard

C++ Components
Doxygen
Examples

Java Components
Javadoc | Doxygen
Examples

C# Components
Doxygen
Examples

DICOM Data Dictionary

Back | Home

© 1999-2013 LAUREL BRIDGE

Figure 7: DCF Documentation Screen Example

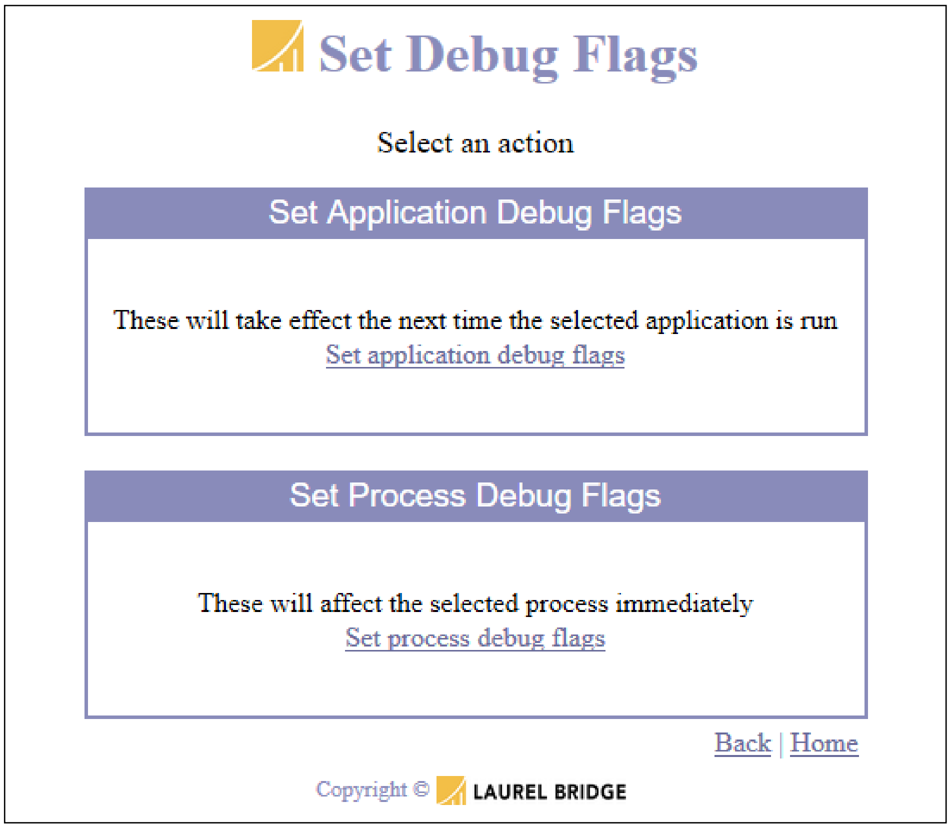
The typical items listed under that page are:

- **Release Notes** – the release notes document for the current release
- **Applications** – man page style docs which are generated for each DCF application
- **User Guides** – tech notes and other higher level documents
- **DICOM Documents** – these are references to the DICOM standard and possibly other information. They are provided as a convenience. These files are normally in .PDF format, so Adobe Acrobat ™ or some other PDF viewer must be accessible by the browser.
- **Conformance Documents** - these are copies of sample documents that may be used as the basis for preparing a custom DICOM statement for a device or application. The DICOM Standard, Chapter 2, provides the definitive examples for preparing a DICOM Conformance Statement.
- **C++ Components** – generated documents for C++ libraries (C++ Toolkits)
- **Java Components** – generated documents for Java libraries (Java Toolkits)
- **C# Components** – generated documents for C#.net assemblies (C# Toolkits)
- **DICOM Data Dictionary** – Selecting this link will display an auto-generated DICOM data dictionary based on the current configuration on the system.

Component documentation for C++, Java, C# and other applications is generated using the Doxygen program. Java documentation is also created using Javadoc.

2.4.3.8. Set Debug Flags

This allows debugging or verbosity settings to be adjusted for applications. Debug flags are a special type of configuration data that are defined for each application; they are typically used to control logging verbosity, but may also be used to enable special operating modes.


Set Debug Flags

Select an action

Set Application Debug Flags

These will take effect the next time the selected application is run
Set application debug flags

Set Process Debug Flags

These will affect the selected process immediately
Set process debug flags

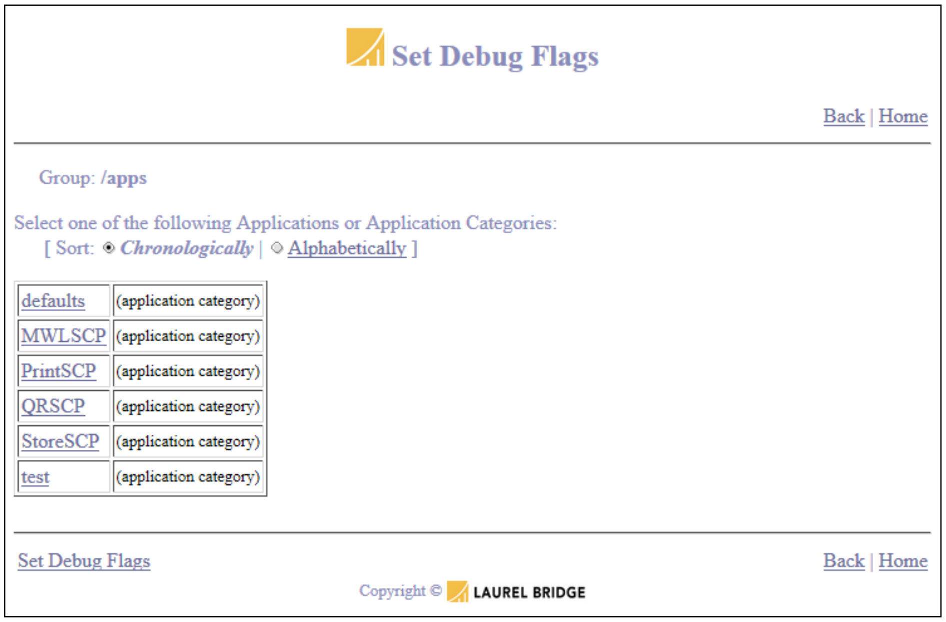
Back | Home

Copyright © LAUREL BRIDGE

Figure 8: DCF Set Debug Flags Screen Example

Two selections are under the set debug flags link: `Set application debug flags` and `Set process debug flags`. The first will save configuration data under the `/apps` CDS hierarchy. Any program can be adjusted here and the settings will take effect the next time that program is run. The second option allows you to make changes to the `/procs` CDS hierarchy. Only programs that are currently running can be adjusted here. Changes will take effect immediately; the process does not need to be restarted. This can prove valuable when it is desirable to adjust the logging verbosity of a server, but the server cannot be shutdown or restarted at the time.

Either of the two "Set Debug Flags" selections prints a list of applications or processes (an application that is running) – see Figure 9 below. Selecting an application or process will take you to a page which prints the list of library components contained by that application, shown below in Figure 10. Selecting one of the components will take you to a page that lists the available debug settings for that component. Debug settings for components can be adjusted independently – see Figure 11 below. This can prove valuable when it is desirable to have verbose output from one component but not from the others. (See Appendix H: Customizing the DCF Remote Service Interface for information on presenting a simpler interface for modifying the debug flags.)



Set Debug Flags

Back | Home

Group: /apps

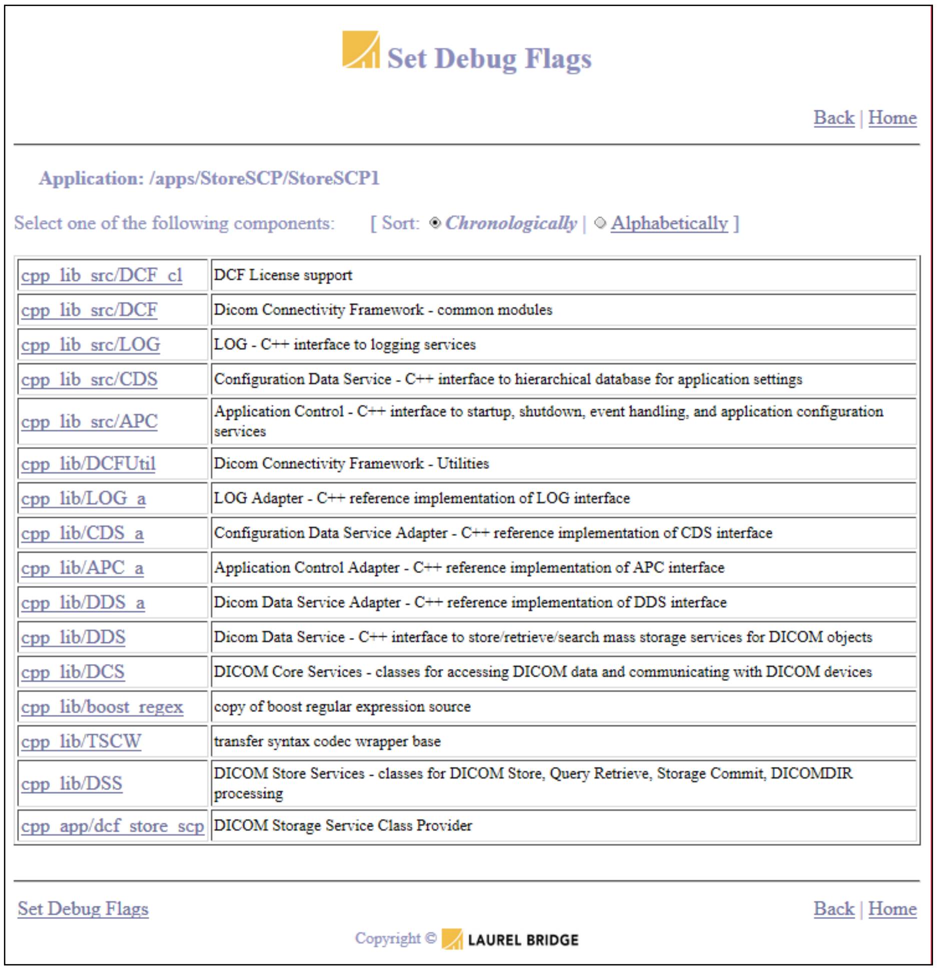
Select one of the following Applications or Application Categories:
[Sort: ● *Chronologically* | ○ Alphabetically]

defaults	(application category)
MWLSCP	(application category)
PrintSCP	(application category)
QRSCP	(application category)
StoreSCP	(application category)
test	(application category)

Set Debug Flags — Back | Home

Copyright © LAUREL BRIDGE

Figure 9:Set Debug Flags Example – List of Running Processes



Set Debug Flags

Back | Home

Application: /apps/StoreSCP/StoreSCP1

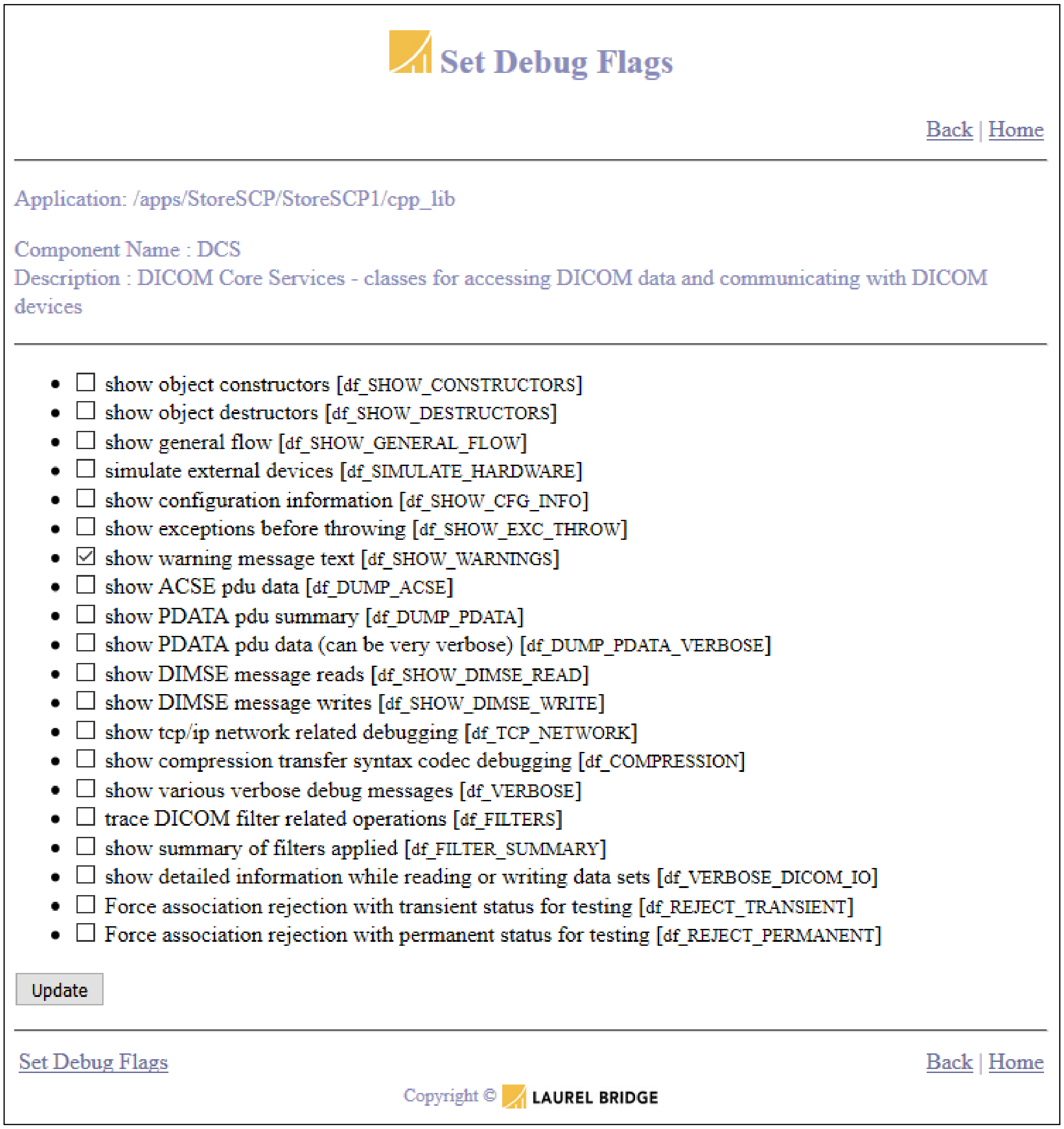
Select one of the following components: [Sort: ● *Chronologically* | ○ Alphabetically]

cpp_lib_src/DCF_cl	DCF License support
cpp_lib_src/DCF	Dicom Connectivity Framework - common modules
cpp_lib_src/LOG	LOG - C++ interface to logging services
cpp_lib_src/CDS	Configuration Data Service - C++ interface to hierarchical database for application settings
cpp_lib_src/APC	Application Control - C++ interface to startup, shutdown, event handling, and application configuration services
cpp_lib/DCFUtil	Dicom Connectivity Framework - Utilities
cpp_lib/LOG_a	LOG Adapter - C++ reference implementation of LOG interface
cpp_lib/CDS_a	Configuration Data Service Adapter - C++ reference implementation of CDS interface
cpp_lib/APC_a	Application Control Adapter - C++ reference implementation of APC interface
cpp_lib/DDS_a	Dicom Data Service Adapter - C++ reference implementation of DDS interface
cpp_lib/DDS	Dicom Data Service - C++ interface to store/retrieve/search mass storage services for DICOM objects
cpp_lib/DCS	DICOM Core Services - classes for accessing DICOM data and communicating with DICOM devices
cpp_lib/boost_regex	copy of boost regular expression source
cpp_lib/TSCW	transfer syntax codec wrapper base
cpp_lib/DSS	DICOM Store Services - classes for DICOM Store, Query Retrieve, Storage Commit, DICOMDIR processing
cpp_app/dcf_store_scp	DICOM Storage Service Class Provider

Set Debug Flags — Back | Home

Copyright © LAUREL BRIDGE

Figure 10: Set Debug Flags example – List of Components in Store SCP


Set Debug Flags

Back | Home

Application: /apps/StoreSCP/StoreSCP1/cpp_lib

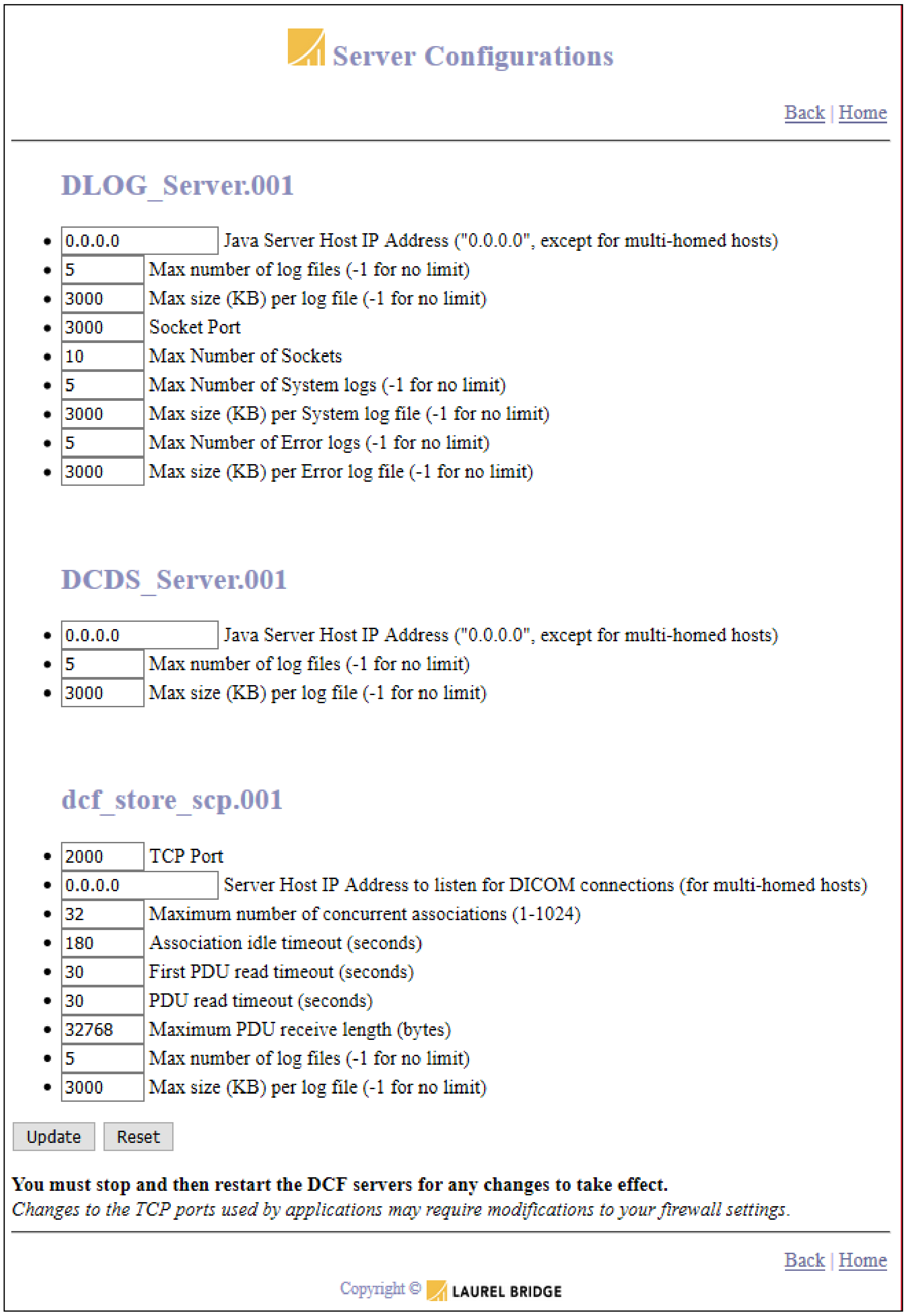
Component Name : DCS
Description : DICOM Core Services - classes for accessing DICOM data and communicating with DICOM devices

- ☐ show object constructors [df_SHOW_CONSTRUCTORS]
- ☐ show object destructors [df_SHOW_DESTRUCTORS]
- ☐ show general flow [df_SHOW_GENERAL_FLOW]
- ☐ simulate external devices [df_SIMULATE_HARDWARE]
- ☐ show configuration information [df_SHOW_CFG_INFO]
- ☐ show exceptions before throwing [df_SHOW_EXC_THROW]
- ☑ show warning message text [df_SHOW_WARNINGS]
- ☐ show ACSE pdu data [df_DUMP_ACSE]
- ☐ show PDATA pdu summary [df_DUMP_PDATA]
- ☐ show PDATA pdu data (can be very verbose) [df_DUMP_PDATA_VERBOSE]
- ☐ show DIMSE message reads [df_SHOW_DIMSE_READ]
- ☐ show DIMSE message writes [df_SHOW_DIMSE_WRITE]
- ☐ show tcp/ip network related debugging [df_TCP_NETWORK]
- ☐ show compression transfer syntax codec debugging [df_COMPRESSION]
- ☐ show various verbose debug messages [df_VERBOSE]
- ☐ trace DICOM filter related operations [df_FILTERS]
- ☐ show summary of filters applied [df_FILTER_SUMMARY]
- ☐ show detailed information while reading or writing data sets [df_VERBOSE_DICOM_IO]
- ☐ Force association rejection with transient status for testing [df_REJECT_TRANSIENT]
- ☐ Force association rejection with permanent status for testing [df_REJECT_PERMANENT]

Figure 11: Set Debug Flags Example – List of Debug Flags in the DCS Component

2.4.3.9. Configure [DCF system config name]

This provides convenient access to a simplified view into the configuration data for the currently selected system configuration. Each application may have a large configuration file that is created by combining copies of each subcomponent's configuration. Often, only a few items need to be edited. (See Appendix H: Customizing the DCF Remote Service Interface for information on this option and creating your own customized configuration screen.)



Server Configurations

Back | Home

DLOG_Server.001

- 0.0.0.0 Java Server Host IP Address ("0.0.0.0", except for multi-homed hosts)
- 5 Max number of log files (-1 for no limit)
- 3000 Max size (KB) per log file (-1 for no limit)
- 3000 Socket Port
- 10 Max Number of Sockets
- 5 Max Number of System logs (-1 for no limit)
- 3000 Max size (KB) per System log file (-1 for no limit)
- 5 Max Number of Error logs (-1 for no limit)
- 3000 Max size (KB) per Error log file (-1 for no limit)

DCDS_Server.001

- 0.0.0.0 Java Server Host IP Address ("0.0.0.0", except for multi-homed hosts)
- 5 Max number of log files (-1 for no limit)
- 3000 Max size (KB) per log file (-1 for no limit)

dcf_store_scp.001

- 2000 TCP Port
- 0.0.0.0 Server Host IP Address to listen for DICOM connections (for multi-homed hosts)
- 32 Maximum number of concurrent associations (1-1024)
- 180 Association idle timeout (seconds)
- 30 First PDU read timeout (seconds)
- 30 PDU read timeout (seconds)
- 32768 Maximum PDU receive length (bytes)
- 5 Max number of log files (-1 for no limit)
- 3000 Max size (KB) per log file (-1 for no limit)

You must stop and then restart the DCF servers for any changes to take effect.
Changes to the TCP ports used by applications may require modifications to your firewall settings.

Back | Home

Copyright © LAUREL BRIDGE

Figure 12: Modifying the Server Configuration

2.4.3.10. Edit Global Filter Sets

This option accesses an interface that allows the user to edit DICOM Filter Sets.

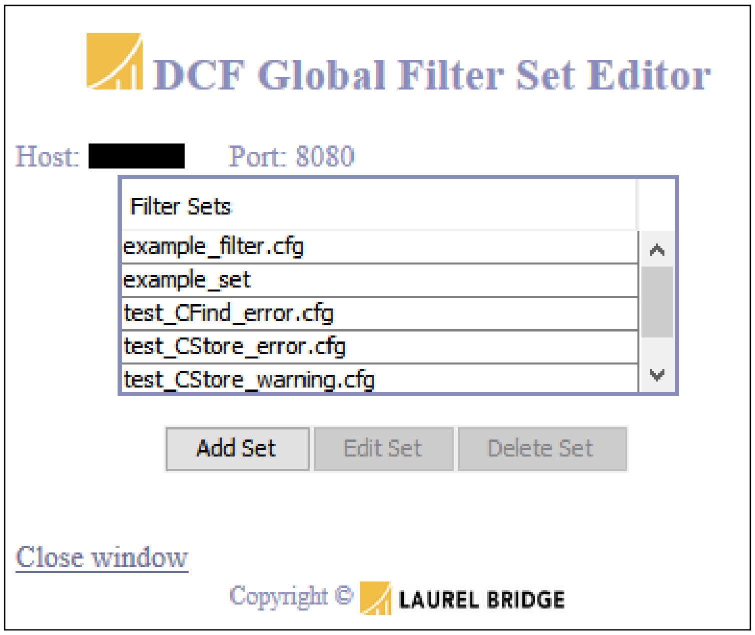


Figure 13: DCF Global Filter Set Editor Screen Example

A `DicomElementFilter` is used to modify DICOM elements in a DIMSE message. For example, you can specify that a certain tag is always changed to another value or that the data element with a particular tag is removed from the message. Many ways to use the `DicomElementFilter` are provided and not all are detailed here.

The `DicomElementFilter` has five versions. The "Copy Filter" version allows you to copy elements in a DIMSE message. The "Remove Filter" will remove elements from a message, while the "Add/Replace Filter" will add or replace elements in a message, and the "Modify Filter" allows you to modify the data in a DIMSE message via regular expressions. The fifth version, "Element Filter (full)", allows you to do all of these things. As the fifth is the most general, its basic functionality is described here, with the understanding that these descriptions apply to the other options, each of which has a subset of these capabilities.

The "Element Filter (full)" has six tables each displaying a list of elements. These elements are checked against the elements in a message and applied accordingly to the rule for the section where they are listed. (As you enter data into the filter's fields, please note that the *rules are applied top-to-bottom*. That is, the results of the first filter's actions are passed to the next filter, e.g., the results of the elements that are copied will then be processed through the list of elements that are to be removed, etc.)

Elements may be:

- matched
- copied
- removed
- removed if null
- added and/or replaced
- edited and moved

The various types of `DicomElementFilter` have subsets of these capabilities; for example, the Copy Filter will only include the capability to copy data.

The Mapping List Filter (`DicomMappingListFilter`) is a specialized and more advanced version of the Add/Replace Filter. It provides an efficient way to match a large number of possible values for a particular attribute (key tag) and then add/replace one or more elements, depending on the key tag value (or values) that is matched.

The Element Composer Filter (`DicomElementComposerFilter`) is a more advanced version of the Element Filter. It uses regular expressions to parse input elements, and then the captured results from the inputs can be put back together to create new output elements.

The Pixel Value Shift Filter (`PixelValueShiftFilter`) can be used to shift the bits in pixel data values left or right, for image data manipulation. (This filter is primarily used in unit/integration tests for data-set and DIMSE-message filtering, but it is also available for real world image data manipulation.)

The Planar Configuration Convert Filter (`DicomPlanarConfigConvertFilter`) is used to convert color pixel data from interleaved (RGB RGB RGB…) to planar (RRR…GGG…BBB…), or vice versa. Note that this filter will not modify the data unless samples-per-pixel is greater than 1, and bits-allocated is 8. This filter will recognize pixel data that is stored in attribute 7FE0,0010 in the top level data set, as well as pixel data that is contained in either of the attributes Basic-Grayscale-image-sequence or Basic-Color-image-sequence.

The Pad Value Filter (PadValueFilter) can be used to pad a string value with a null, a space character, or a user-specified character until the string is a given length. For example, you can make sure that the Accession Number is a certain length and has leading zeroes, or make sure that the Patient Name is padded with spaces until it is a certain length.

Note: The Filter Set Editor is a Java applet. You will need to have Java installed to run the applet. See Appendix H: Section 5, Java Applet Issues for information on possible applet issues.

2.4.4. The DCF Command-line Operation

It is possible to run DCF examples without running Apache and without using the web-based service interface. A web service interface is provided as a simple way to start/stop the servers that are used by the DCF; it also provides a convenient way to access documentation, configure applications or components, etc. But it is not necessary for the DCF's operation, and this section will explain how you can start and stop DCF components from the command line.

Look in the *`%DCF_ROOT%/cfg/systems`* subdirectory; you will see the configuration files and scripts that are used to start various DCF servers and applications. These configurations can be used without the web interface as follows:

To "Start" or "Restart" from a Windows DCF command prompt, type:

```
perl –S dcfstart.pl -cfgfile "%DCF_CFG%/systems/config_file_name"
```

- or -

```
perl –S dcfrestart.pl -cfgfile "%DCF_CFG%/systems/config_file_name"
```

Example:

```
perl –S dcfrestart.pl -cfgfile "%DCF_CFG%/systems/store_server_win32.cfg"
```

To stop the running servers, you would type in a dcfstop command at a DCF Command Prompt:

```
perl –S dcfstop.pl
```

This sort of common activity is often done more easily through the web service interface, which is one reason that option is provided.

Note that similar commands may be used under Windows or UNIX OSes; syntax changes slightly for path and environment variable specification.

For more information about starting and stopping systems see Chapter 7 (Using DCF System Manager to control processes).

(Appendix D: Section 1.2 shows an example of starting a configuration of DCF servers from the command line to run a Java app from the command line.)

(Appendix G: has helpful information about invoking the Perl interpreter.)

2.5. Using Multiple Versions of the DCF

During active development using the DCF, you might find yourself in a situation in which you are using multiple versions of the toolkit. For example, you might be doing maintenance – such as bug fixing – on a product built with one version of the DCF, and doing development on a new product that uses another version of the DCF. It is generally not a problem to develop and test with multiple versions of the DCF.

2.5.1. UNIX

On UNIX systems, the OS itself separates the installations and their environments, providing discrete development environments for each version of the DCF.

2.5.2. Windows

For Windows platforms the situation is slightly more complicated. After DCF version 3.2.0, it is possible to install multiple versions of the DCF on a Windows box and have them operate separately and discretely.

DCF versions 3.2.0 and later can be installed concurrently on a single system. For example, you can have DCF 3.2.0 and DCF 3.2.2 installed on the same box – they will be installed in parallel separate directories in *`Program Files`* and will have parallel options on the *`Start`* menu. You can also install DCF 3.2.0 (or later) alongside a single copy of an older version of the DCF, e.g., you can have both DCF 2.8.8 and DCF 3.2.2 since they will be installed in parallel separate directories.

Please note that you can **not** use the *`DCF installer`* to install multiple copies of older versions of the DCF (before DCF 3.2.0) at the same time on a Windows system. If you attempt this, the second copy will overwrite the previous copy, and you will be unable to access the *`Start`* menu options or otherwise use the previous copy.

Certainly the safest approach to developing and testing with multiple versions of the DCF is to install each version on its own dedicated system. Certain library components that are used by C# applications may present difficulties, usually when a wrong version has been unregistered by the operating system.

2.5.3. Testing

To test an application built on a box with multiple versions of the DCF installed, you just need to make sure that you are using the correct environment variables that refer to the DCF. Setting the environment is done most easily by selecting the “DCF Command Prompt” option on the *`Start`* menu for the desired version of the DCF – this action correctly sets the environment used for all commands that are run inside that command prompt window.

2.6. Windows x86 vs. x64

The DCF is available in a 64-bit (x64) and 32-bit (x86) versions. The x86 version of DCF can be used by a developer on a Windows x64 Operating System. However, .NET projects built with x86 DCF must have the target machine architecture set to “x86” and not “AnyCPU” in order to run on the x64 development machine.

If you took your AnyCPU .NET project (which was built on x64 OS) and ran it on a x86 machine, it would work. When compiling on a x64 development machine and the target platform is AnyCPU, then you are always going to run the 64-bit CLR even if you launch your .NET app from the 32-bit command shell. The AnyCPU .NET project that is referencing x86 DCF .NET assemblies, it will fail when run. This is because DCF uses IN_PROC COM dll’s that was compiled for x86. By setting your projects target to “x86” you effectively are forcing the 32-bit CLR to run, which means any native COM objects that were compiled for x86 will run. The .NET project would run if deployed to a Windows x86 Operating System.

If you want to target x64 Windows Operating systems, use an x64 DCF toolkit, where all the native code is compiled for x64.

3. DICOM Programming Overview

The DCF supports DICOM programming using numerous components, which are available to programmers on various platforms using various languages. Most DICOM programmers will interact with the classes described in this chapter.

For specific and detailed examples of using these classes in C++, Java, and C#, see the language-specific chapters and associated examples, or the various example source files and their associated online documentation included with the DCF toolkit installation.

Beyond the examples in the language chapters of this guide, the complete working source code for examples of common DICOM integration tasks are found in the following installation directories:

C++:	*$DCF_ROOT/devel/csrc/com/lbs/examples*
Java:	*$DCF_ROOT/devel/jsrc/com/lbs/examples*
C#:	*$DCF_ROOT/devel/cssrc/com/lbs/examples*

For additional information, see also Chapter 8 – The DCF Development Environment.

Refer to the DICOM Standard, Chapter4 for DICOM service class specifications:

A Service Class Specification defines a group of one or more SOP Classes related to a specific function which is to be accomplished by communicating Application Entities. A Service Class Specification also defines rules which allow implementations to state some pre-defined level of conformance to one or more SOP Classes. Applications may conform to SOP Classes as either a Service Class User (SCU) or Service Class Provider (SCP).

Note: Such interaction between peer Application Entities work on a 'client/server model.' The SCU acts as the 'client,' while the SCP acts as the 'server'. The SCU/SCP roles are determined during Association establishment.

3.1. Core DCF DICOM classes

3.1.1. Element related

Classes such as `DicomElement`, `DicomDataSet`, and `DicomDataDictionary` are found in the DCS library in various forms. For details, see the online documentation for these classes.

3.1.2. Association Manager

`AssociationManager` is the daemon for DICOM server side associations. `AssociationManager` listens to a configurable TCP port for incoming DICOM association requests. There can be multiple AssociationManager objects in a single process, each listening on its own port.

One object that implements the AssociationConfigPolicyManager interface may be registered with an AssociationManager. This object is called at the start of an association. The AssociationConfigPolicyManager can examine the connection request (A-Assoc-Rq-PDU) and determine if the connection should be allowed, and what set of configuration parameters (DicomSessionSettings in C# and Java) should be used. By default, for each association, the AssociationManager creates an instance of AssociationAcceptor which handles communications for that association in a separate thread.

Objects that implement the AssociationListener interface can be registered with an AssociationManager. Each time an association or DICOM connection starts or ends, each registered listener is notified. At association startup, for example, an AssociationListener object might create an SCP object of some type. This object may register as a PresentationContextAcceptor with the AssociationAcceptor that is handling the connection. If negotiation succeeds, messages for a given DICOM presentation context are dispatched to the appropriate SCP objects. In this way, complex servers can be created that can handle multiple service classes on a particular connection.

Note: At the time of this writing the maximum number of concurrent associations that DCF allows is 4096. In practice, system resources will limit your application to a lower concurrent association count; you will probably run out of either threads or memory before you hit 4096 associations. We recommend you make the MaxConcurrentAssociations parameter configurable from within your application. Doing so will allow you to adjust it in the field as necessary.

3.2. Verification Service Class

3.2.1. Verification Client (SCU)

The `VerificationClient` is used to communicate with Verification Service Class providers or servers. Verification service class is used to test connectivity between DICOM application entities (AE's).

The user requests `VerificationClient` to connect to the SCP and then send a C-Echo-Request DIMSE message. The SCP is expected to respond by sending a C-Echo-Response DIMSE message back to the client.

3.2.2. Verification Server (SCP)

`VerificationServer` provides an implementation of the Verification Service class. When the verification SOP class is requested, the `VerificationServer` will create an instance of `VerificationSCP` to handle that presentation context on that association.

3.3. Storage-related Service Classes

3.3.1. Store Client (SCU)

The `StoreClient` provides a batch interface for sending collections of images or other objects to a Storage Service Class provider or server. StoreClient makes use of the StoreSCU class for lower level functions.

Storage service class is used to transmit images or other DICOM objects (SOP instances) to an archive or other storage device.

3.3.2. Store Server (SCP)

`StoreServer` provides an implementation of the Storage Service class. When one of the Storage SOP classes is requested, `StoreServer` will create an instance of `StoreSCP` to handle that presentation context on that association. The SOP classes that will be accepted can be configured on a per association basis.

`StoreServer` receives SOP instances from a Store SCU.

The `DicomDataService::storeObject()` method is invoked for each image that is received. By providing a `DicomDataService` adapter, an OEM can see every image/object that is received, without needing to be involved with the details involved with handling concurrent DICOM associations.

3.3.3. Storage Commitment Server (SCP)

For Java and C#:

The `StoreCommitSCP, StoreCommitServer` provide the services described below for the DICOM Storage Commitment Push Model SOP class. See the online docs for additional details.

Overview for C++:

`StorageCommitmentServer` provides an implementation of the Storage Commitment Service class. When the Storage Commitment SOP class is requested (Push-Model only), `StorageCommitmentServer` will create an instance of `StorageCommitmentSCP` to handle that presentation context on that association.

It receives Commitment requests from a Storage commitment SCU.

The `DicomDataService::commitRequestReceived()` method will be invoked when an N-Action-Request DIMSE message is received from the SCU.

The `acceptStorageCommitment()` method can be invoked on `StorageCommitmentServer` to forward "commit-completed" messages back to the requesting SCU. If the SCU is still connected, and the configuration allows, the N-Event-Report-Request message will be sent to the SCU over the same association. Otherwise, `StorageCommitmentServer` (or `StorageCommitmentSCP`) will create a new association on which to send the notification.

3.3.4. Storage Commitment Client (SCU)

For Java and C#:

The `StoreCommitSCU` provides the services described below. See the online docs for additional details.

Overview for C++:

The `StorageCommitmentClient` is used to communicate with Storage Commitment Service Class providers or servers.

Storage Commitment service class is used to allow one device to request that another device accept long term storage responsibility for images or other SOP instances.

The `DicomDataService::commitRequestSent()` method will be invoked once for each SOP instance that was referenced in an outbound N-Action-Request DIMSE message.

The `DicomDataService::commitRequestAcknowledged()` method will be invoked when the N-Action-Response DIMSE message is received from the SCP.

3.3.5. Storage Commitment Client Agent

For Java and C#:

The `StoreCommitSCUAgent, StoreCommitSCUAgentMessageHandler` provide the services described below. See the online docs for additional details.

Overview for C++:

The `StorageCommitmentClientAgent` is used receive storage commitment acknowledgements from a Storage Commitment Service Class provider or server. Note that this is the unusual case in DICOM where the SCU is also the association accepter.

The `DicomDataService::commitRequestCompleted()` method will be invoked when an N-Event-Report-Request DIMSE message is received from the SCP.

3.4. Query/Retrieve (Q/R) Service Class

3.4.1. Q/R Client (SCU)

The `QRSCU` is used to communicate with Query Retrieve Service Class providers or servers.

Query Retrieve service class is used to query some type of a database for images or other SOP instances. These objects may be retrieved using a C-Move request (in which case the Q/R server functions as a Storage client, and sends images to the requestor, or to a third party over a new association), or a C-Get request, in which case the images are transmitted back to the requestor over the same association.

The Query is created from a data set provided directly by the user that is combined with elements from an optional query configuration file.

The user can choose to receive responses either all at once, in a list, or as they arrive, by providing an implementation of the `DicomQueryListener` interface (C++), or by implementing the `QueryListener` interface (Java, C#).

In addition to sending DIMSE responses to a C-Move-Request, the SCP sends requested instances to a Storage service class SCP over a new association. That store SCP may be on the same host as the Query/Retrieve client. Currently, support for C-Get in `QRSCU` is disabled in the C++ version. Contact Laurel Bridge Software if you wish to use this functionality.

A `QRSCU` user can also provide a `DicomDataService` adapter that is invoked each time the local StoreServer receives an image.

3.4.2. Q/R Server (SCP)

`QRServer` provides an implementation of the Query Retrieve Service class. When one of the Query Retrieve SOP classes is requested, `QRServer` will create an instance of `QRSCP` to handle that presentation context on that association. The SOP classes that will be accepted can be configured on a per association basis.

`QRServer` receives C-Find, C-Move or C-Get requests and sends responses. For C-Move and C-Get requests, matching instances are sent by a `StoreClient`, to a remote Store SCP.

`QRServer` uses the `DDS::DicomDataService` interface to communicate with the data base provider. The OEM typically provides a custom implementation of `DicomDataService`, which searches their database. The `DicomDataService` adapter implements the `findObjects()` methods. These methods take a query identifier (`QRIdentifier`), and return either a list of matching data sets, or references to matching instances in mass storage. `QRSCP` takes that information and forms the appropriate DIMSE response messages, and/or `StoreClient` requests.

3.5. Modality Worklist Service Class

3.5.1. MWL SCU

The `MWLSCU` is used to communicate with Modality Worklist (or General Purpose Worklist) Service Class providers or servers.

The Modality Worklist service class is similar to Query/Retrieve in that C-Find-Request DIMSE messages are sent to find objects on the server and matching objects are returned in one or more C-Find-Response messages. It is different from Query/Retrieve in that different SOP classes are requested and only the C-Find operation is supported.

You can also use the `MWLSCU` class for General Purpose Worklist by setting the SOP class UID in your Association Info object to General Purpose Worklist.

There are also examples in the C# examples directory called `ex_nmwl_scu` and in Java examples called `ex_jmwl_scu` that may be modified to do GPWorklist instead of MWL. There are high level GP objects in the LaurelBridge.DIS namespace (.NET toolkit) such as `GPPerformedProcedureStep.cs, GPScheduledProcedureStep.cs, GPWorklistItem.cs` that wrap up some of the IOD modules that may be needed for General Purpose Worklist.

3.5.2. MWL Server (SCP)

`MWLServer` provides an implementation of the Modality Worklist (or General Purpose Worklist) Service class. When one of the Worklist SOP classes is requested, `MWLServer` will create an instance of `MWLSCP` to handle that presentation context on that association. The SOP classes that will be accepted can be configured on a per association basis.

This is similar to `QRServer`, except that different SOP classes are requested, and only the C-Find operation is supported. `MWLServer` is easily customized by providing a `DicomDataService` implementation.

3.6. Modality Performed Procedure Step Service Class

3.6.1. MPPS Client (SCU)

The `MPPSSCU` is used to communicate with Modality Performed Procedure Step Service Class providers or servers.

`MPPSSCU` creates and updates instances of Modality Performed Procedure Step objects. It sends N-Create and N-Set DIMSE messages to an MPPS SCP or server. The user instructs the `MPPSSCU` to connect to the SCP, and uses the `n_set()` and `n_create()` methods to send the appropriate DIMSE messages.

3.6.2. MPPS Server (SCP)

`MPPSServer` provides an implementation of the Modality Performed Procedure Step Service class. When the MPPS SOP class is requested, `MPPSServer` will create an instance of `MPPSSCP` to handle that presentation context on that association. The SOP classes that will be accepted can be configured on a per association basis.

`MPPSServer` is easily customized by providing a `DicomDataService` implementation.

`MPPSServer` receives N-Set and N-Create DIMSE messages from MPPS SCUs. The `DicomDataService` interface is used to forward these requests to the server application. The user provides a `DicomDataService` adapter. When N-Create is received, the `DicomDataService::storeObject()` method is invoked. When N-Set is received, `DicomDataService::updateObject()` is invoked.

3.7. Print Service Class

3.7.1. Print Client (SCU)

`PrintClient` provides a batch job interface to allow clients to specify DICOM print requests in a simplified structured manner. The user provides a full description of the print job to `PrintClient`. `PrintClient` then handles the complexities of sending the various DIMSE request messages to complete the job.

The user can provide an implementation of the `PrintClientListener` interface, which will be called each time either the `Printer` or `PrintJob` status changes.

3.7.2. Print Server (SCP) – (C++ only)

`PrintServer` provides an implementation of the Print Service class. When one of the Print SOP classes is requested, `PrintServer` will create an instance of `PrintSCP` to handle the Print Management meta-SOP-class or the Printer SOP class. The SOP classes that will be accepted can be configured on a per association basis.

The `PrintServer` class handles all complexities of the DICOM print protocol from the server's perspective. After the various DIMSE message transactions (which describe one or more films) have been completed, `PrintServer` forwards the composite print job information to an implementation of the Printer interface. The implementation of the Printer interface communicates status information back to PrintServer via the `PrinterListener` and `PrintJobListener` interfaces.

3.8. DICOM File (Media Storage) Services

3.8.1. DICOM File Set Reader (FSR role)

The `DicomDirectoryRecord` and `DicomDir` classes along with `DicomFileInput` provide read functionality for DICOM File Sets, as defined in Part 10 of the standard. This amounts to reading a DICOMDIR file.

3.8.2. DICOM File Set Creator and Updater (FSC and FSU roles)

The `DicomDirectoryRecord` and `DicomDir` classes along with `DicomFileOutput` provide write and update functionality for DICOM File Sets, as defined in Part 10 of the standard. This amounts to creating and updating a DICOMDIR file.

4. C++ Programming Examples

This section presents a variety of C++ programming examples for common DICOM integration tasks. See `$DCF_ROOT/devel/csrc/com/lbs/examples/` for the complete working source code for these and additional examples.

This chapter includes the following sections:

- Running Example Servers provides notes on starting pre-installed applications
- DICOM Programming Examples section shows how simple DICOM related tasks are performed.
- Common Services Examples section covers use of the DCF framework services.
- Advanced DICOM Programming Examples covers more complex server concepts.

For additional information, see also

- Chapter 8 – The DCF Development Environment,
- Chapter 13 – Deploying a DCF-based application.

4.1. Running Example Servers

Using the DCF Remote Service Interface to run the DCF tools and/or servers generally makes running these examples easier. Taking this approach allows convenient access to tools for starting and stopping DCF server processes, viewing log files, and controlling trace/debug settings. Alternately, you may also manually run these servers from a DCF Command Prompt. Three approaches are described below.

4.1.1. Using the Web Service Interface

This approach allows you to conveniently control DCF servers, configuration, logging, etc. from a web browser that has connectivity to the system running the servers.

In a Windows environment invocation is all handled for you by the “DCF Remote Service Interface” startup script:

```
Select “Start”  → “All Programs”  → “DICOM Connectivity Framework” →
“DCF Service Interface”
```

This runs an Apache web server in its own window and invokes the default browser client to display the “DCF Remote Service Interface”.

Alternately, if you need a manual approach to start this interface, then open a DCF command window, type “*run_apache.pl*”, and then use your favorite web browser to browse to “*localhost:8080*”, which will display the “DCF Remote Service Interface” page.

Once the “DCF Remote Service Interface” is available in your web browser, select “choose a configuration”, then from that page select a server to start. For instance, select “store_server_win32.cfg” to start a generic DICOM store server.

4.1.2. Using a DCF Command Prompt – w/Common Services

This approach produces the same result as above for starting a generic DICOM store server, but without using the web browser service interface. This example approach does use the DCF Common Services, which provide configuration and logging servers to support the application.

Open a DCF command prompt:

Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Command Prompt"

At the prompt, type:

```
perl -S dcfstart.pl -cfg %DCF_cfg%\systems\store_server_win32.cfg
```

This command runs a few utility scripts up front, then the configuration and logging servers, and finally dcf_store_scp. Use a similar command to start any of the other server configurations that are available.

At this point you may open another DCF command window to run other clients or servers, etc.

4.1.3. Using a DCF Command Prompt – w/Minimal Resources

This approach produces the same result as above for starting a generic DICOM store server, but without using the web browser service interface and without using the DCF Common Services servers.

Open a DCF command prompt:

Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Command Prompt"

To just run the `dcf_store_scp` without any other supporting servers, do the following:

A) Edit the file `%DCF_CFG%\apps\defaults\dcf_store_scp`

Two changes are required and are highlighted in the code snippet below:

- Look for the text shown below and change value of "`use_log_server`" to `FALSE` or `NO`.
- Add the attribute "`handle_external_shutdown_rq`" to `APC_a`, if it is not there. Set its value to `FALSE` or `NO`.

```
[ cpp_lib/LOG_a/outputs ]
[ cpp_lib/LOG_a/outputs/server_output_1 ]
type = LOGSERVER
use_log_server = FALSE

...

[ cpp_lib/APC_a ]
debug_flags = 0
handle_external_shutdown_rq = no
save_proc_cfg_in_cds = no
save_exit_status_in_proc_cfg = no
delete_proc_cfg = yes
```

B) Run the app and tell it not to use the CFGDB server (DCDS_Server); to do this from a DCF command prompt, type:

```
dcf_store_scp  -no_dcds
```

Use a similar process to start any of the other servers that are available.

4.2. DICOM Programming Examples

4.2.1. Reading a DICOM file and extracting an element from the header

The following complete application demonstrates loading a DICOM encoded file, and extracting a value for a particular element or attribute.

```
#include <iostream>
#include <DCS/DicomFileInput.h>

using namespace std;
```

```
//
// print a single dicom element from a file
//
int main( int argc, char** argv )
{
   try
   {
      if (argc != 3)
      {
         cerr <<
            << "use: print_element_value <dicom_filename> <element_tag as hhhh,hhhh>"
            << endl;
         return 1;
      }
      LBS::DCS::DicomFileInput dfi( *++argv );
      LBS::DCS::DicomDataSet ds;
      dfi.readDataSetNoPixels(ds);

      cout
         << ds.findElement( LBS::DCS::AttributeTag(*++argv) ).getValueAsString()
         << endl;
   }
   catch ( LBS::DCF::DCFException &e )
   {
      cerr << e << endl;
   }
   return 0;
}
```

This application can be built on any supported platform using the `dcfmake.pl` utility. You can use any standard C++ build tools, as long as you direct them to the required include and library files. For example, using GNU make on Linux build this application with the following makefile:

```
CCFLAGS=-g -Wall -DLINUX -I$(DCF_ROOT)/include -I/opt/omniORB-4.1.4/include \
      -D__OMNIORB4__ -D_REENTRANT -D__linux__ -D__OSVERSION__=2 -D__x86__

LDFLAGS= -g -Wall -L$(DCF_ROOT)/lib -L/opt/omniORB-4.0.7/lib \
        -lDCF_dcfcore -lDCF_dcfutil -lDCF_boost_regex \
       -lDCF_dcs -lomniORB4 -lomnithread -lpthread


all: print_element_value

print_element_value: print_element_value.o
        gcc $(LDFLAGS) -o print_element_value print_element_value.o

print_element_value.o: print_element_value.cpp
        gcc $(CCFLAGS) -c print_element_value.cpp
```

See *`$DCF_ROOT/devel/csrc/examples/print_element_value`* for the source code for this example.

Note: This example intentionally does not configure the DCF common services – i.e., there are no calls to LOG/CDS/APC adapter setup methods or to “Framework::initDefaultServices()”. This is to demonstrate that a very simple application can be created with no reliance on the larger DCF infrastructure.

4.2.2. Creating a DICOM file that contains image data and patient demographics

One of the first things a DICOM developer often needs to do is to create an image file. DICOM defines IODs – Information Object Descriptions – to represent images from different modalities. DICOM also defines the encoding rules for communicating those IODs. An Image IOD can be written to the file system for later use. Many systems store images to local mass storage as they are acquired from the image generation source. Later these images are sent to DICOM printers or archive devices.

The following code fragment shows how DCF objects are used to create an image and store it to the `DicomDataService`.

```
//
//  create an image object
//
DCS::DicomObject image_object( UID_SOPCLASSXRAYANGIO, DCS::DCMUID::makeUID() );
dicom_data_set& image_ds = image_object.getDataSet();

image_ds.insert( dicom_element( E_PATIENTS_NAME, "Doe^John" ) );
image_ds.insert( dicom_element( E_PATIENT_ID, "12345" ) );
image_ds.insert( dicom_element( E_PATIENTS_BIRTH_DATE, "19610517" ) );
image_ds.insert( dicom_element( E_STUDY_ID, "4455" ) );
image_ds.insert( dicom_element( E_SERIES_NUMBER, "6677" ) );
// ... add more demographic fields ...
//
// now add image header fields
image_ds.insert( dicom_element( E_SAMPLES_PER_PIXEL, (UINT16)1 ) );
image_ds.insert( dicom_element( E_PHOTOMETRIC_INTERPRETATION, "MONOCHROME2" ) );
image_ds.insert( dicom_element( E_ROWS, (UINT16)512 ) );
image_ds.insert( dicom_element( E_COLUMNS, (UINT16)512 ) );
image_ds.insert( dicom_element( E_PIXEL_ASPECT_RATIO, "1\\1" ) );
image_ds.insert( dicom_element( E_BITS_ALLOCATED, (UINT16)8 ) );
image_ds.insert( dicom_element( E_BITS_STORED, (UINT16)8 ) );
image_ds.insert( dicom_element( E_HIGH_BIT, (UINT16)8 ) );
image_ds.insert( dicom_element( E_PIXEL_REPRESENTATION, (UINT16)0 ) );

// create some bogus pixel data, and add to data set
unsigned int rows = 512;
unsigned int cols = 512;
unsigned int size = rows * cols;
BYTE value;
BYTE *p_pixel_data = new BYTE[size];
BYTE *p_tmp = p_pixel_data;
for (unsigned int row = 0; row < rows; row++ )
{
    for (unsigned int col = 0; col < cols; col++ )
    {
        // really dumb way to draw a white grid on a black background
        value = (((row % 20) == 0) || ((col%20) == 0)) ? 0xff : 0;
        *p_tmp++ = value;
    }
}
image_ds.insert( dicom_element( E_PIXEL_DATA, DCM_VR_OW, size, (void*)p_pixel_data)
);

//
// display the image info in a log message
//
LOG_INFO_MSG << "Image created by combo_example:\n"
          << image_object << endl;


//
// save to local storage
```

```
//
DicomPersistentObjectDescriptor image_dpod;

LOG_INFO_MSG << "saving image to storage" << endl;

DDS::DicomDataService::instance()->storeObject( image_object, image_dpod );

LOG_INFO_MSG << "done: stored @" << image_dpod << endl;
```

The reference implementation of the `DicomDataService` adapter saves the data set as a DICOM format file. An alternate implementation can be provided which stores the data using some other format or mechanism. You could also use the `DicomFileOutput` class directly.

This code was taken from the C++ *`combo_example`*, which shows a combination of basic DICOM tasks being performed by a demonstration program. The full source for the program can be found in *`$DCF_ROOT/devel/csrc/examples/ex_combo`*.

4.2.3. Using the C++ StoreClient

4.2.3.1. Creating a job from DicomPersistentObjectDescriptors

The easiest way for a C++ application to send images or other DICOM objects to a storage service class provider is to use the `StoreClient` object. The `StoreClient` object is a high level class that sends a DICOM object to an SCP. The job, i.e., host, port, AE title, and location of DICOM object, should be described by a `StoreJobDescription` object.

```
//
//  initialize the StoreJobDescription
//
DSS::StoreJobDescrption job;
job.serverAddress( “archive:3004:StoreSCP” );
job.clientAddress( “StoreSCU” );
DicomPersistentObjectDescriptor dpod("","", “/tmp/test.dcm”, UID_TRANSFERLITTLEENDIAN);
DSS::StoreObjectInfo soi( dpod );
job.addObject( soi );

//
//  create the StoreClient object, and submit the job!
//
DSS::StoreClient client;
DSS::StoreJobStatus status;
client.submitStoreJob( job, status );
```

4.2.3.2. Using C++ StoreClient to C-Store DicomDataSets in memory

4.2.3.2.1. Use Store SCU directly

The problem is that you give up the fairly complex functionality that StoreClient uses to come up with the list of RequestedPresentationContext objects as well as a large quantity of job processing and status reporting logic.

4.2.3.2.2. Create a "special" DPOD (DicomPersistentObjectDescriptor) and use StoreClient

The DicomPersistentObjectDescriptor is a generic way to describe a DicomDataSet that can be retrieved using the DicomDataService::loadObject method. The special DPOD object will describe your in-memory dataset. You then add functions to your DicomDataService adapter (DDS_a) implementation to add/remove the dataset from the cache and override the "loadObject()"

method of DicomDataService to return the cached dataset. StoreClient calls loadObject for each instance in the StoreJobDescription.

We have described this approach here in detail and included source code to demonstrate what you need to do.

You add two new methods to DicomDataService_a.h:

```
virtual LBS::DDS::DicomPersistentObjectDescriptor addDataSetToCache( const
LBS::DCS::DicomDataSet& dds )
     throw (LBS::DDS::DDSException);

virtual void removeDataSetFromCache( const LBS::DDS::DicomPersistentObjectDescriptor&
dpod)
     throw (LBS::DDS::DDSException);
```

You will also need to add a private std::map to hold the cache:

```
    std::map< std::string, LBS::DCS::DicomDataSet > ds_cache_;
```

Their implementations would look like this:

```
DicomPersistentObjectDescriptor DicomDataService_a::addDataSetToCache(const
DicomDataSet& dds)
throw(DDSException)
{
    DicomElement e = dds.findElement( E_SOPINSTANCE_UID);
    std::string sop_instance_uid= e.getValueAsString();
    std::string persistent_id = sop_instance_uid;
    e = dds.findElement( E_SOPCLASS_UID);
    std::string sop_class_uid = e.getValueAsString();
    //use the persistent_id as the key in the cache
    ds_cache_[ persistent_id ] = dds;
    //here we set a special persistent_info_ of "memory_ds"
    DicomPersistentObjectDescriptor dpod( sop_class_uid, sop_instance_uid,
persistent_id, "memory_ds");
    return dpod;
}

void DicomDataService_a::removeDataSetFromCache ( const DicomPersistentObjectDescriptor&
dpod)
throw(DDSException)
{
    //use the persistent_id in the dpod as the key for deleting from the cache
    ds_cache_.erase(dpod.persistent_id_);
}
```

We also need to modify the private loadObject method in DicomDataService_a to use the special DPOD cache we created:

```
LBS::DCS::DicomObject* loadObject(
     const LBS::DDS::DicomPersistentObjectDescriptor& dpod,
     const LBS::CDS::CFGGroup* p_filter_cfg,
     bool f_read_pixel_data )
     throw ( LBS::DDS::DDSException );

DicomObject *DicomDataService_a::loadObject(
        const DicomPersistentObjectDescriptor& dpod,
        const CFGGroup* p_filter_cfg,
        bool f_read_pixel_data )
    throw ( DDSException )
{
   DicomInput* p_source = NULL;
   DicomObject* p_obj = NULL;
   if ( ! dpod.persistent_info_.compare( "memory_ds"))
   {
       std::map<std::string,LBS::DCS::DicomDataSet>::const_iterator itr =
ds_cache_.find(dpod.persistent_id_);
       if (itr != ds_cache_.end() )
       {
           DicomDataSet dds = (*itr).second;
           //StoreClient will delete this pointer
          return new DicomObject(dds);
       }
   }

..... existing code
```

This example code is using the SOP instance UID as the key in a std::map, so you cannot put the same dataset in the cache twice, and instead you might want to generate a unique key for each DicomDataSet you insert. You also might want to change the map to use DicomDataSet pointers as the value instead of DicomDataSet objects for efficiency.

You should also note that LBS::DSS::StoreClient will delete the DicomPersistentObjectDescriptor* returned from loadObject(...), so you should not delete it anywhere.

Then, to test these modifications, you could modify dcf_store_scu.cpp to use this new cache instead of loading them directly from the file system.

```
while (optind < argc)
{
   LBS::DCS::DicomDataSet ds;
   LBS::DCS::DicomFileInput* p_dfi = new LBS::DCS::DicomFileInput( argv[optind] );
   p_dfi->reference();
   p_dfi->readDataSet( ds );
   p_dfi->dereference();

    //notice we have to cast to our DDS implementation pointer to DDS_a*
    LBS::DDS_a::DicomDataService_a* p_dds_a = (LBS::DDS_a::DicomDataService_a*)
DicomDataService::instance();
    DicomPersistentObjectDescriptor dpod = p_dds_a->addDataSetToCache(ds);
    //need code to delete dpods somewhere in this example
    StoreObjectInfo soi( dpod );
    job.addObject( soi );
    optind++;
}
```

4.2.4. Using the C++ PrintClient

To send images from a C++ program to a DICOM Printer or Print “service class provider”, use the `PrintClient` class. `PrintClient` provides a very high level interface to a DICOM print SCP. The application developer is removed from the process of negotiating an association, sending DIMSE messages, managing the complex relationships between objects in the normalized service classes, and handling printer and print job status notifications. The sheets of images that are to be printed are defined in an intuitive hierarchical structure. The `PrintClient` object handles the messy details of DICOM Print.

The `PrintJobDescription` object contains basic attributes of the job, such as the server address, and various job level options. Also included in the `PrintJobDescription` is a single

`PrintJobFilmSession` object. This corresponds to the DICOM film-session object. `PrintJobFilmSession` contains one or more `PrintJobFilmBox` objects. A `PrintJobFilmBox` corresponds to the DICOM film-box object, which represents a sheet or film to be printed. PrintJobFilmBox contains one or more `PrintJobImageBox` objects. A `PrintJobImageBox` corresponds to a DICOM image-box and represents a single image to be placed somewhere on the film. When the job has completed, a `PrintJobStatus` object is returned which summarizes the results of the print operation.

The `PrintClient` also supports a listener or notification interface. If the user provides an object that inherits from or "implements" the `PrintClientListener` interface, then notifications will be sent to that object as DICOM print-job or printer status values change.

```
//
// initialize the PrintJobDescription
//
PrintJobDescription job; // describes the job we want to do
PrintJobFilmSession film_session;
PrintJobFilmBox film_box;

job.serverAddress( print_server_address );
job.clientAddress( "DEMO");
job.requestPrintJobSOPClass( true );
job.pollPrintJob(true);
job.printJobPollRateSeconds(2);
job.jobTimeoutSeconds( 30 );


film_session.numberOfCopies("1");
film_session.printPriority("HIGH");
film_session.mediumType("BLUE FILM");
film_session.filmDestination("MAGAZINE");
film_session.filmSessionLabel("test");
film_session.memoryAllocation("0");
film_session.ownerId("DCF");

film_box.imageDisplayFormat( "STANDARD\\1,1" );
film_box.filmOrientation("PORTRAIT");
film_box.filmSizeId("14INX17IN");
film_box.magnificationType("NONE");
film_box.smoothingType("NONE");
film_box.borderDensity("0");
film_box.emptyImageDensity("0");
film_box.minDensity(0);
film_box.maxDensity(280);
film_box.trim("YES");
film_box.configurationInformation("NONE");
film_box.illumination(0);
film_box.reflectedAmbientLight(0);
film_box.requestedResolutionId("HIGH");

PrintJobImageBox image_box;
image_box.imagePosition(1);
image_box.polarity("NORMAL");
image_box.magnificationType("NONE");
image_box.smoothingType("NONE");
image_box.configurationInformation("NONE");
image_box.requestedImageSize("0");
image_box.reqdDecimatecropBehavior("DECIMATE");
image_box.imageDPOD( image_dpod );

film_box.addImageBox( image_box );
film_session.addFilmBox( film_box );
job.filmSession( film_session );
```

```
//
// create the PrintClient object, and submit the job!
//
DPS::PrintClient client;
DPS::PrintJobStatus print_job_status("1"); // id of this job is 1


LOG_INFO_MSG << "submitting print job:\n" << job << endl;
client.submitPrintJob( job, (PrintClientListener*)0, print_job_status );
LOG_INFO_MSG << "print_job_status after completion:\n" << print_job_status << endl;
```

This code was taken from the C++ *ex_combo*, which shows a combination of basic DICOM tasks being performed by a demonstration program. The full source for the program can be found in *$DCF_ROOT/devel/csrc/examples/ex_combo*.

4.2.5. Media Storage Application Profiles – DICOMDIR files

Currently, File Set Creator (FSC), File Set Reader (FSR), and File Set Updater (FSU) functionality is provided by the classes `DicomDir`, `DicomDirectoryRecord`, and various DirectoryRecord subclasses. These classes are all in the DSS (DICOM Store Services) library component.

The `DicomDir` class is used to provide access to DICOM Directories which are defined as part of the media storage specifications (DICOM chapters: 10, 11, & 12). See Appendix B: Bibliography - The DICOM Standard.

In its persistent form, a DICOM directory (usually in a file called "DICOMDIR") contains some general file-set and directory information attributes, followed by a sequence of directory records.

4.2.5.1. Example – Creating a DICOMDIR

Create a DICOMDIR which references two images for the same patient/study/series and save it to a file.

A CD-R might be created from all files in the directory */tmp/cdrom-image*. That directory would contain the following files and subdirectories:

DICOMDIR - written by this example
PATIENT-A/STUDY-1/SERIES-2/IMAGE-1 - chapter 10 image file copied here
PATIENT-A/STUDY-1/SERIES-2/IMAGE-2 - chapter 10 image file copied here

```
DicomDir dicom_dir;

dicom_dir.fileSetId("EXAMPLE1");

PatientDirectoryRecord& patient_dir = PatientDirectoryRecord::create( dicom_dir );
patient_dir.patientName("patientA");
patient_dir.patientId("12345");

StudyDirectoryRecord& study_dir = StudyDirectoryRecord::create( patient_dir );
study_dir.studyInstanceUid("1.2.3.4");
study_dir.studyId("1");

SeriesDirectoryRecord& series_dir = SeriesDirectoryRecord::create( study_dir );
series_dir.modality("MR");
series_dir.seriesNumber("2");
```

```
ImageDirectoryRecord& image_dir_1 = ImageDirectoryRecord::create( series_dir );
image_dir_1.referencedFileId("PATIENT-A\STUDY-1\SERIES-2\IMAGE-1");
image_dir_1.referencedSopsclassUidInFile( UID_SOPCLASSMR );
image_dir_1.referencedSopinstanceUidInFile( "1.2.3.1" );

ImageDirectoryRecord& image_dir_2 = ImageDirectoryRecord::create( series_dir );
image_dir_2.referencedFileId("PATIENT-A\STUDY-1\SERIES-2\IMAGE-2");
image_dir_2.referencedSopsclassUidInFile( UID_SOPCLASSMR );
image_dir_2.referencedSopinstanceUidInFile( "1.2.3.2" );

dicom_dir.save("/tmp/cdrom-image/DICOMDIR");
```

4.2.5.2. Example – Adding to a DICOMDIR

Create a DICOMDIR, and add patient/study/series records as needed from multiple image files. The example shows a function that reads a data set from an image file, and updates the `DicomDir` object appropriately. This might be called for example, once for each image filename on a command line.

The `getXXDirRecord()` methods search for matching records and create new ones if a match is not found. Elements are copied from the image data set into the various record types according to configuration settings.

```
addImageToDirectory( DicomDir& dir, const string& image_fname )
{
      DicomDataSet image_ds;
      DicomFileInput in( image_fname );
      in.readDataSet( image_ds );

      DicomDirectoryRecord& root = dir.getRootDirectory();
      PatientDirectoryRecord& patient = PatientDirectoryRecord::find(
                  root,
                  image_ds,
                  true,
                  true );
      StudyDirectoryRecord& study = StudyDirectoryRecord::find(
                  patient,
                  image_ds,
                  true,
                  true );
      SeriesDirectoryRecord& series = SeriesDirectoryRecord::find(
                  study,
                  image_ds,
                  true,
                  true );
      if ( ImageDirectoryRecord::exists( series, image_ds ) )
      {
         throw DCSException("a matching image record already exists under that series");
      }
      else
      {
         ImageDirectoryRecord& image = ImageDirectoryRecord::create(
                                                  series, image_ds, true );
         image.referencedFileId( image_fname );
      }
}
```

4.2.5.3. Example – Reading a DICOMDIR

Read a DICOMDIR from a file, and traverse the contents.

Note: this produces similar output to the code "cout << dir;" but the method of descending into the directory record objects is illustrated here.

```
DicomDir dir("/cdrom/DICOMDIR");
DicomDirectoryRecord& root = dir.getRootDirectory();
DicomDirectoryRecordPtrList& root_records = root.getDirEntries();
DicomDirectoryRecordPtrList::iterator itr = root_records.begin();
string indent;
while ( itr != root_records.end() )
{
     DicomDirectoryRecord* p_dirrec = *itr++;
     displayDirRecord( *p_dirrec, indent );
}

void displayDirRecord( DicomDirectoryRecord& dirrec, string& indent )
{
     cout << indent << "record type is: " << dirrec.directoryRecordType() << endl;
     cout << indent << dirrec << endl;
     string new_indent(indent);
     new_indent += "\t";

     DicomDirectoryRecordPtrList& lower_records = dirrec.getDirEntries();

     DicomDirectoryRecordPtrList::iterator itr = lower_records.begin();
     while ( itr != lower_records.end() )
     {
         DicomDirectoryRecord* p_child = *itr++;
         displayDirRecord( *p_child, new_indent );
     }
}
```

See the online class documentation for `DSS::DicomDir` for additional information about DICOM media storage.

4.3. Deploying a Simple Standalone DCF C++ Application

The following procedure shows a simple method of deploying a DCF C++ application to a Windows host. The application (.exe), its required libraries (.dll), and configuration data can be installed into a single directory on the target system. The application can then be run from the installation directory.

We'll show the process of creating the install directory on your DCF developer box (the host with the DCF toolkit installed). Once created, that install directory can then be copied to the target using any number of methods: zip on your DCF developer box, and unzip on the target; or perhaps burn this directory to a CD-ROM and then run directly from the CD on the target.

This example shows deploying the C++ `dcf_filter` example and `dcf_dump`. The process would be modified somewhat for your own application.

Perform the following steps:

1. Open a DCF command window:
   `Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Command Prompt"`
2. Create the test install directory:
   Note: You could paste this text into a batch file and run it to automate this process.

```
    REM ###
    REM ### create install dir
    REM ###
    mkdir DCF_test_cpp_install
    cd DCF_test_cpp_install

    REM ###
    REM ### copy required library files from %DCF_LIB% (../DCF/lib)
    REM ###
    copy %DCF_LIB%\DCF_DCFCore.dll
    copy %DCF_LIB%\DCF_ljpeg12.dll
    copy %DCF_LIB%\DCF_ljpeg16.dll
    copy %DCF_LIB%\DCF_ljpeg8.dll
    copy %DCF_LIB%\DCF_APC_a.dll
    copy %DCF_LIB%\DCF_CDS_a.dll
    copy %DCF_LIB%\DCF_LOG_a.dll
    copy %DCF_LIB%\DCF_boost_regex.dll
    copy %DCF_LIB%\DCF_DAPC.dll
    copy %DCF_LIB%\DCF_DCDS.dll
    copy %DCF_LIB%\DCF_DLOG.dll
    copy %DCF_LIB%\DCF_DCS.dll
    copy %DCF_LIB%\DCF_DCFUtil.dll
    copy %DCF_LIB%\DCF_TSCW.dll
    copy %DCF_LIB%\DCF_TSCWIJG.dll
    copy %DCF_LIB%\DCF_TSCWJasper.dll
    REM ### The Aware wrapper dll is needed only if using Aware's JPEG libraries.
    REM ### Note the actual Aware JPEG library (awj2k.dll) must be purchased separately
    copy %DCF_LIB%\DCF_TSCWAware.dll

    REM ### copy required library files from %DCF_BIN% (../DCF/bin).
    REM ### These may exist in other places on the system, but copies
    REM ### are put here during DCF toolkit install for convenience,
    REM ### (Note omniORB dlls may not be required depending on the
    REM ### application and your DCF version)

    REM ### If you are building from a DCF VisualStudio8.x .NET toolkit:
    copy %DCF_BIN%\msvcp80.dll
    copy %DCF_BIN%\msvcr80.dll

    REM ### Note that the filenames may differ somewhat from what is specified here.
    copy %DCF_BIN%\omniORB414_rt.dll
    copy %DCF_BIN%\omniDynamic414_rt.dll
    copy %DCF_BIN%\omnithread34_rt.dll

    REM ###
    REM ### Copy the application that you want - for example,
    REM ### include both the C++ dcf_filter example, and the dcf_dump
    REM ### utilities.
    REM ###
    copy %DCF_BIN%\dcf_dump.exe
    copy %DCF_BIN%\dcf_filter.exe

    REM ###
    REM ### Create a minimal configuration directory.
    REM ###
    mkdir cfg
    mkdir cfg\apps
    mkdir cfg\apps\defaults
    mkdir cfg\procs

    REM ###
    REM ### Copy the license configuration file, and the application configs
    REM ### for the installed programs.
    REM ###
    copy %DCF_CFG%\systeminfo cfg\systeminfo
    copy %DCF_CFG%\apps\defaults\dcf_filter cfg\apps\defaults
```

```
     copy %DCF_CFG%\apps\defaults\dcf_dump cfg\apps\defaults
```

3. Create the media by which you will deliver the install directory
4. On the target machine do the following:
   a) Unpack, copy, or otherwise make the DCF app install directory available. For example, copy or unzip to `C:\temp\DCF`
   b) From a command window, go to the install directory. For example, use `C:\temp\DCF`.
      `cd C:\temp\DCF`
   c) Set environment vars and run your apps (you could put these steps in a *`run_app.bat`* file).

```
    set DCF_CFG=C:\temp\DCF\cfg
    set DCF_LIB=C:\temp\DCF
    set DCF_TMP=C:\temp\DCF
    ### display input image (choose a DICOM file in the line below)
    dcf_dump \temp\test.dcm
```

Note that currently, all DCF standard C++ dll's are prefixed with "DCF_".

4.4. Common Services Programming Examples

4.4.1. C++ "hello world" Example Application Using the DCF

To demonstrate some of the capabilities of the DCF, you can create and run the most basic of code examples: the "Hello World" program. The DCF "Hello World" program demo will make use of the DCF development tools, as well as the common services APIs and implementations.

Change to the source directory for the C++ "hello world" example, then build and execute the example application:

```
cd $DCF_ROOT/devel/csrc/examples/ex_hello_world
```

Build the application, if needed.

```
perl -S dcfmake.pl
```

Run the application.

```
ex_hello_world -no_dcds
```

The "`-no_dcds`" option allows the program to access configuration data directly from the filesystem.

From your web browser, select "View Log Files" from the DCF Remote Service Interface. Select the log file for the *`ex_hello_world`* application and view the output.

To create the *`ex_hello_world`* application the following steps were followed:

1. Create a directory for the application
2. Create a component information file for the application
3. Create the source code for the application
4. Build the application
5. Update the configuration database

1. Create a directory for the new application component – in the DCF, every application or library is a component, and has its own source directory.
   ```
   mkdir $DCF_USER_ROOT/devel/csrc/examples/ex_hello_world
   ```

2. Create a component information file in that directory. This file must be called "*cinfo.cfg*". For this example it contains the following:

```
#==============================================================================
# static information common to all instances of the ex_hello_world component
#==============================================================================
[ component_info ]
name = ex_hello_world
type = cpp_app
category = examples
docfile = ex_hello_world.cpp
description = Example program that uses DCF common services to implement the classic
first application

[ build_info ]
gen_app_cfg = yes
bin_dir = .

[ debug_controls ]
debug_flag = df_TEST1, 0x10000, place holder for test 1 debug setting
debug_flag = df_TEST2, 0x20000, Do something cool

[ required_components ]
component = cpp_lib_pkg/DCFCore
component = cpp_lib/DCFUtil
component = cpp_lib/LOG_a
component = cpp_lib/APC_a
component = cpp_lib/CDS_a
component = idl_lib/DCDS

#==============================================================================
# per-instance information for the ex_hello_world component
#==============================================================================
[ ex_hello_world ]
debug_flags = 0x00000

#==============================================================================
# The following sections allow the customization of the generated default
# application configuration.
# After the application configuration is created,
# selected library component configuration settings can be overridden.
# Note that this affects the settings for that library only within the context
# of this application.
#==============================================================================
[ lib_cfg_overrides ]

[ lib_cfg_overrides/LOG_a ]
use_log_server = FALSE
```

The file is in the DCF configuration file format, which provides for attributes, groups, and nested groups.

Note: The easiest way to create the cinfo.cfg file for your application or library is to copy one from a similar component, then edit as needed.

Explanation:

The first group `[ component_info ]` describes basic attributes of the component. The `name` "`ex_hello_world`" is simply the file name. The component `type` is "`cpp_app`" which indicates a C++ application.

Note: you can use *`dcfmake.pl`* to create applications in any directory, as long as you create a *`cinfo.cfg`* file in that directory.

The group `[ debug_controls ]` is where the developer can add support for conditional logging or other behavior specific to this component. Debug controls that are defined here can be accessed via the web interface.

The `[ required_components ]` group specifies the components needed by this application.

The `[ex_hello_world ]` group contains the instance configuration for the component. This data is used directly in the example code.

3. Create the application source code

   For this example, the file is called “*`ex_hello_world.cpp`*”.

```
#include <iostream>

#include <DCF/Framework.h>

#include "ex_hello_worldCInfoL.h"
using namespace LBS::ex_hello_world;
using namespace LBS;
using namespace LBS::DCF;
using namespace std;

int main( int argc, char **argv )
{
   int status;

   try
   {
      DCF::Framework::initDefaultServices( argc, argv );

      LOG_INFO_MSG << "Hello World!" << endl;

      LOG_DEBUG_MSG(df_TEST2) << "only print this if df_TEST2 is set" << endl;
      //
      // clean up
      //
      APC::AppControl::instance()->shutdown(0);

      status = APC::AppControl::instance()->exitStatus();
   }
   catch (DCF::DCFException& e)
   {
      LOG_ERROR_MSG(-1) << e << endl;
      status = -1;
   }

   if (status == 0)
   {
      cerr << "Test succeeded. See the generated log file for more information" << endl;
   }
   else
   {
      cerr << "Test failed. See the generated log file for more information" << endl;
   }
   return(status);
}
```

4. Build the application.

   To build the application, simply type the command

   ```
   perl –S dcfmake.pl
   ```

Invoking `dcfmake.pl` will perform the following steps for this example:

Read the `cinfo.cfg` file in the current working directory.
Read the component configuration for each `required component` in the `cinfo.cfg`. Component configurations come from the `$DCF_USER_ROOT/devel/cfggen/components` directory. That data was created when `dcfmake.pl` built those components.
Recursively read component configurations for other required components.
Generate the component configuration for this component. This data is written to the file `$DCF_USER_ROOT/devel/cfggen/components/cpp_app/ex_hello_world`
Generate the application configuration for this component. This data is written to the file `$DCF_USER_ROOT/devel/cfggen/apps/defaults/ex_hello_world`
Generate the `ex_hello_worldCInfo.cpp` source file in the current directory. The `CINFO` class contains the debug-flag mask constants, as well as code to initialize and update the debug flags value from the CDS database. `CINFO` also provides convenience mechanisms for getting the instance configuration group for the component within a particular application.
Generate the `ex_hello_worldCInfo.h` source file in the current directory. It contains the component specific debug flag constants.
Generate the `ex_hello_worldCInfoL.h` source file in the current directory. It contains various `LOG` macros that simplify checking debug flag settings, and provide message header fields that remain constant for the component. See notes below in section 4.4.2.
Generate the make file. To avoid confusion with a handcrafted makefile, the file is called `makefile.dcf`.
Invoke "`make -f makefile.dcf`". Any arguments given to `dcfmake.pl` are forwarded to make. After the make completes, the generated makefile is removed. You can have `dcfmake.pl` leave the generated file by using the "`-keep`" option.

5. Update the configuration data service repository.

   The developer can determine when to deploy any newly created or edited configuration data. This can be useful if you are testing with non-default configurations and do not want the fact that you have rebuilt something to affect your working configuration files. To update the data, execute the command:

   ```
   perl -S update_cds.pl
   ```

   This will copy all files from the temporary areas `$DCF_USER_ROOT/devel/cfggen` and `$DCF_USER_ROOT/devel/cfgsrc` to the working area: `$DCF_USER_ROOT/cfg`. As the files are copied various macros are expanded, so, for example, the files in the working config can have the correct port numbers, path names, etc.

The application is now ready to run!

4.4.2. Using the LOG interface – Logging from C++ programs

To access DCF logging facilities from C++, use the macros that are generated in the <component_name>`CInfoL.h` (Component Information – Local) file. These include the macros:

```
LOG_INFO_MSG
LOG_ERROR_MSG
LOG_DEBUG_MSG
```

First, the `LOG` adapter must be initialized. Normally, all of the common services are installed at once, during application initialization. This can be done with either the lines:

```
        LBS::LOG_a::LogClient_a::setup( argc, argv);
        LBS::CDS_a::CFGDB_a::setup( argc, argv );
        LBS::APC_a::AppControl_a::setup( argc, argv );
        LBS::LOG_a::LogClient_a::setup( argc, argv );
```

or

```
        LBS::DCF::Framework::initDefaultServices( argc, argv );
```

All of the `LOG` macros establish a message header, and then evaluate to a C++ standard `ostream` (output stream) object reference, so the interface is similar to using the familiar `cout` or `cerr` streams. For example:

```
        LOG_INFO_MSG << "this message will always be printed" << endl;
```

The `endl` is significant. When the `ostream` is flushed (using the standard `endl` or `flush` manipulator), a message boundary is established, and all text between the beginning of the message and the flush is logged with a single message header. This is preferable to having either a continuous stream of output text, or adding a header to each line of text.

To print debug messages, use:

```
        LOG_DEBUG_MSG( df_SOME_DEBUG_SETTING ) << "the value of x is: " << x
           << " the value of y in hex is: " << hex << y << endl;
```

The previous message will only be logged if the `df_SOME_DEBUG_SETTING` bit is set in the debug flags for the component that contains the code. Note the use of the standard `ostream` insertion (`<<`) operators, and the various manipulators (`hex`, `endl`).

Error messages are logged with:

```
        LOG_ERROR_MSG(-1) << "an exception was caught: " << exception << endl;
```

In that example, "exception" is some object that provides an `ostream` print method, i.e.,

```
        friend ostream& operator<<( ostream&, SomeExceptionClass& );
```

4.4.3. Using the CDS interface

See language-specific class documentation for `CDS.CFGGroup`, `CDS.CFGAttribute`, `CDS.CFGDB`, and `CDS_a.CFGDB_a`.

4.4.4. Using the APC interface

See language-specific class documentation for `APC.AppControl` and `APC_a.AppControl_a`.

4.5. Advanced DICOM Programming Examples

4.5.1. Writing a customized storage SCP

A common application of the DICOM protocol is in creating an image archive. An OEM may have special requirements for how images and patient information are stored in a database. The DCF provides APIs that are structured such that the OEM can easily customize the handling of image or other DICOM datasets without needing to deal with the mechanics of negotiating associations, handling sockets, PDUs or DIMSE messages.

The `DicomDataService` interface provides the mechanism for customizing the handling of DICOM image handling. Generic DCF protocol handling objects such as `StoreSCP`, `QRSCP` (Query Retrieve), `MWLSCP` (Modality Worklist) invoke `DicomDataService` methods to access the local storage facilities. The reference implementation adapter for the `DicomDataService` interface stores objects in the file system and provides minimal searching capabilities to support testing. Other implementations or adapters can be written that behave differently.

The directory *$DCF_ROOT/devel/csrc/dcf_store_scp* shows a simple storage server that sends incoming DICOM objects to the file system using the default `DicomDataService_a` (DICOM data service adapter) in *$DCF_ROOT/devel/csrc/DDS_a*. By installing a particular `DicomDataService_a`, all incoming DICOM images are passed to the `storeObject()` method defined in that class.

The source file *dcf_store_scp.cpp* contains the function `main()` which installs the `DicomDataService_a` adapter, and enters the loop which waits for incoming DICOM associations.

```
int main( int argc, char *argv[], char *[] )
{
   int status = -1;
   try
   {
      //
      // Quick check for -h or -help option
      //
      for ( int i=0; i<argc; i++ )
      {
         string arg = argv[i];
         if (arg.find("-h") == 0)
         {
            usage();
            exit( 0 );
         }
      }

      //
      //  Setup adapters.
      //
      AppControl_a::setupORB( argc, argv );
      CFGDB_a::setup( argc, argv );
      AppControl_a::setup( argc, argv );
      LOG_a::LOGClient_a::setup( argc, argv );
      DPS::OEMPrinterInfo_a::setup( argc, argv );
      DDS_a::DicomDataService_a::setup( argc, argv );

      //
      // run Event loop returns in multi-threaded mode. The CORBA
      // services are enabled, and running in their own threads.
      //
      AppControl::instance()->runEventLoop( false );

      //
      // create an Association Manager
      //
      AssociationManager amgr;

      //
      // create a StoreServer object. It will register with the
      // AssociationManager and receive notifications when an
      // association is being started.
      //
      StoreServer store_server( amgr );
      // also create a Verification server object. It will handle the
      // Verification SOP class.
      //
      VerificationServer verification_server( amgr );

      //
      // start the Association Manager. It will wait for incoming
      // connections.
      //
      amgr.run();

      status = 0;
   }

   catch (IOTimeoutException& e )
   {
      LOG_FATAL_ERROR_MSG(-1) << (DCFException&)e << endl; // cast stops solaris CC error
      status = 1;
   }
   catch (IOException& e )
   {
      LOG_FATAL_ERROR_MSG(-1) << (DCFException&)e << endl;
      status = 2;
   }
   catch (DCFException& e )
   {
      LOG_FATAL_ERROR_MSG(-1) << e << endl;
      status = 3;
   }
   catch (std::exception& e )
   {
      LOG_FATAL_ERROR_MSG(-1) << "dcf_store_scp: caught unexpected C++ exception:\n" <<
e.what() << endl;
      return( 4 );
   }
   catch (...)
   {
      LOG_FATAL_ERROR_MSG(-1) << "dcf_store_scp: caught unknown exception:\n" << endl;
      return( 5 );
   }

   AppControl *p_appctrl = AppControl::instance();
   if (p_appctrl)
   {
      p_appctrl->shutdown( status );
      status = p_appctrl->exitStatus(); // may not be what we just gave it
   }

   LOG_INFO_MSG << "dcf_print_scp exiting with status:" << status << endl;
   return status;
}
```

The `DicomDataService` adapter class (`DicomDataService_a`) defines methods used by various DICOM servers to access mass storage. The method `storeObject()` is invoked each time a C-Store-Request is received on an association.

`DicomDataService` uses the singleton pattern to allow a single implementation object to provide services for the process. When the abstract base class method `DicomDataService::instance()` is invoked, the object returned is the sub-class or concrete implementation. The code throughout DCF that uses the returned instance does not know or care what instance has been installed.

The declaration of the `storeObject()` method is contained in the header file *`DicomDataService_a.h`*

```
/**
* Store a DICOM object, and return the DicomPersistentObjectDescriptor which
* references that object. The initial reference count for the object will be
* one.
* @param obj reference to LBS::DCS::DicomObject whose data will be stored.
* @param dpod_ret reference to persistent object descriptor which will be
* filled in. The object may be retrieved later, using this descriptor.
* @throw LBS::DCS::IONoSpaceException if there is insufficient mass storage to save
* the object
* @throw DDSException if any other error occurs
*/
virtual void storeObject(
      const LBS::DCS::DicomAssociation& association,
      const LBS::DCS::DimseMessage& c_store_rq,
      DicomPersistentObjectDescriptor& dpod_ret )
   throw ( LBS::DCS::IONoSpaceException, DDSException );
```

The implementation of the `storeObject()` method is in the file *`DicomDataService_a.cpp`*

```
void DicomDataService_a::storeObject(
      const LBS::DCS::DicomAssociation& association,
      const LBS::DCS::DimseMessage& c_store_rq,
      DicomPersistentObjectDescriptor& dpod_ret )
   throw ( LBS::DCS::IONoSpaceException, DDSException )
{
   LOG_DEBUG_MSG( df_SHOW_GENERAL_FLOW ) << "storeObject:" << obj << endl;

   try
   {
      LBS::DCS::DicomObject obj( c_store_rq.data() );

      string sop_instance_uid =
             f_make_new_uids_ ? DCMUID::makeUID() : obj.sopinstanceUid();

      string persistent_id = image_directory_;
      persistent_id += "/";
      persistent_id += sop_instance_uid;
      persistent_id += ".dcm";

      dpod_ret.sopinstanceUid( sop_instance_uid );
      dpod_ret.persistentId( persistent_id );
      dpod_ret.persistentInfo( ts_uid_ );

      DicomFileOutput dfo( dpod_ret.persistentId(), dpod_ret.persistentInfo() );
      dfo.writeDataSet( obj.getDataSet() );
   }
   catch (LBS::DCS::IONoSpaceException)
   {
      throw;
   }
   catch (LBS::DCF::DCFException& e)
   {
      ostringstream os;
      os << "storeObject failed:\n" << e;
      throw DDSException(os.str());
   }
}
```

The example `storeObject()` method performs the following steps:

- Print a log message, if the debug flag `DF_SHOW_GENERAL_FLOW` is set.

- Store the object to a DICOM format file using the `DicomFileOutput` class.

4.5.2. Writing a customized query retrieve SCP

As with the storage SCP in the previous example, customizing the standard query retrieve SCP can also be done by providing a modified implementation of the `LBS::DDS::DicomDataService` interface.

While the DCF handles the complexity of association negotiation, receiving and sending DIMSE messages and PDUs for multiple concurrent associations, the OEM need only provide an implementation of the `findObjects()` or `findObjectsForTransfer()` methods in `DicomDataService`.

The `QRSCP` (Query Retrieve Service Class Provider) invokes `DicomDataService::instance()->findObjects()` when it receives a C-Find request. The `findObjects()` method extracts the query attributes from the message, and performs the search, using the mechanisms appropriate for the local database. For each matching entry, a `DicomObject` is created and returned to `QRSCP` via the `DicomQueryListener::returnQueryResult()` callback method. `QRSCP` will form the appropriate pending C-Find-Response DIMSE message and queue it for transmission back to the SCU. When the matches are complete, `DicomQueryListener::queryComplete()` is called; `QRSCP` will then send a C-Find-Response with a final status back to the SCU.

If any exceptions are thrown by `findObjects()`, `QRSCP` will return an error status to the SCU.

The `QRSCP` invokes `DicomDataService::instance()->findObjectsForTransfer()` when it receives a C-Move or C-Get request. The `findObjectsForTransfer()` method performs a similar data base search, but only the storage information for the matched instances is returned – e.g., the filename of the matching images. The `QRSCP` then manages sending the objects to the appropriate destination using the Storage service class.

Below is an excerpt of the reference implementation of `DicomDataService_a`: *`$DCF_ROOT/devel/csrc/DDS_a/DicomDataService_a.cpp`*

```
/**
* Find objects that match the given query criterion, and return the location
* in storage for those objects. This method is normally called by a Query Retrieve
* SCP when it receives a C-Move-Request or C-Get-Request. The QRSCP will handle the
* Store sub-operations when it gets back the list of matching instances.
* @param association the current DICOM association. This contains information about the
* client AE, negotiated contexts, etc.
* @param transfer_request the c-move or c-get request dimse message which contains the
* query criterion. This object
* contains both the command dataset, and the "data" dataset, which contains the
* query identifier.
* @param results reference to list which will be filled in with the matching objects.
* Only the information on how to load the object from storage is returned.
* @throw DDSException if an error occurs
*/
void DicomDataService_a::findObjectsForTransfer(
      const LBS::DCS::DicomAssociation& association,
      const LBS::DCS::DimseMessage& transfer_request,
      DicomPersistentObjectDescriptor::List& results )
   throw ( DDSException )
{
   omni_mutex_lock lock( mutex_ );
   findObjects( association, transfer_request, &results, (DicomQueryListener*)0 );
}
```

```
void DicomDataService_a::findObjects(
      const LBS::DCS::DicomAssociation& association,
      const LBS::DCS::DimseMessage& c_find_request,
      DicomQueryListener* p_listener )
   throw ( DDSException )
{
   omni_mutex_lock lock( mutex_ );
   findObjects( association, c_find_request, (DicomPersistentObjectDescriptor::List*)0,
p_listener );
}

. . .

void DicomDataService_a::findObjects(
      const LBS::DCS::DicomAssociation& association,
      const LBS::DCS::DimseMessage& query,
      DicomPersistentObjectDescriptor::List* p_result_dpods,
      DicomQueryListener* p_listener    )
   throw ( DDSException )
{
   DicomObject* p_test_object = 0;
   try
   {
      LOG_DEBUG_MSG( df_SHOW_GENERAL_FLOW ) << "findObjects: query =\n" << query <<
endl;

      if ( DCF_DEBUG( df_FORCE_ERROR_ON_FIND_OBJECTS ) )
      {
         throw DDSException("simulating error condition during findObjects");
      }

      //
      // prepare the query identifier by making a writable copy of it, and then
      // removing the query-retrieve-level element
      //
      DicomDataSet query_dataset( query.data() );
      query_dataset.removeElement( E_QUERYRETRIEVE_LEVEL );

      //
      // search for matches in the list of stored objects.
      // This is a simple linear search using the compare() method
      // in DicomDataSet/Element/Sequence. Only the header portion
      // (everything up to 7FE0,0010) of each stored object is loaded
      // prior to comparing.
      //
      updateInstanceDPODList();
      DicomPersistentObjectDescriptor::List::const_iterator itr =
instance_dpod_list_.begin();
      while ( itr != instance_dpod_list_.end() )
      {
         const DicomPersistentObjectDescriptor& dpod = *itr++;

         p_test_object = loadObject( dpod , (LBS::CDS::CFGGroup*)0, false );

         if ( p_test_object->getDataSet().compare( query_dataset ) )
         {
            LOG_DEBUG_MSG( df_SHOW_GENERAL_FLOW )
               << "DicomDataService_a::findObjects: adding DPOD for DicomObject to
result set:\n:"
               << *p_test_object
                    << "\n" << dpod << endl;
            if (p_result_dpods)
            {
               p_result_dpods->push_back( dpod );
               delete p_test_object;
            }
            if (p_listener)
```

```
                {
                    p_listener->returnQueryResult( *p_test_object );
                    delete p_test_object;   // make sure listener has his own copy of this!
                }
                p_test_object = 0;
            }
        }

        if (p_listener)
        {
            p_listener->queryComplete( 0 );
        }
    }
    catch (LBS::DCF::DCFException& e)
    {
        delete p_test_object;
        if (p_listener)
        {
            try
            {
            p_listener->queryComplete( 2 );
            }
            catch (...)
            {
                LOG_ERROR_MSG(-1) << "DicomQueryListener threw unexpected exception" <<
endl;
            }
        }
        if ( p_result_dpods )
        {
            p_result_dpods->clear();
        }

        ostringstream os;
        os << "DicomDataService_a::findObjects failed:\n" << e;
        throw DDSException(os.str());
    }
}
```

The method `findObjects()` extracts the supported query attributes from the C-Find-Request message and creates a query. The attributes supported are defined in a `CFGGroup` for each level. This example forms the query as a `DicomDataSet`.

A local database is searched at the appropriate level, using the query created above. The `DicomDataSet::compare()` method is used to compare the query data set with data sets created from stored data at the selected level.
Note: this is not intended to show an efficient search algorithm!

Each matching data set is returned in the `DicomObjectPtrList`, which is simply an STL list of pointers to allocated objects of type `DicomObject`.

To create an SQL query from the C-Find-Request, you might do something like:

```
QRIdentifier qi( c_find_rq.data() );
string query = "SELECT * from PATIENT where ";
query += "patientId like \""
query += qi.patientId();
query += "\"";
```

To create a `DicomObject` corresponding to a matching SQL result:

```
//Let's say each result can be described as an array of strings, one per column in the
fetched result:
```

```
string    db_result[ n_columns ];

//The results as a list of pointers
DicomObjectPtrList result_list;

// create a DicomObject, and get a reference to the contained data set
DicomObject* p_result = new DicomObject();
DicomDataSet& dds = p_result->getDataSet();

// pull columns from known offsets in the result, and create elements
dds.insert( DicomElement( E_PATIENTS_NAME, db_result[0] ) );
dds.insert( DicomElement( E_PATIENT_ID, db_result[1] ) );
dds.insert( DicomElement( E_PATIENTS_BIRTH_DATE, db_result[2] ) );


// add the object to the list.
result_list.push_back( p_result );
```

See `$DCF_ROOT/devel/csrc/examples/ex_qr_scp_sql` for a more complete SQL example.

4.6. Using the C++ Modality Worklist examples

The dcf_mwl_scp can be run and controlled like dcf_qr_scp.
There is a startup configuration, $DCF_CFG/systems/mwl_server_unix.cfg,
and in the $DCF_CFG/apps/MWLSCP directory is where per AE Title
configuration files should go. It will use
LBS::DDS::DicomDataService::findObjects as its interface to a database.
The reference implementation file system based database uses
$DCF_CFG/test/dcf_mwl_scp/sample_objects.cfg as the index to locate
"Worklist items" The worklist items are DICOM files that contain a
Modality Worklist Scheduled Procedure Step (SPS).

dcf_mwl_scu uses the LBS::DIS::MWLSCU class. The dcf_mwl_scu and MWLSCU
class are very similar to the dcf_qr_scu and QRSCU class. You'll
notice that the MWLSCU::c_find method takes a
LBS::DIS::ModalityWorklistItem as an argument. This is a convenience
wrapper around a DicomDataSet, which lets you access DICOM elements in
the Modality Worklist Scheduled Procedure Step Sequence with get/set
methods instead of having to access the nested sequences contained in an
SPS.

4.7. DICOM compression transfer syntax support for C++

DCF C++ applications can handle DICOM datasets in any transfer syntax for non-pixel data operations provided that compression pass through mode is turned on (except for DICOM Deflated Little Endian Syntax and JPIP Transfer syntaxes).

DCF C++ applications can compress and decompress data sets in these encapsulated transfer syntaxes:

• 1.2.840.10008.1.2.4.5	RLE Lossless
• 1.2.840.10008.1.2.4.50	JPEG 8 bit lossy
• 1.2.840.10008.1.2.4.51	JPEG 12 bit lossy
• 1.2.840.10008.1.2.4.57	JPEG lossless

1.2.840.10008.1.2.4.70	JPEG lossless (predictor selection=1)
1.2.840.10008.1.2.4.90	JPEG-2000 lossless
1.2.840.10008.1.2.4.91	JPEG-2000 lossy

RLE Lossless transfer syntax is supported for compression of single frame data sets. RLE Lossless transfer syntax is supported for the decompression of single frame and multi-frame data sets.

Note: If you are using the Aware, Inc., JPEG library, that this does not support .57.

Look at the settings under the DCS section of an application configuration file or in a DCS component configuration file to see options that may be configured for compression.

5. Java Programming Examples

This section presents a variety of Java programming examples for common DICOM integration tasks. See *$DCF_ROOT/devel/jsrc/com/lbs/examples/* for the complete working source code for these and additional examples.

This chapter includes the following sections:

- DICOM Programming Examples section shows how simple DICOM related tasks are performed.
- Common Services Examples section covers use of the DCF framework services.
- Advanced DICOM Programming Examples covers some more complex server concepts.

For additional information, see also

- Chapter 8 – The DCF Development Environment,
- Chapter 13 – Deploying a DCF-based application.

5.1. Running Example Servers

Running the DCF tools and/or servers via the DCF Remote Service Interface generally makes running these examples easier. Taking this approach allows easy access to convenient tools for starting and stopping DCF server processes, viewing log files, and controlling trace/debug settings.

Start the Apache web server and open the DCF Remote Service Interface. In a Windows environment this is all handled for you by the startup script:

> *Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Service Interface"*

This command runs an Apache web server in its own window and invokes the default browser client to display the DCF home page, called the "DCF Remote Service Interface".

Alternately, if you prefer a manual approach, type "*run_apache.pl*" from a DCF command window, and then use your favorite web browser to browse to "*localhost:8080*", which will display the DCF home page.

5.2. Using the DCF with Java IDE tools

5.2.1. Using the DCF with Eclipse for Java

Notes for using Eclipse v3.0.1
(extracted from `README.win32.setup.txt` which is found on the install CD).

If you are using a different version of Eclipse, you may have to modify these instructions for your version of the Eclipse Platform.

To build a DCF example using Eclipse Platform version 3.0.1:

- Change to the directory of the example you wish to build, e.g., `ex_jdcf_HelloWorld`
- If you have changed the cinfo.cfg file, generate new `CINFO.java` and `LOG.java` java files by running *perl -S dcfmake.pl -g* from a DCF Command Prompt.
- Run eclipse from a DCF Command Prompt.
  Select
  Start → All Programs → DICOM Connectivity Framework → DCF Command

Prompt
(Note: this only needs to be done when first setting up the environment variables in the Eclipse run dialog. After the project is set up, Eclipse may be started normally, since the configuration will be saved.)

- From the File menu, select "*New...->Project*"; expand *Java*, select *Java Project*, and click *Next*.
- Enter a Project Name as usual, e.g., *DCF Java Examples*.
- Select "*Create project in external location*"; browse to or enter the complete path to the jsrc directory as the `Directory` option.
- Click next.
- Select the `Libraries` tab, click "*Add External JARS...*"; browse to *<install directory>\classes\LaurelBridge.jar*, select it and click *Open*.
- Back in the `Libraries` tab, select *Finish*.
- Expand the newly created project, and select your examples from the list, e.g., `com.lbs.examples.ex_jdcf_HelloWorld`.
- Expand your example and select the source file containing the main class, e.g., `ex_jdcf_HelloWorld.java`.
- Right-click on the source file, and choose "*Run->Run...*".
- Select and highlight `Java Application` under `Configurations:`.
- Click *New* (this should create a new configuration under `Java Application`).
- From the `Environment` tab, click *Select*.
- Click *Select all*, and click *Ok*.
- Click radio button "*Replace native environment with specified environment*".
- Add the necessary program arguments under the `Arguments` tab, e.g., "`-appcfg /apps/defaults/ex_jdcf_HelloWorld`".
- Click *Apply*.
- Make sure the DCF is running, and click *Run*. Your example application should run successfully.

5.3. DICOM Programming Examples

5.3.1. Using Java Print Element Value Program

Open a command window and change to the *devel/jsrc/com/lbs/examples/ex_jprint_element_value* directory under the DCF install directory, build (optional) using *dcfmake.pl*, and then execute the example application:

```
cd $DCF_USER_ROOT/devel/jsrc/com/lbs/examples/ex_jprint_element_value

# if you want to rebuild the application (you must have the appropriate
# version of Visual Studio (6, 7, or 8) installed, and in the path.
# (you can run the script vsvars.bat (7.x or 8.x or 9.x)
# from the visual studio directory, or perhaps run in the command window
# shortcut provided with studio 7, 8 or 9). **** Note for Java, this is only because
# dcfmake.pl uses the Visual Studio simply to process the "makefile".
#
```

```
perl -S dcfmake.pl

#
# alternately, you can run the javac compiler on the source code directly.
# make sure that %DCF_ROOT%\classes is in the CLASSPATH.

# run the application using the perl launcher script.
#
perl -S jrun_example.pl
com.lbs.examples.ex_jprint_element_value.ex_jprint_element_value
%DCF_ROOT%\test\images\test.dcm 0010,0010
```

This application shows how easy it is to load DICOM format files, and extract elements from them. The "`jrun_example.pl`" script calls the Java interpreter with the common DCF options (most of the Java examples use this launcher script; you may also create individual wrapper scripts for each example – see "`ex_jdcf_HelloWorld`" for an example; see Appendix G: Using Perl with the DCF for information about using Perl with the DCF). The first argument is the class name of the Java example to be run; specifically, it is the name of the class that defines the `main()` method. The second argument is the name of some DICOM file. The third argument is the DICOM tag for which value data is to be extracted – in this example the file is `test.dcm` and the attribute to be extracted is "Patients Name":

```
perl -S jrun_example.pl
com.lbs.examples.ex_jprint_element_value.ex_jprint_element_value
%DCF_ROOT%\test\images\test.dcm 0010,0010
```

5.3.1.1. Example – ex_jprint_element_value

The source code for this example is shown below:

```
package com.lbs.examples.ex_jprint_element_value;

import com.lbs.DCS.*;
import com.lbs.LOG_a.*;
import com.lbs.APC_a.*;
import com.lbs.DCF.*;

public class ex_jprint_element_value
{
   public static void main( String args[] )
   {
      try
      {
         if ( args.length != 2 )
         {
            throw new DCFException( "use: ex_jprint_element_value <dicom_filename> <tag
as hhhh,hhhh>" );
         }

         //
         // setup DCF Logger and start DDCSServer I/O subsystem.
         // logger will write to stdout in this configuration.
         //
         LOGClient_a.setup( args );

            DicomFileInput dfi = new DicomFileInput( args[0] );
            DicomDataSet   dds = dfi.readDataSetNoPixels();
            AttributeTag   tag = new AttributeTag( args[1] );
            String value = dds.getElementStringValue( tag );
            LOG.info( tag.toString() + " = " + value );
      }
      catch ( Exception e )
      {
         System.err.println( "error: " + e );
         System.exit( -1 );
      }

      if (AppControl.isInitialized())
      {
         AppControl.instance().shutdown(0);
      }

      System.exit( 0 );
   }
}
```

5.3.2. Using Java modify DICOM image data program

Open a command window and change to the *devel/jsrc/com/lbs/examples/ex_jModPixelData* directory under the DCF install directory, build, and execute the example application:

```
cd $DCF_USER_ROOT/devel/jsrc/com/lbs/examples/ex_jModPixelData
# optionally, rebuild
perl -S dcfmake.pl
# run. Note that the output file is written relative to the current
# directory, since I/O is no longer handled by a separate server.
#
perl -S jrun_example.pl
com.lbs.examples.ex_jModPixelData.ex_jModPixelData
%DCF_ROOT%\test\images\ct-ab-8.dcm new8bit.dcm
```

This application shows how to extract image header fields and modify the image pixel data. The DCF provides an image processing framework that allows standard and custom filter objects to be chained together. This example shows a simplified alternative for users who just want to gain access to the pixel data to change it, or perhaps to display it.

5.3.2.1. Example – ex_jModPixelData

The source code for the example is shown below:

```
//===========================================================================
// Copyright (C) 2004, Laurel Bridge Software, Inc.
// 160 East Main St.
// Newark, Delaware 19711 USA
// All Rights Reserved
//===========================================================================

package com.lbs.examples.ex_jModPixelData;

import com.lbs.DCF.*;
import com.lbs.LOG_a.*;
import com.lbs.APC_a.*;
import com.lbs.DCS.*;

public class ex_jModPixelData
{
   public static void main( String args[] )
```

```
    {
        if ( args.length != 2 )
        {
            System.err.println( "use: ex_jModPixelData <infile> <outfile>" );
            System.exit( -1 );
        }

        String infile  = args[ 0 ];
        String outfile = args[ 1 ];
        try
        {
            DicomFileInput dfi = new DicomFileInput( infile );

            DicomDataSet dds = dfi.readDataSet();
            dfi.close();

            // Get some of the basic image header fields.  Use the DicomDataSet
            // convenience methods to get integer values for these.
            int rows = dds.getElementIntValue( DCM.E_ROWS );
            int cols = dds.getElementIntValue( DCM.E_COLUMNS );
            int bits_allocated = dds.getElementIntValue( DCM.E_BITS_ALLOCATED );
            int bits_stored = dds.getElementIntValue( DCM.E_BITS_STORED );
            int high_bit = dds.getElementIntValue( DCM.E_HIGH_BIT );
            int pixel_count = rows * cols;

            // Get the pixel data element.
            DicomElement e_pixel_data = dds.findElement( DCM.E_PIXEL_DATA );

            // Get the pixels as an array.
            // Note that DICOM defines two possible types for pixel data:
            // OB (Other byte) and OW (Other word).  We cast our DicomElement
            // to the appropriate derived class.
            if ( e_pixel_data.vr() == DCM.VR_OB )
            {
                byte[] pixel_data = ((DicomOBElement)e_pixel_data).getOBData();
                byte[] new_pixel_data = new byte[ pixel_count ];
                // do some silly modification of pixel data, like an invert.
                byte offset = (byte)((1<<bits_stored)&0xFF);
                for (int i=0; i<pixel_count; i++)
                {
                    new_pixel_data[i] = (byte)(offset - pixel_data[i]);
                }
                dds.insert( new DicomOBElement( DCM.E_PIXEL_DATA, new_pixel_data ) );
            }
            else
            {
                short[] pixel_data = ((DicomOWElement)e_pixel_data).getOWData();
                short[] new_pixel_data = new short[ pixel_count ];

                short offset = (short)((1<<bits_stored)&0xFFFF);
                for (int i=0; i<pixel_count; i++)
                {
                    new_pixel_data[i] = (short)(offset - pixel_data[i]);
                }
                dds.insert( new DicomOWElement( DCM.E_PIXEL_DATA, new_pixel_data ) );
            }

            // Give the object a new UID.
            // There are other attributes that should also be set to
            // indicate that this is a "derived image".
            // That is beyond the scope of this example.
            dds.insert( DicomElementFactory.create(
                    DCM.E_SOPINSTANCE_UID, DicomDataDictionary.makeUID() ) );

            // Save the data set with the new pixel data.
            DicomFileOutput dfo =
                new DicomFileOutput( outfile, UID.TRANSFERLITTLEENDIAN, false );
            dfo.writeDataSet( dds );
            dfo.close();
        }
        catch( DCFException e )
        {
            System.err.println( "error: " + e );
            System.exit( -1 );
        }
        if (AppControl.isInitialized())
        {
            AppControl.instance().shutdown(0);
        }
        System.exit( 0 );
    }
```

5.3.3. Using Java DICOM Verification (Echo) Client Classes

This example shows the use of `com.lbs.DCS.DicomSCU` to connect to an SCP, send, and receive DIMSE messages.

Before running the example, you will need to provide an SCP for the verification service class. From the DCF Remote Service Interface page in your web browser (see Section 2.4.3), click “Choose a Configuration”, and then choose “*mwl_server_win32.cfg*”.

(Note: You could select other server system configurations, but this one will work for this example as well as for the following one.)

When the message indicating that the system is started appears, click “Back” or wait for an auto-redirect to return to the main DCF Remote Service Interface.

Background: Starting this DCF system configuration (*mwl_server_win32.cfg*) performs the following:

- The script *dcfstart.pl* reads a configuration file and starts a specified list of utility or server processes. (Utility processes are run in the foreground and are typically used for pre-startup cleanup, etc. Server processes are run in the background.)
- Various system cleanup utilities are run; for example, log files from the last session are archived to a subdirectory under *$DCF_TMP/log*, stale process configuration files are removed, etc.
- The DCF log server application is started. Client `LOG` adapters for C++, Java, and C# optionally forward messages to this server.
- The `DCDS_Server` (**D**istributed **C**onfiguration **D**ata **S**ervice) is started. This provides a lightweight distributed object, hierarchical database for configuration data.
- The `dcf_mwl_scp` is started, which is the DCF simple MWL server implemented in C++.

Open a command window and change to the *devel/jsrc/com/lbs/examples/ex_jecho_scu* directory under the DCF install directory, then build and execute the example application:

```
cd $DCF_USER_ROOT/devel/jsrc/com/lbs/examples/ex_jecho_scu
perl -S dcfmake.pl
perl -S jrun_example.pl com.lbs.examples.ex_jecho_scu.ex_jecho_scu
MWLSCP1 localhost 2000
```

5.3.3.1. Example – ex_jecho_scu

The source code for the example is shown below:

```
//=====================================================================
// Copyright (C) 2002-2005, Laurel Bridge Software, Inc.
// 160 East Main St.
// Newark, Delaware 19711 USA
// All Rights Reserved
//=====================================================================
package com.lbs.examples.ex_jecho_scu;

import com.lbs.LOG.*;
import com.lbs.APC.*;
import com.lbs.CDS.*;
import com.lbs.DCF.*;
import com.lbs.DCS.*;
import com.lbs.LOG_a.*;
import com.lbs.APC_a.*;

public class ex_jecho_scu
{
   private static String usage_ = "use ex_jecho_scu <called_ae> <called_host>
<called_port>\n";

   public static void main( String args[] )
   {
      try
      {
         AppControl_a.setupORB( args );
         CFGDB_a.setFSysMode(true);
         CFGDB_a.setup( args );
         AppControl_a.setup( args, CINFO.instance() );
         LOGClient_a.setup( args );

         if ( args.length != 3 )
         {
            throw new DCFException( usage_ );
         }

         try
         {
            VerificationClient client = new VerificationClient(
                                          "ECHO_SCU",
                                          args[0],
                                          args[1] + ":" + args[2]
                                          );
            client.requestAssociation();
            client.cEcho(10);
            client.releaseAssociation();
         }
         catch ( DCFException e)
         {
            LOG.error( -1, "DCFException caught:\n", e );
            System.err.println("test failed - see log files for output");
         }
      }
      catch (Exception e )
      {
         LOG.error( -1, "Exception caught:\n", e );
         System.err.println("test failed - see log files for output");
      }
      if (AppControl.isInitialized())
      {
         AppControl.instance().shutdown(0);
      }
      System.Exit(0);
   }
}
```

5.3.4. Using Java Modality Worklist Query SCU Classes

This example shows how to extend `com.lbs.DCS.AssociationRequester` to create a working Modality Worklist SCU. The example creates a worklist query, using the generated IOD wrappers, and then sends the query as a C-Find and processes C-Find-Response messages until a final status is returned by the SCP.

This example uses the server configuration (*`mwl_server_win32.cfg`*) started in the previous example. This server should be running to allow this example to run (see 5.3.3).

Open a command window and change to the *`devel/jsrc/com/lbs/examples/ex_jmwl_client`* directory under the DCF install directory, then build and execute the example application:

```
cd $DCF_USER_ROOT/devel/jsrc/com/lbs/examples/ex_jmwl_client
# optionally rebuild the example
perl -S dcfmake.pl
# run the example with the parameters: <called-ae-title> <host> <port>
perl -S jrun_example.pl com.lbs.examples.ex_jmwl_client.ex_jmwl_client
MWLSCP1 localhost 2000
```

5.3.4.1. Example – ex_jmwl_client

The source code for the example is shown below:

```
//=========================================================================
// Copyright (C) 2007, Laurel Bridge Software, Inc.
// 160 E. Main Street
// Newark, Delaware 19711 USA
// All Rights Reserved
//=========================================================================

package com.lbs.examples.ex_jmwl_client;

import java.util.Vector;
import com.lbs.DCF.*;
import com.lbs.APC.*;
import com.lbs.DCS.*;
import com.lbs.DIS.*;


/**
*/
public class ex_jmwl_client extends AssociationRequester
{

   private AssociationInfo ainfo_ = null;

   private boolean f_connected_;

   public ex_jmwl_client( String calling_ae, String called_ae, String called_addr )
      throws DCSException
   {
      f_connected_ = false;

      ainfo_ = new AssociationInfo();
      ainfo_.callingTitle( calling_ae );
      ainfo_.calledTitle( called_ae );
      ainfo_.calledPresentationAddress( called_addr );

      String ts_list[] = new String[3];
      ts_list[0] = UID.TRANSFERLITTLEENDIANEXPLICIT;
      ts_list[1] = UID.TRANSFERLITTLEENDIAN;
      ts_list[2] = UID.TRANSFERBIGENDIANEXPLICIT;
      RequestedPresentationContext rq_ctx
         = new RequestedPresentationContext( (byte)1,
               UID.SOPMODALITYWORKLIST_FIND, ts_list );

      ainfo_.addRequestedPresentationContext( rq_ctx );

   }

   public void requestAssociation()
      throws DCSException
   {
      setAssociationInfo( ainfo_ );
      super.requestAssociation();
      ainfo_ = getAssociationInfo();

      Vector accepted_ctx_list = ainfo_.acceptedPresentationContextList();

      for( int i=0; i<accepted_ctx_list.size(); i++ )
      {
         AcceptedPresentationContext ctx =
            (AcceptedPresentationContext)
               accepted_ctx_list.elementAt(i);

         if (ctx.id() == 1)
         {
            f_connected_ = true;
         }
      }

      if (!f_connected_)
      {
         releaseAssociation();
         throw new DCSException("Association was accepted, but the required presentation
context was not");
      }

   }


   public int sendCFind( DicomDataSet query, int timeout )
      throws DCSException
   {
      if (!isConnected())
      {
         throw new DCSException("invalid state: not connected");
      }

      DimseMessage msg = new DimseMessage();

      msg.commandField( DimseMessage.C_FIND_RQ );
      msg.affectedSopclassUid( UID.SOPMODALITYWORKLIST_FIND );
      msg.dataSetType(0x0100);
      msg.context_id(1);
      msg.priority( 1 );

      msg.data( query );

      LOG.debug( CINFO.df_SHOW_GENERAL_FLOW,
         "sending C-Find Request:\n" + msg );

      sendDimseMessage( msg, 30 );

      return 0;

   }

   DimseMessage getCFindResponse( int timeout )
```

```
      throws DCSException
   {
      DimseMessage rsp = receiveDimseMessage( (short)1, timeout );


      // Note: do this, and we avoid calling toString() on the
      // response message when debug is off.
      if (CINFO.testDebugFlags( CINFO.df_SHOW_GENERAL_FLOW ))
      {
         LOG.debug( "received response message:\n" + rsp );
      }

      return rsp;
   }


   private static String usage_ = "use ex_jmwl_client <called_ae> <called_host>
<called_port>\n";

   public static void main( String args[] )
   {
      int status;

      try
      {

         //
         // Uncomment if you want the standard common services setup -
         // i.e.,
         //=================================================================
         // CDS_a.CFGDB adapter uses DCDS_Server,
         //
         // APC_a.AppControl adapter loads app config from
         // /apps/defaults/examples/ex_jmwl_client,
         // saves proc config to /procs/ex_jmwl_client.<pid>.
         // DDCSServer jni Dicom IO library uses this proc configuration.
         //
         // LOG_a.LOGClient adapter uses file and DLOG_Server outputs, per
         // app config settings.
         //
         //=================================================================
         // Leave commented for minimal common services setup -
         //=================================================================
         // CFGDB adapter reads from files in $DCF_CFG as needed
         //
         // AppControl does not maintain an application or proc config. (DDCSServer
         // jni Dicom IO library loads config info as needed from
         // /apps/defaults/dcf_java_default)
         //
         // LOGClient writes to console
         //
         //-----------------------------------
         // Framework.initDefaultServices( CINFO );
         //-----------------------------------


         if ( args.length != 3 )
         {
            throw new DCFException( usage_ );
         }


         // create SCU
         ex_jmwl_client client = new ex_jmwl_client("MWL_SCU",
               args[0],
               args[1] + ":" + args[2]
               );
         // connect
         client.requestAssociation();

         // make a query, using the generated IOD wrappers for convenience
         ModalityWorklistItem query = new ModalityWorklistItem();

         query.patientsName("");
         query.patientId("");
         query.studyInstanceUid("");

         ScheduledProcStepSeq sps = new ScheduledProcStepSeq();
         sps.schedProcStepStartDate("20000112-20041231");
         sps.modality("CR");

         query.scheduledProcStepSeq( sps );

         // send the query (C-Find-Request)
         int timeout=10;

         System.err.println("query dataset =\n" + query.data_set() );
         client.sendCFind( query.data_set(), timeout );

         // process responses until we get a final status
         for (;;)
         {
            DimseMessage response = client.getCFindResponse( timeout );
            System.err.println("received C-Find Response:\n" + response );

            if ( ( response.status() == DimseStatus.DIMSE_SUCCESS )
               || ( response.status() == DimseStatus.DIMSE_FAILURE ) )
            {
               break;
            }
         }

         // hang up
         client.releaseAssociation();

         status = 0;
      }
      catch ( Exception e )
      {
         LOG.error(-1, "error", e );
         status = 1;
      }

      if (AppControl.isInitialized())
      {
         AppControl.instance().shutdown(status);
      }

      System.exit(status);
   }

}
```

5.3.5. Using Java Print Client Classes

A common application of the DICOM protocol is sending DICOM images or other objects to a DICOM Print Service Class Provider. The simplest way to use the DCF for printing is to use the `PrintClient` class. `PrintClient` provides a high level mechanism for printing DICOM objects and communicating with the print SCP. The print job itself is described by the `PrintJobDescription` class. Users of `PrintClient` that want to receive notifications as the status of the DICOM print-job or printer changes should implement the `PrintClientListener` interface.

The DCF Java `PrintClient` class provides a very high level interface to a DICOM print SCP. The application developer is removed from the process of negotiating an association, sending the n-create and n-action and other DIMSE messages, managing the complex relationships between objects in the normalized service classes, and handling printer and print job status notifications. The sheets of images that are to be printed are defined in an intuitive hierarchical structure. The `PrintClient` object handles the messy details of DICOM Print.

The `PrintJobDescription` object contains basic attributes of the job, such as the server address, and various job level options. Also included in the `PrintJobDescription` is a single `PrintJobFilmSession` object. This corresponds to the DICOM film-session object. `PrintJobFilmSession` contains one or more `PrintJobFilmBox` objects. A `PrintJobFilmBox` corresponds to the DICOM film-box object, which represents a sheet or film to be printed. `PrintJobFilmBox` contains one or more `PrintJobImageBox` objects. A `PrintJobImageBox` corresponds to a DICOM image-box and represents a single image to be placed somewhere on the film. When the job has completed, a `PrintJobStatus` object is returned that summarizes the results of the print operation.

The source code for a simple example print client application is shown below. The example may be found in the *`devel/jsrc/com/lbs/examples/ex_jprint_client`* directory under the DCF installation directory.

Open a command window and change to the *`devel/jsrc/com/lbs/examples/ex_jprint_client`* directory under the DCF install directory, then build and execute the example application:

```
cd $DCF_USER_ROOT/devel/jsrc/com/lbs/examples/ex_jprint_client
# optionally rebuild the example
perl -S dcfmake.pl
# run the example with the parameters: <image name> <host:port:CALLED_AE>
perl -S jrun_example.pl
com.lbs.examples.ex_jprint_client.ex_jprint_client
%DCF_ROOT%/test/images/test.dcm localhost:2000:PrintSCP1
```

5.3.5.1. Example – ex_jprint_client

The application initializes the DCF core framework – CDS, APC, etc – and then the `DicomDataService` adapter, which is used by the `DicomInstanceInfo` class to load objects. It sets up the print job – specifying items like the address of the DICOM printer, and configures the job with a film box inside a film session and a print job object – and submits it. A `PrintJobStatus` object reports the final status of the print operation.

The source code for the example is shown below:

```
//=====================================================================
// Copyright (C) 2005, Laurel Bridge Software, Inc.
// 160 East Main St.
// Newark, Delaware 19711 USA
// All Rights Reserved
//=====================================================================

package com.lbs.examples.ex_jprint_client;

import com.lbs.APC.*;
import com.lbs.LOG_a.*;
import com.lbs.CDS_a.*;
import com.lbs.APC_a.*;
import com.lbs.DDS.*;
import com.lbs.DDS_a.*;
import com.lbs.DCF.*;
import com.lbs.DCS.*;
import com.lbs.DPS.*;

public class ex_jprint_client
{
   public ex_jprint_client()
   {
   }

   public static void usage()
   {
      System.out.println( "Usage: ex_jprint_client image name host:port:CALLED_AE_TITLE"
);
   }

   public void runPrint( String args[] )
      throws DCSException, DCFException
   {
      // Load image
      // Note:  In the future you will be able to use a DicomDataSet in
      // place of a file.
      DicomPersistentObjectDescriptor dpod = new DicomPersistentObjectDescriptor();
      dpod.persistentId( args[ 0 ] );

      // Explicit Little Endian
      dpod.persistentInfo( "1.2.840.10008.1.2.1" );

      // Print server
      String print_server_address = args[ 1 ];

      PrintJobDescription job          = new PrintJobDescription();  // describes the
job we want to do
      PrintJobFilmSession film_session = new PrintJobFilmSession();
      PrintJobFilmBox film_box         = new PrintJobFilmBox();
      PrintJobImageBox image_box       = new PrintJobImageBox();

      job.serverAddress( print_server_address );
      job.clientAddress( "DEMO" );
      job.requestPrintJobSOPClass( true );
      job.pollPrintJob( true );
      job.printJobPollRateSeconds( 2 );
      job.jobTimeoutSeconds( 30 );

      film_session.numberOfCopies( "1" );
      film_session.printPriority( "HIGH" );
      film_session.mediumType( "BLUE FILM" );
      film_session.filmDestination( "MAGAZINE" );
      film_session.filmSessionLabel( "test" );
      film_session.memoryAllocation( "0" );
      film_session.ownerId( "DCF" );

      film_box.imageDisplayFormat( "STANDARD\\1,1" );
      film_box.filmOrientation( "PORTRAIT" );
      film_box.filmSizeId( "14INX17IN" );
      film_box.magnificationType( "NONE" );
      film_box.smoothingType( "NONE" );
      film_box.borderDensity( "0" );
      film_box.emptyImageDensity( "0" );
      film_box.minDensity( 0 );
      film_box.maxDensity( 280 );
      film_box.trim( "YES" );
      film_box.configurationInformation( "NONE" );
      film_box.illumination( 0 );
      film_box.reflectedAmbientLight( 0 );
      film_box.requestedResolutionId( "HIGH" );

      image_box.imagePosition( 1 );
      image_box.polarity( "NORMAL" );
      image_box.magnificationType( "NONE" );
      image_box.smoothingType( "NONE" );
      image_box.configurationInformation( "NONE" );
      image_box.requestedImageSize( "0" );
      image_box.reqdDecimatecropBehavior( "DECIMATE" );
      image_box.imageInstanceInfo( new DicomInstanceInfo( dpod ) );

      film_box.addImageBox( image_box );
      film_session.addFilmBox( film_box );
      job.filmSession( film_session );

      PrintClient print_client = new PrintClient();

      LOG.info( "submitting print job:" + job.toString() );

      PrintJobStatus job_status = new PrintJobStatus( job.jobUID() );
      job_status.status( "RUNNING" );
      job_status.statusInfo( "NORMAL" );
      print_client.submitPrintJob( job, null, job_status );

      LOG.info( "Done! Print job status: " + job_status );
   }

   public static void main( String args[] )
   {
      if ( args.length < 2 )
      {
         usage();
         System.exit( 0 );
      }

      try
      {
         AppControl_a.setupORB( args );
         CFGDB_a.setFSysMode( true );
         CFGDB_a.setup( args );
         AppControl_a.setup( args, CINFO.instance() );

         // DicomDataService must be set up for the DicomInstanceInfo
         // to load the image.
         DicomDataService_a.setup( args );
         ex_jprint_client client = new ex_jprint_client();

         client.runPrint( args );
      }
      catch( DCSException e )
      {
         LOG.error( -1, "DCS Exception caught: ", e );

         if ( AppControl.isInitialized() )
         {
            AppControl.instance().shutdown( -1 );
         }
         System.exit( -1 );
      }
      catch( DCFException e )
      {
         LOG.error( -1, "DCF Exception caught: ", e );

         if ( AppControl.isInitialized() )
```

```
            {
               AppControl.instance().shutdown( -1 );
            }
            System.exit( -1 );
         }
         catch( Exception e )
         {
            LOG.error( -1, "Exception caught: ", e );
            LOG.error( -1, LOG.formatStackTraceMsg( e ) );

            if ( AppControl.isInitialized() )
            {
               AppControl.instance().shutdown( -1 );
            }
            System.exit( -1 );
         }

         AppControl.instance().shutdown( 0 );
         System.exit( 0 );
   }
}
```

5.3.6. Using Java Store Client Classes

A common application of the DICOM protocol is sending DICOM images or other objects to a DICOM Storage Service Class Provider. The simplest way to use the DCF is to use the `StoreClient` class. `StoreClient` provides a high level mechanism for sending DICOM objects using the C-STORE-RQ DIMSE message. The store job itself is described by the `StoreJobDescription` class. Users of `StoreClient` that want to receive notifications when each image has been stored and a final status for the job should implement the `StoreClientListener` interface.

5.3.6.1. Example – ex_jstore_client

See *`$DCF_ROOT/devel/jsrc/com/lbs/examples/ex_jstore_client`* for the complete source code, including code implementing the `StoreClientListener` interface.

- Listed here are two methods from *`ex_jstore_client`* that do most of the work: `createStoreJobDescription()` creates a `StoreJobDescription` object;
- `runJob()` uses a `StoreClient` with the created `StoreJobDescription` to send the job to an SCP.

The source code for the examples is shown below:

`createStoreJobDescription:`

```
private StoreJobDescription createStoreJobDescription()
         throws DCSException, DCFException
   {
         StoreJobDescription sjd;
         //If we got a Job Description cfg file name on the command line.
         if( cfgstr_ != null )
         {
            CFGGroup storeJobConfiguration = CFGDB.instance().loadGroup( cfgstr_ );
            CFGGroup configuration =
storeJobConfiguration.getGroup("store_job_description");
            LOG.info( "Job CFGGroup = " + configuration.toString() );
            sjd = new StoreJobDescription( configuration );
         }
         else
         {
```

```
         //Build up a StoreJobDescription from command line options and a list of
filenames provided
         //on the command line.
         sjd = new StoreJobDescription();

         // server name is either an AE name in the config db,
         // (currently %DCF_CFG%/dicom/network/ae_title_mappings)
         //
         // or is <host>:<port>:<called-ae>
         // e.g.:  sjd.ServerName = "localhost:2000:StoreSCP1";
         sjd.calledHost( called_host_ );
         sjd.calledPort( called_port_ );
         sjd.calledAETitle( called_ae_title_ );

         sjd.callingAETitle( calling_ae_title_ );

         sjd.responseTimeoutSeconds( 40 );

         //filenames from command line args
         for (int i=0 ; i<file_list_.size(); i++ )
         {
            String filename = (String)file_list_.get( i );
            LOG.info( "Filename = " +  filename );

            sjd.addInstance( new StoreJobInstanceInfo( filename ) );
         }
      }
      return sjd;
   }
```

runJob:

```
   public int runJob()
   {
      int exit_status= -1;
      try
      {
         // create a job
         StoreJobDescription sjd = createStoreJobDescription();

         for( int i=0; i<=repeat_count_; i++ )
         {
            // create a StoreClient
            StoreClient client = new StoreClient();

            StoreJobStatus status_ret = new StoreJobStatus( null,null,null,null,0,0);

            DicomSessionSettings session_settings = new DicomSessionSettings();

            // submit the job and wait for it to complete.
            client.submitStoreJob( sjd, this, status_ret );

            if( status_ret.status().compareTo( "SUCCESS" ) == 0 )
            {
               exit_status = 0;
            }
            else
            {
               exit_status = 1;
            }
            LOG.info( "Current status = " + status_ret.toString() );
         }
      }
      catch( Exception e )
      {
         LOG.error( -1, "", e );
```

```
            exit_status = 1;
        }
        return exit_status;
    }
```

5.3.7. Using Java Q/R Client Classes

A common application of the DICOM protocol is querying an image archive for images or other composite objects. The simplest way to do this with the DCF is to use the `QRSCU` class. `QRSCU` provides a high level mechanism for interoperating with `QRSCP`'s via the C-FIND, C-MOVE and C-GET DIMSE messages.

5.3.7.1. Example – ex_jqr_scu

See *`$DCF_ROOT/devel/jsrc/com/lbs/examples/ex_jqr_scu/ex_jqr_scu.java`* for a command line application that uses these classes and interface.

See *`$DCF_ROOT/devel/jsrc/com/lbs/examples/ex_jquery_scu`* for a GUI application that uses these classes and interface.

Here is a code fragment that creates a `QRSCU` object and a `QRIdentifier` (query attributes) then sends it as a C-FIND Request.

```
String host = hostname_box_.getText().trim();
String port = port_box_.getText().trim();

AssociationInfo ainfo = new AssociationInfo();
ainfo.calledTitle( called_ae_box_.getText().trim() );
ainfo.callingTitle( calling_ae_box_.getText().trim() );
ainfo.calledPresentationAddress( host + ":" + port );

RequestedPresentationContext ctx = new RequestedPresentationContext( (byte)1,
        UID.SOPPATIENTQUERY_FIND, new String[] {ts_uid_} );
ainfo.addRequestedPresentationContext( ctx );

CFGGroup session_cfg;
DicomSessionSettings session_settings = new DicomSessionSettings();
String session_cfg_name = "/dicom/QRSCU_default_session_settings.cfg";
try
{
    session_cfg = CFGDB.instance().loadGroup( session_cfg_name, true );
    session_settings = new DicomSessionSettings( session_cfg );
}
catch( CDSException e1 )
{
    LOG.error( -1, "Error loading session settings from CFGDB group name = "
            + session_cfg_name, e1 );
}

QRSCU scu_ = new QRSCU( ainfo, session_settings );

Configuration config = parent_panel_.getConfiguration();
scu_.maxReturnedResults( config.maxReturnedResults() );
scu_.queryTimeoutSeconds( config.queryTimeoutSeconds() );
scu_.setRequestedSopClassUid( UID.SOPPATIENTQUERY_FIND );


QRIdentifier query = new QRIdentifier();
// Fields to query on:
query.patientsName( patient_name_box_.getText().trim() );
query.queryretrieveLevel( (String)query_level_combo_box_.getSelectedItem() );
query.studyInstanceUid( study_uid_box_.getText().trim() );
```

```
// Fields to return:
query.patientId( "" );
query.patientsSex( "" );
query.modality( "" );
query.studyDate( "" );
query.numberStudyRelInstances( "" );
query.data_set().insert( DCM.E_INSTANCE_AVAILABILITY, "" );

addExtraDicomElementsToQuery( query, config.additionalDicomElements() );

DicomDataSet expected_fields = new DicomDataSet();
setTagsToDisplay( expected_fields, config.tagsToDisplay() );

parent_panel_.setFieldsExpected( expected_fields );

LOG.debug( CINFO.df_SHOW_GENERAL_FLOW, "Query Data set is " + query.data_set() );

// TODO: figure out how to bold some parts of the output
query_status_.append( "Query data set:\n" );
query_status_.append( query.data_set().toString() );
query_status_.append( "\n" );

scu_.requestAssociation();
scu_.cFind( query.data_set(), QueryInfoPanel.this, false );
```

By implementing the `QueryListener` interface an object will notified when DIMSE Response messages are received, such as C-FIND-RSP or C-MOVE-RSP or C-GET-RSP.

The `queryEvent()` method is called for each intermediate response.

```
public void queryEvent( DimseMessage rsp )
{
    //output contents of message to log file
    LOG.info( "Received a DIMSE response message " + rsp );
}
```

The `queryComplete()` method is called when a final response is received or if an internal error occurs in the `QRSCU` class.

```
public void queryComplete( int status )
{
    try
    {
        //internal error send an abort message.
        if( status == 4 )
        {
            LOG.error( -1, "Aborting association\n" );
            scu_.abortAssociation();
        }
        else
        {
            LOG.info( "Releasing association\n" );
            scu_.releaseAssociation();
        }
        scu_ = null;
    }
    catch( DCSException e )
    {
        LOG.error( -1, "Error occurred while disconnecting association", e );
    }
}
```

5.3.8. Handling alternate character sets with DCF (Java)

The DCF provides limited support for string elements that are encoded in non-default character sets (i.e., a dataset that contains element `0008,0005` Specific Character Set and where that element's value is set).

Essentially, when reading any string data from a DICOM file or DIMSE message, DCF will explicitly specify that the raw bytes should be converted to a string using `ISO8859-1` decoding. Since `ISO8859-1` is an 8 bit character set, you can get the bytes back from the element and create a new string that decodes the bytes using a different character set. Likewise, when the DCF retrieves the bytes from a string element prior to writing a file or DIMSE message, it will specify that `ISO8859-1` encoding should be used. For example:

```
DicomStringElement e = dds.getElement( DCM.E_PATIENTS_NAME );
String s = e.getEncodedStringValue();
byte[] raw_data = s.getBytes( "ISO8859-1"); // since we decoded this way, this should
                                           // produce the original bytes again.
String s2 = new String( raw_data, some_other_charset_name );
```

The method `getEncodedStringValue` concatenates all of the values using "\" as the delimiter, and adds any needed pad chars (space or null for UIDs). The resultant value should end up looking the same as the original raw data.

Notice that the number of values (`DicomStringElement.vm()` or `DicomStringElement.values().length`) when we force `ISO8859-1` decoding may be different from the number of values when using some other character set. That is, a "\" byte value in a string in an alternate char set may actually be something other than a DICOM string VM delimiter.

Also note that if you create `DicomStringElements` containing text in an alternate character set, you may need to re-encode it as `ISO8859-1` before passing it to the constructor: For example:

```
String s = some_string_in_an_altnate_char_set;
byte[] raw_data = s.getBytes(  alternate_char_set_name );
String s2 = new String( raw_data, "ISO8859-1"  );
DicomPNElement e = new DicomPNElement( DCM.E_PATIENTS_NAME, s2 );
  // or
DicomElement e = DicomElementFactory.create( DCM.E_PATIENTS_NAME, s2 );
```

Note that Chapter 3, Section C.12.1.1.2 Specific Character Set in the DICOM standard outlines all the various defined terms for DICOM element `0008,0005` Specific Character Set. Another list is provided in Chapter 18, Annex D - IANA Mapping (informative).

The Windows character set names, for example, don't match up to the values DICOM requires for the `0008,0005` tag. The OEM developer will need to write a mapping method of some kind to translate back and forth between the various naming conventions. For example, if the encoding name for Java is "`ISO-8859-2`" (ISO Latin 2), then, by these tables, the DICOM value to put in the `0008,0005` attribute would be "`ISO_IR 101`".

5.4. Deploying a Simple Standalone DCF Java Application

The following procedure shows a simple method of deploying a DCF Java application. The application (.class), its required libraries, and configuration data can be installed into a single directory on the target system. The application can then be run from the installation directory.

We'll show the process of creating the install directory on your DCF developer box (the host with the DCF toolkit installed). Once created, that install directory can then be copied to the target using any number of methods: zip on your DCF developer box, and unzip on the target; or perhaps burn this directory to a CD-ROM and then run directly from the CD on the target.

This example shows deploying the `ex_jdcf_filter` example and `ex_jdcf_dump`. The process would be modified somewhat for your own application.

Perform the following steps:

1. Open a DCF command window:
   Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Command Prompt"
2. Create the test install directory:
   Note: You could paste this text into a batch file and run it to automate this process.

```
REM ###
REM ### create install dir
REM ###
mkdir DCF_test_java_install
cd DCF_test_java_install

REM ###
REM ### copy required library files from %DCF_LIB% (../DCF/lib)
REM ###
copy %DCF_LIB%\DCF_DCFCore.dll

REM ###
REM ### copy required Java JAR file from %DCF_ROOT%\classes
copy %DCF_ROOT%\classes\LaurelBridge.jar

REM ### copy required library files from %DCF_BIN% (../DCF/bin).
REM ### These may exist in other places on the system, but copies
REM ### are put here during DCF toolkit install for convenience.
REM ### (Note omniORB dlls may not be required depending on the
REM ### application and your DCF version.)
copy %DCF_LIB%\DCF_com_lbs_DCS_DicomTSCWCodec.dll
copy %DCF_LIB%\DCF_TSCW.dll
copy %DCF_LIB%\DCF_TSCWIJG.dll
copy %DCF_LIB%\DCF_TSCWJasper.dll
REM ### The Aware wrapper dll is needed only if using Aware's JPEG libraries.
REM ### Note the actual Aware JPEG library (awj2k.dll) must be purchased separately
copy %DCF_LIB%\DCF_TSCWAware.dll

REM ### If you are building from a DCF VisualStudio8.x .NET toolkit:
copy %DCF_BIN%\msvcp80.dll
copy %DCF_BIN%\msvcr80.dll

REM ###
REM ### Copy the classes for the application that you have - for example,
REM ### include both the Java ex_jdcf_filter example and the ex_jdcf_dump
REM ### utilities.  Make sure that the directory structure matches, i.e.,
REM ### you must have the class files in com\lbs\examples.
REM ###
copy %DCF_ROOT%\classes\com\lbs\examples\ex_jdcf_filter\*
         classes\com\lbs\examples\ex_jdcf_filter
copy %DCF_ROOT%\classes\com\lbs\examples\ex_jdcf_dump\*
         classes\com\lbs\examples\ex_dcf_dump

REM ###
REM ### Create a minimal configuration directory.
REM ###
mkdir cfg
mkdir cfg\apps
mkdir cfg\apps\defaults
mkdir cfg\procs
```

```
    REM ###
    REM ### Copy the license configuration file, and the application configs
    REM ### for the installed programs.
    REM ###
    copy %DCF_CFG%\systeminfo cfg\systeminfo
    copy %DCF_CFG%\apps\defaults\ex_jdcf_filter cfg\apps\defaults
    copy %DCF_CFG%\apps\defaults\ex_jdcf_dump cfg\apps\defaults
```

3. Create the media by which you will deliver the install directory.
4. On the target machine do the following:
   a) Unpack, copy, or otherwise make the DCF app install directory available. For example, copy or unzip to `C:\temp\DCF`
   b) Install the Java JRE on the target box if it is not already installed; some applications may also require the JAI and JAI Image I/O installations to read and write DICOM files with JPEG data.
   c) From a command window, go to the install directory. For example, use `C:\temp\DCF`.
      `cd C:\temp\DCF`
   d) Set environment variables and run your apps (you could put these steps in a *`run_app.bat`* file). You need to put the path to the classes and the DCF JAR file in the CLASSPATH, as well as putting your installation directory into the PATH so that the DLLs can be found.

```
    set DCF_CFG=C:\temp\DCF\cfg
    set CLASSPATH=C:\temp\DCF\classes;C:\temp\DCF\LaurelBridge.jar;%CLASSPATH%
    set PATH=C:\temp\DCF;%PATH%

    ### Filter a file
    java com.lbs.examples.ex_jdcf_filter.ex_jdcf_filter –f file:/some_filter.cfg

    ### display input image (choose a dicom file in the line below)
    java com.lbs.examples.ex_jdcf_dump.ex_jdcf_dump \temp\test.dcm
```

5.5. Common Services Programming Examples

5.5.1. Java “Hello World” Example Using DCF Common Services

To demonstrate some of the capabilities of the DCF, you can create and run the most basic of code examples: the “Hello World” program. The DCF “hello world” demo will make use of the DCF development tools, as well as the common services APIs and implementations.

Change to the *`devel/jsrc/com/lbs/examples/ex_jdcf_HelloWorld`* directory under the DCF install directory, then build and execute the example application:

```
cd $DCF_USER_ROOT/devel/jsrc/com/lbs/examples/ex_jdcf_HelloWorld
perl –S dcfmake.pl
perl ./ex_jdcf_HelloWorld.pl
```

From your web browser, select “View Log Files” from the DCF Remote Service Interface. Select the log file for the ex_jdcf_HelloWorld application, and view the output.

To create the ex_jdcf_HelloWorld application, the following steps were followed:

1. Create a directory for the application
2. Create a component information file for the application
3. Create the source code for the application
4. Create a wrapper script to invoke the application

5. Build the application
6. Update the configuration database

1. Create a directory for the new application component – In the DCF, every application or library is a component, and has its own source directory.
   `mkdir $DCF_ROOT/devel/jsrc/com/lbs/examples/ex_jdcf_HelloWorld`
2. Create a component information file in that directory. This file must be called “*`cinfo.cfg`*”. For this example it contains the following:

```
#==============================================================================
# static information common to all instances of ex_jdcf_HelloWorld component
#==============================================================================
[ component_info ]
name = ex_jdcf_HelloWorld
type = java_app
category = examples
description = Java test application for DCF common services
package_prefix = com.lbs.examples
version = 0.1

[ required_components ]
component = java_lib/LOG_a
component = java_lib/APC_a
component = java_lib/CDS_a

[ debug_controls ]
debug_flag = DF_EXAMPLE,     0x10000, do something special if this debug flag is set

#==============================================================================
# per-instance information for the ex_jdcf_HelloWorld component
#==============================================================================
[ ex_jdcf_HelloWorld ]
debug_flags = 0x00000

[ ex_jdcf_HelloWorld/english ]
hello_world = hello world

[ ex_jdcf_HelloWorld/spanish ]
hello_world = hola mundo

[ ex_jdcf_HelloWorld/french ]
hello_world = bonjour le monde

[ ex_jdcf_HelloWorld/german ]
hello_world = hallo welt
```

The file is in the DCF configuration file format, which provides for attributes, groups, and nested groups.

Note: The easiest way to create the cinfo.cfg file for your application or library is to copy one from a similar component, then edit as needed.

The first group `[ component_info ]` describes basic attributes of the component. The `name` “ex_jdcf_HelloWorld” is combined with the `package_prefix` “com.lbs.examples” to form the Java package name “com.lbs.examples.ex_jdcf_HelloWorld”. Note that this corresponds to the source directory name, relative to the Java source directory (*`DCF_ROOT/devel/jsrc`*). The component `type` is “java_app” which indicates a Java application.

You can use *dcfmake.pl* to create applications in any directory, as long as you create a *cinfo.cfg* file in that directory. You can also use DCF Java classes from your application as you would any other Java class library.

The `[ required_components ]` group specifies three components needed by this application. The group `[ debug_controls ]` is where the developer can add support for conditional logging or other behavior specific to this component. Debug controls that are defined here can be accessed via the web interface.

The `[ ex_jdcf_HelloWorld ]` group contains the instance configuration for the component. This data is used directly in the example code.

3. Create the application source code

   For this example, the file is called "*ex_jdcf_HelloWorld.java*". Because of the package name that was specified in the cinfo.cfg file, the generated class will thus be

   `com.lbs.examples.ex_jdcf_HelloWorld.ex_jdcf_HelloWorld`

```
package com.lbs.examples.ex_jdcf_HelloWorld;

import com.lbs.LOG.*;
import com.lbs.APC.*;
import com.lbs.CDS.*;
import com.lbs.DCF.*;

/**
* The class ex_jdcf_HelloWorld demonstrates the most basic of programs
* using the DCF common services interfaces for Application Control (APC),
* Configuration Data Services (CDS), and Logging (LOG)
*
* To make it interesting, messages from different languages
* are retrieved from the application configuration. This is not intended
* to illustrate a production approach to internationalization.
*/
public class ex_jdcf_HelloWorld
{
    private static String usage_ = "use ex_jdcf_HelloWorld [adapter options]
[ex_jdcf_HelloWorld options]\n"
        + "adapter options are passed to LOG_a, CDS_a, and APC_a setup methods\n"
        + "use -help to get each adapter to display its options without setting up\n"
        + "[ex_jdcf_HelloWorld options ]\n"
        + "-help display this message\n"
        + "-lang [english|spanish|french] specifies which messages to display\n";

    public static void main( String args[] )
    {
            try
            {
                    String language = "english";
                    Framework.initDefaultServices( args );

                    for (int i=0; i < args.length; i++)
                    {
                            if (args[i].equals("-help"))
                            {
                                    System.err.println( usage_ );
                                    System.exit(0);
                            }
                            else if ( args[i].equals("-lang") )
                            {
                                    if ((i+1) >= args.length)
                                    {
                                            System.err.println( usage_ );
                                            System.exit(1);
                                    }
                                    language = args[i+1];
                            }
                    }

                    // Get the ex_jdcf_HelloWorld component configuration from within
the
                    // ex_jdcf_HelloWorld application instance configuration
                    CFGGroup component_cfg = CINFO.getConfig();

                    // get the configuration group containing various message strings
                    CFGGroup messages = component_cfg.getGroup( language );

                    // write the appropriate hello world message to the logger.
                    LOG.info( messages.getAttributeValue("hello_world") );

                    // write a debug message
                    LOG.debug( CINFO.DF_EXAMPLE, "this only prints if the correct debug
flag is set");
                    // shutdown this application
                    AppControl.instance().shutdown(0);
            }
            catch ( DCFException e)
            {
                    LOG.error( -1, "DCFException caught:\n", e );
                    AppControl.instance().shutdown(1);
            }
            catch (Exception e )
            {
                    LOG.error( -1, "Exception caught:\n", e );
                    AppControl.instance().shutdown(1);
            }
    }
}
```

4. Create a wrapper script – The wrapper script is simply a convenience for Java applications. The wrapper invokes the Java interpreter with the required options. (Note that many of the DCF's Java examples may use the same wrapper script – *`jrun_example.pl`* – that takes the class name that defines the `main()` function as an argument.)

   For this example, this script contains:

```
#!/usr/bin/perl -w

use strict;
use English;
use Env qw( DCF_CFG DCF_TMP);
use File::Basename;
my $myname = basename( $0 );
$ENV{"JAVA_NS"}="true";
exec ( "java "
        . "-Dorg.omg.CORBA.ORBClass=com.sun.corba.se.internal.iiop.ORB "
        . "-Dorg.omg.CORBA.ORBSingletonClass=com.sun.corba.se.internal.iiop.ORB "
        . "-DDCF_CFG=\"${DCF_CFG}\" "
        . "-DDCF_TMP=\"${DCF_TMP}\" "
        . "com.lbs.examples.ex_jdcf_HelloWorld.ex_jdcf_HelloWorld "
        . "-appcfg /apps/defaults/ex_jdcf_HelloWorld "
        . "-CDS_a_use_fsys "
        . join( " ", @ARGV ) );
```

The Java arguments seen above are listed and explained below:

`-D property-name=property-value  (multiple instances of this)`

These four are always set for DCF Java applications, and are the Java equivalent of environment variables

`com.lbs.examples.ex_jdcf_HelloWorld.ex_jdcf_HelloWorld`

This is the Java class name, i.e., the class that defines the "main" method.

`appcfg /apps/defaults/ex_jdcf_HelloWorld`

This tells the Application Control adapter where to find the configuration for the application. This configuration data is generated automatically when the component is built.

`-CDS_a_use_fsys`

Tells the Configuration Data Service adapter to use the file system to access configuration data. The default mode is to access configuration data via a server.

`join("",@ARGV)`

Appends any additional parameters that were passed to the script. For the ex_jdcf_HelloWorld application, this would include the "-lang" param.

5. Build the application.

   To build the application, simply type the command

   ```
   perl -S dcfmake.pl
   ```

   Invoking *dcfmake.pl* will perform the following steps for this example:

   a) Read the *cinfo.cfg* file in the current working directory.
   b) Read the component configuration for each `required component` in the *cinfo.cfg*. Component configurations come from the *$DCF_USER_ROOT/devel/cfggen/components* directory. That data was created when *dcfmake.pl* built those components.
   c) Recursively read component configurations for other required components.
   d) Generate the component configuration for this component. This data is written to the file *$DCF_USER_ROOT/devel/cfggen/components/java_app/ex_jdcf_HelloWorld*
   e) Generate the application configuration for this component. This data is written to the file *$DCF_USER_ROOT/devel/cfggen/apps/defaults/ex_jdcf_HelloWorld*
   f) Generate the *CINFO.java* source file in the current directory. The `CINFO` class (which is private to the component's package) contains the debug-flag mask constants, as well as code to initialize and update the debug flags value from the CDS database. `CINFO` also provides convenience mechanisms for getting the instance configuration group for the component within a particular application.
   g) Generate the *LOG.java* source file in the current directory. The LOG class (which is private to the component's package) is simply a wrapper for the DCF LOG interface. It simplifies checking debug flag settings in `CINFO`, and provides message header fields that remain constant for the component. (In C++ pre-processor macros are typically used for this type of work.)
   h) Generate the make file. To avoid confusion with a handcrafted makefile, the file is called *makefile.dcf*.
   i) Invoke "*make -f makefile.dcf*". Any arguments given to *dcfmake.pl* are forwarded to make. After the make completes, the generated makefile is removed. You can have *dcfmake.pl* leave the generated file by using the "`-keep`" option.

6. Update the configuration data service repository.

   The developer can determine when to deploy any newly created or edited configuration data. This can be useful if you are testing with non-default configurations, and do not want the fact that you have rebuilt something to affect your working configuration files. To update the data, execute the command:

   ```
   perl -S update_cds.pl
   ```

   This will copy all files from the temporary areas *`$DCF_USER_ROOT/devel/cfggen`* and *`$DCF_USER_ROOT/devel/cfgsrc`* to the working area: *`$DCF_USER_ROOT/cfg`*. As the files are copied, various macros are expanded, so for example the files in the working config can have the correct port numbers, path names, etc.

The application is now ready to run!

5.5.2. Using the LOG interface – Logging from Java programs

Each Java component has a package private class named "`LOG`". This class is generated by *`dcfmake.pl`* in the file *`LOG.java`*. Component specific debug flags are generated in the package-private file *`CINFO.java`*.

First, the `LOG` adapter must be initialized. Normally, all of the common services are installed at once, during application initialization. This can be done with the lines:

```
com.lbs.LOG_a.LogClient_a.setup( args );
com.lbs.APC_a.AppControl_a.setupORB( args );
com.lbs.CDS_a.CFGDB_a.setup( args );
com.lbs.APC_a.AppControl_a.setup( args );
com.lbs.LogClient_a.setup( args );
```

Messages are logged using the following methods:

```
LOG.info( "this message will always print" );
LOG.error( -1, "this is an error" );
LOG.error( -1, "the stack trace contained in the exception (e) will print\n", e );
LOG.debug(  CINFO.DF_SHOW_GENERAL_FLOW, "this is a conditional debug message" );
```

Generally it is best to keep possibly expensive expressions like

```
("Here is a data set: " + ds.toString() )
```

in conditionals, since in Java (and C#), the args to LOG.debug are evaluated before calling the method, which then may decide to not log anything.

A better approach is to do something like what is illustrated in the following example:

```
if ( CINFO.debug( CINFO.df_SHOW_GENERAL_FLOW )
{
    LOG.debug( "com.oem.module.StoreSCP.DicomDataService_a.storeObject: dimse-message =
" + c_store_rq);
}
```

By wrapping the debug message in a conditional at least you're not doing extra work when you are in non-debug mode.

5.5.3. Using the CDS interface

See language-specific class documentation for `CDS.CFGGroup`, `CDS.CFGAttribute`, and `CDS.CFGDB`.

5.5.4. Using the APC interface

See language-specific on-line documentation for `APC.AppControl`.

5.6. Advanced DICOM Programming Examples

5.6.1. Using StorageCommitmentSCU

The StoreCommitSCU and StoreCommitSCUAgent classes provide the user with an interface to the Storage Commitment Push Model SOP class as a Service Class User. The StoreCommitSCU class allows the user to send a list of DICOM SOP instances to a Storage Commitment SCP for which storage commitment is requested. The StoreCommitSCU class provides the interface for creating an association, creating a transaction UID, and sending the appropriate N-ACTION DIMSE message. The DicomDataService singleton's commitRequestSent method will be called in order to notify the OEM's database that the commit has been requested.

After sending the requests via N-Action, the `StoreCommitSCU` can be configured to hold the outbound association open. Otherwise, the StoreCommitSCUAgent class can be used to wait for inbound associations. In either case, the SCP will send N-Event-Report DIMSE messages back to the SCU (`StoreCommitSCU`). The DicomDataService singleton's commitCompleted method will be called so that the OEM can be updated with the commit completion status from the SCP.

Store related classes are in the `com.lbs.DSS` (DICOM Store Services) package.

5.6.1.1. Example – Send store commit requests, and receive StoreCommitClientListener notifications

Send store commit requests and receive N-Event-Report notifications as objects are committed to long term storage.

See the ex_nstorecommit_scu.exe example for a complete program which can optionally start a StoreCommitSCUAgent in a new thread to receive incoming N-Event-Reports on a new association.

```
com.lbs.DSS.StoreCommitRequest request = new com.lbs.DSS.StoreCommitRequest();

//
// Put together the server_address.
//
String server_address = called_host_ + ":" + called_port_;
LOG.info("Called presentation address = " + server_address);

AssociationInfo ainfo = new AssociationInfo();
RequestedPresentationContext sc_ctx = new RequestedPresentationContext(1,
UID.SOPCLASSSTORECOMMITPUSHMODEL, new String[] { UID.TRANSFERLITTLEENDIANEXPLICIT,
UID.TRANSFERLITTLEENDIAN });

ainfo.calledPresentationAddress(server_address);
ainfo.calledTitle(called_ae_title_);
ainfo.callingTitle(calling_ae_title_);
ainfo.addRequestedPresentationContext(sc_ctx);

scu_ = new StoreCommitSCU(ainfo);
```

```
try
{
   // populate the referenced sop sequence from the command line args
   int count = 0;
   ReferencedSopSequence[] ref_sop_sequence = new
ReferencedSopSequence[file_list_.size()];
   request.transactionUid( DCMUID.makeUID());
   while (count < file_list_.size())
   {
      DicomFileInput dfi = new DicomFileInput( (String)file_list_.elementAt(count));
      dfi.open();
      DicomDataSet dds = dfi.readDataSetNoPixels();
      dfi.close();

      // create a sequence item with the uids
      ReferencedSopSequence ref_sop_sequence_item = new ReferencedSopSequence();
      ref_sop_sequence_item.referencedSopclassUid(
dds.getElementStringValue(DCM.E_SOPCLASS_UID));
      ref_sop_sequence_item.referencedSopinstanceUid(
dds.getElementStringValue(DCM.E_SOPINSTANCE_UID));

      // add the sequence item to the vector of items
      ref_sop_sequence[count] = ref_sop_sequence_item;

      count++;
   }

   // add the vector of sequence items to the request, it will be converted to
   // a sequence element containing one data set item for each object in the
   // vector
   request.referencedSopSequence(ref_sop_sequence);

   scu_.requestAssociation();
   scu_.nAction(request, 10, 10);
   scu_.waitForNEvent( 1 );
   scu_.releaseAssociation();
   exit_status_ = 0;
   }
catch (Exception e)
{
   LOG.error(-1, "Exception caught:" + System.getProperty( "line.separator"), e);
   System.err.println("test failed");
   exit_status_ = 1;
}
finally
{
   try
   {
      if ((scu_ != null) && scu_.connected())
      {
         scu_.releaseAssociation();
      }
   }
   catch (DCSException e)
   {
      LOG.error(-1, "Error releasing Association, aborting", e);
      scu_.abortAssociation();
      exit_status_ = 1;
   }
}
```

5.6.2. Using Java MWLClient Classes

A common application of the DICOM protocol is querying an image manager for a list of scheduled procedure steps. The simplest way to accomplish this using the DCF is to use the `MWLSCU` class. `MWLSCU` provides a high level mechanism for interoperating with MWL SCPs via C-FIND DIMSE messages.

`MWLSCU` functions identically to the `QRSCU` class (see the `QRClient` example for code examples, section 5.3.7).

5.6.2.1. Example – ex_jmwl_scu

See *`$DCF_ROOT/devel/jsrc/com/lbs/examples/ex_jmwl_scu/ex_jmwl_scu.java`* for a command line application that uses this class.

5.6.3. Using Java MPPSClient Classes

The `MPPSClient` is used to communicate with Modality Performed Procedure Step Service Class providers or servers.

`MPPSClient` creates and updates instances of Modality Performed Procedure Step objects. It sends N-Create and N-Set DIMSE messages to an MPPS SCP or server. The user instructs the `MPPSClient` to connect to the SCP, and uses the `n_set()` and `n_create()` methods to send the appropriate DIMSE messages.

5.6.3.1. Example – ex_jmpps_scu

See `$DCF_ROOT/devel/jsrc/com/lbs/examples/ex_jmpps_scu/ex_jmpps_scu.java` for a complete console application example.

The method below will send the appropriate DIMSE N-CREATE or N-SET message to a MPPS Server.

```
    public void runJob()
        throws DCSException
    {
        //
        // Decide on whether to do an n-create or n-set
        //
        if( f_opt_create_ && f_opt_set_ )
        {
            throw new DCSException( "Cannot n-set and n-create at the same time" );
        }

        //
        // Put together the server_address.
        //
        String server_address = host_ + ":" + port_;
        LOG.info( "Called presentation address = " + server_address );

        AssociationInfo ainfo = new AssociationInfo();
        RequestedPresentationContext mpps_ctx =
            new RequestedPresentationContext( (byte)1, UID.SOPPERFORMEDPROCEDURESTEP, new
String[] {ts_uid_} );

        ainfo.calledPresentationAddress( server_address );
        ainfo.calledTitle( called_ae_title_ );
        ainfo.callingTitle( calling_ae_title_ );
        ainfo.addRequestedPresentationContext( mpps_ctx );

        scu_ = new MPPSSCU( ainfo );

        scu_.requestAssociation();
        CFGGroup mpps_cfg = null;
        try
        {
           mpps_cfg = CFGDB.instance().loadGroup( cfg_file_, true );
        }
           catch( CDSException cds_e )
        {
           LOG.error( -1, "Error loading mpps cfg file " + usage(), cds_e );
           System.exit( -1 );
        }
        LOG.info( "MPPS CFGGroup = " + mpps_cfg );
        DicomDataSet ds = new DicomDataSet( mpps_cfg );
        ModalityPerformedProcedureStep procedure =
           new ModalityPerformedProcedureStep( ds );

        if( f_opt_create_ )
        {
           scu_.nCreate( procedure, 10 );
        }
        else if( f_opt_set_ )
        {
           scu_.nSet( procedure, 10 );
        }

        scu_.releaseAssociation();
        exit_status_ = 0;
    }
```

5.6.4. Using Java Store, Q/R, and MWL Server-Related Classes

A common application of the DICOM protocol is in creating an image archive. An OEM may have special requirements for how images and patient information are stored in a database. The DCF provides APIs that are structured such that the OEM can easily customize the handling of image or other DICOM datasets without the need to deal with the mechanics of negotiating associations, handling sockets, PDUs or DIMSE messages.

The `DicomDataService` interface provides the mechanism for customizing the handling of DICOM datasets. Generic DCF protocol handling objects such as `StoreSCP`, `QRSCP` (Query Retrieve), `MWLSCP` (Modality Worklist) invoke `DicomDataService` methods to access the local storage facilities. The reference implementation adapter for the `DicomDataService` interface stores objects in the file system and provides minimal searching capabilities to support testing. Other implementations or adapters can be written that behave differently.

5.6.4.1. Example – ex_jstore_scp

The example *`$DCF_ROOT/devel/jsrc/com/lbs/examples/ex_jstore_scp`* shows a simple storage server that sends incoming DICOM objects to the file system using the default `DicomDataService_a` (DICOM Data Service adapter) in *`$DCF_ROOT/devel/jsrc/com/lbs/DDS_a`*. By installing a particular `DicomDataService_a`, all incoming DICOM images are passed to the `storeObject()` method defined in that class.

The source file *`ex_jstore_scp.java`* contains the function `main()` which installs the `DicomDataService` adapter, and enters the loop which waits for incoming DICOM associations.

Below is a listing of the source code:

```
//=========================================================================
// Copyright (C) 2005, Laurel Bridge Software, Inc.
// 160 East Main St.
// Newark, Delaware 19711 USA
// All Rights Reserved
```

```
//=============================================================================

package com.lbs.examples.ex_jstore_scp;

import com.lbs.CDS.*;
import com.lbs.CDS_a.*;
import com.lbs.APC.*;
import com.lbs.APC_a.*;
import com.lbs.LOG_a.*;
import com.lbs.DCS.*;
import com.lbs.DSS.*;

/**
* Example Java Store Service Class Provider (SCP).
* Uses DCF StoreServer and StoreSCP classes, along with a
* locally defined DicomDataService adapter to implement
* a DICOM Storage server application.
*/
public class ex_jstore_scp
   implements AssociationConfigPolicyManager, AssociationListener
{
   private AssociationManager mgr_;
   private StoreServer store_server_;
   private VerificationServer verification_server_;

   public ex_jstore_scp( String args[] )
      throws DCSException, CDSException
   {
      LOG.info( "Creating new ex_jstore_scp" );

      // Create an AssociationManager
      // You can override association manager settings after creating,
      // e.g., tcp-port, max-number-concurrent-assocs, max-total-assocs, etc.
      // For now, it will get them from proc-cfg/java_lib/DCS/AssociationManager.
      //
      mgr_ = new AssociationManager();

      // Run this line if you want AssociationManager to call
      // our getSessionSettings() each time an association is
      // requested. There can only be one association-config-policy-manager.
      // If you don't do this, you also don't need the
      // "implements AssociationConfigPolicyManager" above.
      mgr_.setAssociationConfigPolicyMgr( this );

      // Run this line if you want AssociationManager to call
      // our beginAssociation()/endAssociation() each time an association is
      // started/ended. There can be any number of association-listeners.
      // If you don't do this, you also don't need the
      // "implements AssociationListener" above.
      mgr_.addAssociationListener( this );

      // Create a StoreServer object which will register as an AssociationListener,
      // and create a StoreSCP object each time an association is requested.
      LOG.info( "Creating new StoreServer Object" );
      store_server_ = new StoreServer( mgr_ );

      // Create a VerificationServer object (creates VerificationSCP's)
      LOG.info( "Creating new VerificationServer Object" );
      verification_server_ = new VerificationServer( mgr_ );
   }

   /**
   * Ask AssociationManager to loop, waiting for connection requests, and handling
   * associations. Each time a socket connect is detected, AssociationManager
   * creates an instance of AssociationAcceptor that will handle all I/O for that
   * association.
   */
   public void runServer()
      throws DCSException
   {
      LOG.info( "Starting ex_jstore_scp" );
      mgr_.run();
   }

   /**
   * Implementation of AssociationConfigPolicyManager interface.
   * Here is where we optionally implement our custom policy of selecting
   * the configuration used for a particular association.
   */
   public DicomSessionSettings getSessionSettings( AssociationAcceptor assoc )
      throws AssociationRejectedException, AssociationAbortedException
   {
      // Get the association info from the current acceptor.
      AssociationInfo ainfo = assoc.getAssociationInfo();

      // Get some fields from the association info
      String called_ae  = ainfo.calledTitle();
      String calling_ae = ainfo.callingTitle();

      //
      // Demonstrate a configuration policy that selects/creates session settings
      // based on the called-AE-title.
      //
      LOG.info( "ex_jstore_scp: getting session settings for '" + called_ae + "'" );

      //
      // Start with the default session settings. (Values for this come from
      // the current app/proc config group java_lib/DCS/default_session_settings.)
      //
      DicomSessionSettings session_settings = new DicomSessionSettings();

      //
      // We can do different things based on the AE, either by loading
      // a particular session settings configuration, or by modifying
      // an existing settings object.
      //
      if ( calling_ae.trim().equals( "ECHO_SCU" ) || called_ae.trim().equals( "ECHO_SCP"
) )
      {
         // do nothing if this is an echo request
      }
      else if ( called_ae.trim().equals( "StoreSCP1" ) )
      {
         // Change the default settings by adding a
         // pre- and post-processing scripts.
         //
         // Note that you either need a full path to the script, or it must be in the
         // current working directory of this process (if you start this app via
         // the web interface, that cwd will be $DCF_ROOT/httpd/cgi-bin).
         session_settings.setPreAssociationScript( "perl pre_assoc.pl" );
         session_settings.setPostAssociationScript( "perl post_assoc.pl" );
      }
      else if ( called_ae.equals( "StoreSCP2" ) )
      {
         // Add a filter set (by reference) to the default session settings.
         String filter_cfg_name = "/dicom/filter_sets/example_filter.cfg";

         session_settings.setInputFilterCfgName( filter_cfg_name );
      }
      else
      {
         // Load the session settings from a CFGGroup, based on the name
         // of the called AE.
         String session_cfg_name = "/dicom/example_session_" + called_ae + ".cfg";
         CFGGroup session_cfg = null;
         try
         {
            session_cfg = CFGDB.instance().loadGroup( session_cfg_name, true );
            session_settings = new DicomSessionSettings( session_cfg );
         }
         catch( CDSException e )
         {
            LOG.error( -1, "Unable to load session cfg: " +
               session_cfg_name + " - " + e + "\nRejecting the association..." );

            // Throwing AssociationRejectedException results in the association being
rejected.
            // Alternately, we could let them in with the default, or some other
settings.
            // Note: previous version required an AssociateRejectPDU here, now the
exception
            // class builds that behind the scenes.
            throw new AssociationRejectedException(
                                       PDUAssociateReject.RESULT_PERMANENT,
                                       PDUAssociateReject.SOURCE_SERVICE_USER,
                                       PDUAssociateReject.REASON_BAD_CALLED );
         }
      }

      return session_settings;
   }

   /**
   * Optional implementation of AssociationListener interface.
   *
   * Indicates that a new association has been created.
   * The AssociationAcceptor has selected configuration
   * settings for the association, and has processed the
   * A-Associate-Request PDU. The association has not yet
   * been accepted.
   * @param assoc the object handling the association.
   * @throws AssociationRejectedException Indicates that
   * the association should be rejected immediately.
   * @throws AssociationAbortedException Indicates that
   * the association should be aborted immediately.
   */
   public void beginAssociation( AssociationAcceptor assoc )
      throws AssociationRejectedException, AssociationAbortedException
   {
      LOG.info("Association has started:\n" + assoc.getAssociationInfo());
   }

   /**
   * Optional implementation of AssociationListener interface.
   *
   * Indicates that an association has ended.
   * @param assoc the object handling the association.
   */
   public void endAssociation( AssociationAcceptor assoc )
   {
      LOG.info("Association has ended." );
   }

   /**
   *
   * Invoke with the following optional arguments:
   *  -CDS_a_use_fsys     Do not require DCDS_Server.
   *                   default is to expect it to
   *                   be running. If this option is
   *                   enabled, shutdown messages and
```

```
    *                    runtime changes to debug settings
    *                    are unsupported.
    *   -apc <proc-attr-name>=<value>
    *                    Overrides any setting in the process
    *                    configuration.
    */
   public static void main( String args[] )
   {
      try
      {
         LOG.info( "Setting up basic services" );
         AppControl_a.setupORB( args );

         // uncomment if you want to default to no DCDS-Server
         //CFGDB_a.setFSysMode( true );
         CFGDB_a.setup( args );
         AppControl_a.setup( args, CINFO.instance() );

         // uncomment if you want to default to console logging - else
         // LOG_a config specifies output files, possibly sends to DLOG_Server, etc.
         // LOGClient_a.setConsoleMode( true );
         LOGClient_a.setup( args );

         // install our custom DicomDataService.
         LOG.info( "Setting up DDS_a service" );
         DicomDataService_a.setup( args );

         // create store scp object
         ex_jstore_scp scp = new ex_jstore_scp( args );

         // start accepting connections
         scp.runServer();

         LOG.info( "ex_jstore_scp exiting!" );
      }
      catch( Exception e )
      {
         LOG.error( -1, "ERROR: " + e );
      }

      if ( AppControl.isInitialized() )
      {
         AppControl.instance().shutdown( 0 );
      }

      System.exit( 0 );
   }
}
```

Note: To support MWL and Query Retrieve, the `findObjects()`, `findObjectsForTransfer()`, and `loadObjects()` methods would have to be implemented. See example implementations, which are listed below.

5.6.4.2. Example – ex_jqr_scp

The directory *`$DCF_ROOT/devel/jsrc/com/lbs/ex_jqr_scp`* shows a simple Query/Retrieve server that searches a "canned" set of DICOM objects in response to C-FIND requests and C-MOVE and C-GET requests and performs the appropriate matching. It either returns the list of found objects for a C-FIND or performs C-STORE operations if a C-MOVE or a C-GET was requested.

See *`$DCF_ROOT/devel/jsrc/com/lbs/examples/ex_jqr_scp/ex_jqr_scp.java`*. This program is almost identical to *`ex_jstore_scp.java`* with the exception that it uses `QRServer` class instead of `StoreServer` class. The directory *`$DCF_ROOT/devel/jsrc/com/lbs/ex_jqr_scp`* shows a simple Query/Retrieve server that searches a "canned" set of DICOM objects in response to C-FIND requests and C-MOVE and C-GET requests and performs the appropriate matching. The list of "canned" objects is created from thefiles found in *`$DCF_ROOT/test/qr`* directory. It either returns the list of found objects for a C-FIND or performs C-STORE operations if a C-MOVE or a C-GET was requested.

5.6.4.3. Example – ex_jmwl_scp

The directory *`$DCF_ROOT/devel/jsrc/com/lbs/ex_jmwl_scp`* shows a simple Modality Worklist server that searches a "canned" set of DICOM objects in response to C-FIND requests performs the appropriate matching. It returns the list of found objects for a C-FIND. The list of "canned" objects is created from thefiles found in *`$DCF_ROOT/test/worklist`* directory.

See *`$DCF_ROOT/devel/jsrc/com/lbs/examples/ex_jmwl_scp/ex_jmwl_scp.java`*. This program is almost identical to *`ex_jstore_scp.java`* with the exception that it uses `MWLServer` class instead of `StoreServer` class.

5.6.4.4. Using the MWL Server as an MPPS Server

Both the C# and Java worklist server examples (`ex_nmwl_scp` and `ex_jmwl_scp`) are also MPPS servers by default.

MPPS N-CREATE messages and N-SET messages are stored to the currently installed `DicomDataService` adapter. If you are using the default file system mode `DicomDataService` adapter, you can tell `DicomDataService` to store the both the command and data data-sets from N-CREATE and N-SET messages. This is useful if you want to be able to tell whether the stored object came from an N-CREATE or an N-SET message as this information is sent in the Command Data set of a DIMSE message. This functionality can be turned on by setting the appropriate CFG attribute to "*`YES`*", e.g.,

```
/apps/defaults/ex_jmwl_server/java_lib/DDS_a/save_command_data YES
```

5.6.4.5. Example – Server Objects

The default example implementations for `findObjects()`, `findObjectsForTransfer()`, and `loadObjects()` can be found in $DCF_ROOT/devel/jsrc/com/lbs/DDS_a/DicomDataService_a.java. These are also similar to the examples found in *`ex_jstore_scp.java`*.

5.6.4.6. Example – StoreSCP: Implementing a custom storeObject method

The following example illustrates how you could create your own `DicomDataService` adapter and implement the `storeObject()` method.

Note: your `DicomDataService` adapter class can have any name or be part of any package. The only requirements on your implementation are that:

- It must implement the abstract class `com.lbs.DDS.DicomDataService`; and
- You must "install" that implementation before the first time it will be used: you should add a static method called `setup()` to your implementation to "install" it.

Each time an incoming C-STORE-RQ message is received by your application, the `cStoreRQ()` method of the `StoreSCP` object is called. This method calls `DicomDataService.instance().storeObject( AssociationAcceptor, DimseMessage)`.

This calls the installed implementation of `DicomDataService`, which in this case will be your customized adapter.

Notes:

- If an error occurs in `DicomDataService.storeObject()`, you should throw an exception, which is logged by `StoreScp.cStoreRq()` method; then `StoreSCP` sets the error text of the exception into the DIMSE response message that will be returned to the Store SCU.
- You should do any logging you need to do regarding errors before throwing that exception.
- If you wish to create a new directory for each association or file, you should do that in the `storeObject()` method as you will have access to the `AssociationInfo` object that is contained in the `AssociationAcceptor` object passed into the `DicomDataService.storeObject()` method.

The following illustrates what part of your `DicomDataService` adapter would look like:

```
package com.oem.StoreTest;

import com.lbs.DDS.*;

public class OEMDataServiceAdapter extends com.lbs.DicomDataService

{

      //protected constructor
      //You can only create one of things be calling the public static setup method.
      protected OEMDataServiceAdapter( String args[] )
      {

            //put initialization code

      }

      //no default constructors allowed
      protected OEMDataServiceAdapter
      {

            LOG.error( -1, "illegal use of default constructor" );

      }

      //Here’s where you add your code to store images to your backend
      public DicomPersistentObjectDescriptor storeObject(
                  AssociationAcceptor association_acceptor,
                  DimseMessage c_store_rq )
            throws DDSException
      {
            //Your implementation goes here
      }

      //do something like the following for the other abstract methods
      //in DicomDataService base class.
      public abstract DicomDataSet loadObject(
            DicomPersistentObjectDescriptor dpod,
            boolean f_read_pixel_data )
                  throws DDSException
      {
            throw new DDSException( “loadObject unimplemented”)
      }

      //Add a public setup method to install your implementation
      public static synchronized void setup( String args[] )
```

```
            throws DDSException
      {
            OEMDataServiceAdapter instance = new OEMDataServiceAdapter( args );
            //base class method that will set a new installed implementation
            setInstance( instance );
      }

}  //end of class OEMDataSetAdapter
```

In your application's `main()` method, before you begin accepting incoming associations, call your `setup()` method. For example:

```
public static void main( String args[] )
{

...

      com.oem.StoreTest.OEMDataServiceAdapter.setup( args );

      //begin accepting associations

...

}
```

You could, in fact, simply replace the following line in *ex_jstore_scp.java*

```
      DicomDataService_a.setup( args );
```

with this line:

```
      com.oem.StoreTest.OEMDataServiceAdapter.setup( args );
```

and then *ex_jstore_scp.java* will use your new `DicomDataService` adapter's `storeObject()` method whenever an incoming DICOM Image is stored.

The other methods in the `DicomDataService` interface support operations like finding previously stored objects (`findObjects()` method) or loading previously stored objects (`loadObject()` method); this functionality is used to support the Query/Retrieve or Worklist SOP classes.

5.6.4.7. Additional coding examples:

- For additional information on how the `DicomDataService` is called, see the example in section 6.6.4.3.
- For information on customizing behavior based on custom `DicomSessionSettings`, see the notes in section 6.6.4.4.

5.6.4.8. Writing a Custom DICOM SCP

You can extend *%DCF_ROOT%\devel\jsrc\com\lbs\examples\ex_j_oem_scp\ex_j_oem_scp.java.*

5.7. DICOM Compression Transfer Syntax Support for Java

DCF Java applications can handle DICOM datasets in any transfer syntax for non-pixel data operations provided that compression pass through mode is turned on (except for DICOM Deflated Little Endian Syntax and JPIP Transfer syntaxes).

DCF Java applications can compress and decompress data sets in these encapsulated transfer syntaxes:

• 1.2.840.10008.1.2.4.5	RLE Lossless
• 1.2.840.10008.1.2.4.50	JPEG 8 bit lossy

1.2.840.10008.1.2.4.51	JPEG 12 bit lossy
1.2.840.10008.1.2.4.57	JPEG lossless
1.2.840.10008.1.2.4.70	JPEG lossless (predictor selection=1)
1.2.840.10008.1.2.4.90	JPEG-2000 lossless
1.2.840.10008.1.2.4.91	JPEG-2000 lossy

RLE Lossless transfer syntax is supported for compression of single frame data sets. RLE Lossless transfer syntax is supported for the decompression of single frame and multi-frame data sets.

Look at the settings under the DCS section of an application configuration file or in a DCS component configuration file to see options that can be configured for compression.

Note: If you are using the Aware, Inc., JPEG library, that this does not support .57.

5.7.1. Example – ex_jdcf_filter: Uncompressed to Compressed

Use ex_jdcf_filter to convert an existing DICOM file in a non-compressed transfer syntax to JPEG-2000 compressed.

Create the following config file, *compress.cfg*:

```
[ filter_info ]
input_file = C:/temp/test.dcm
output_file = C:/test_j2k.dcm
output_ts_uid = 1.2.840.10008.1.2.4.90
part10_output = true
filter_count = 0
```

Run the ex_*jdcf_filter* app, using the Perl wrapper for convenience:

```
ex_jdcf_filter.pl -f file:/compress.cfg
```

5.7.2. Example – ex_jdcf_filter: Compressed to Uncompressed

Use *ex_jdcf_filter* to convert an existing DICOM file in a JPEG-2000 compressed transfer syntax to Explicit-Little-Endian uncompressed.

Create the following config file *decompress.cfg*:

```
[ filter_info ]
input_file = C:/temp/test_j2k.dcm
output_file = C:/test_decompressed.dcm
output_ts_uid = 1.2.840.10008.1.2.1
part10_output = true
filter_count = 0
```

Run the *ex_jdcf_filter* app, using the Perl wrapper for convenience:

```
ex_jdcf_filter.pl -f file:/decompress.cfg
```

5.8. JAI – DCF integration for Java

The DCF includes JAI (Java Advanced Imaging) integration, which provides facilities for converting between DICOM files/datasets and various supported JAI file types.

Notes:

- The JAI is not used by the DCF for DICOM image compression.
  The class com.lbs.DCS.JAIUtil still uses Java Advanced Imaging (JAI) and Java Advanced Imaging I/O (JAI I/O). If you're not using that class, and you are not using DicomEncapsulatedCodecJAI, which is not used by default, then you don't need JAI or JAI I/O.
- The JPEG and JPEG2000 codec functionality is now provided by default by our DicomTSCW (Transfer Syntax Codec Wrapper) plugin library. This is a C/C++ implementation of the low level compress/decompress functionality that is shared by C++, Java, and C# DCF implementations. One of the reasons we adopted this approach, and retired the JAI/ImageIO based codecs as the default, was that there was limited or no support for 64 bit JAI I/O.

Three example applications are provided to demonstrate this functionality.

5.8.1. Example – ex_jdcf_dcm2jai: Convert a DICOM file to a JAI type

`ex_jdcf_dcm2jai` - converts a DICOM file to one of the supported JAI types. This application can convert DICOM files to/from JAI types: jpg, JPEG2000, bmp, tiff, png, pnm, etc.

Support is provided for DICOM images that are:

- Grayscale, 8 or 16 bit
- MONOCHROME2,
- RGB pixel interleaved, and
- RGB planar.

Data that is signed (pixel-representation==1) is first converted to unsigned, since many of the destination formats or supporting software do not handle signed data properly. Certain conversions are not supported, since the destination image format may not support the data format in the DICOM file. For instance, a 16 bit DICOM image can not be converted directly to JPEG.

Facilities for performing certain other conversions are available, in particular the option “-8” will cause a window-level LUT to be created and applied to 16 bit data so that it can be saved in an 8-bit format, e.g., JPEG. The option “-16” will cause a linear LUT to be created and applied to 9-15 bit data. Currently, the JPEG-2000 codec saves 9-15 bit data correctly, but when j2k files are reloaded, the data is assumed to be 16 bit, and may display proportionally darker than the original.

Example conversions:

convert 12 bit DICOM to JPEG:

```
jrun_example.pl com.lbs.examples.ex_jdcf_dcm2jai.ex_jdcf_dcm2jai -i
12bit.dcm -t jpg -o 8bit.jpg -8
```

convert 8 bit DICOM to bmp:

```
jrun_example.pl com.lbs.examples.ex_jdcf_dcm2jai.ex_jdcf_dcm2jai -i
8bit.dcm -t bmp -o 8bit.bmp
```

convert 12 bit DICOM to JPEG2000:

```
jrun_example.pl com.lbs.examples.ex_jdcf_dcm2jai.ex_jdcf_dcm2jai -i
12bit.dcm -t jpeg2000 -o 16bit.j2k -16
```

Note: you could create an individual “**.pl**” wrapper script that would do the same as running `jrun_example.pl` or as:

```
java com.lbs.examples.ex_jdcf_dcm2jai
```

This functionality can easily be added to your SCU or SCP applications for more seamless integration.

5.8.2. Example – ex_jdcf_jai2dcm: Convert JAI types to a DICOM file

The “-h <header-cfg>” option allows you to attach additional DICOM tags to the created image.

Examples:

jrun_example.pl com.lbs.examples.ex_jdcf_jai2dcm.ex_jdcf_jai2dcm -i 8bit.jpg -o 8bit.dcm

jrun_example.pl com.lbs.examples.ex_jdcf_jai2dcm.ex_jdcf_jai2dcm -i test.j2k -o test.dcm -h dcm_header.cfg

Note: for a fuller description of this example, run:

jrun_example.pl com.lbs.examples.ex_jdcf_jai2dcm.ex_jdcf_jai2dcm -help

Currently only grayscale source images are properly converted. *Support for various color formats will be included in the next release.*

5.8.3. Example – ex_jdcf_dcmview: View a DICOM Image file

This utility very simply displays the selected image.

Example:

jrun_example.pl com.lbs.examples.ex_jdcf_dcmview.ex_jdcf_dcmview test.dcm

5.8.4. Example – Writing a JPEG Image with Java

The following code snippets illustrate the basic steps required to output a JPEG version of a DICOM image.

```
// for writing JPEGs
import java.io.*;
import com.sun.image.codec.jpeg.*;

//==============================================================
// Saving Dicom image as JPEG
//==============================================================

// Assuming we have the following Strings:
// String dicomfilename;  // input
// String jpegfilename;   // output

// Step 1: Get Dicom image as a RenderedImage
//--------------------------------------------------------------
// Can be done with a DicomDataSet, using the JAIUtil package,
// and the following sequence:

BufferedImage b_image;
try
{
   DicomFileInput dfi = new DicomFileInput( dicomfilename );
   DicomDataSet dds = dfi.readDataSet();
   RenderedImage r_image = JAIUtil.dataSetToRenderedImage(dds, false, false);

   // Cast to BufferedImage (which is superclass of RenderedImage)
   b_image = (BufferedImage)r_image;
}
catch(Exception e)
{
   System.err.println("Error: " + e.toString());

}

// Step 2: Create an output file stream
```

```
//---------------------------------------------------------------
FileOutputStream filestream;
try
{
   filestream = new FileOutputStream(jpegfilename);
}
catch (Exception e)
{
   System.err.println("Error opening output file: " + e.toString());
   filestream = null;
}

// Step 3: Create a JPEGImageEncoder
//---------------------------------------------------------------
// Making use of the com.sun.image.codec package:
JPEGImageEncoder encoder;
try
{
   encoder = JPEGCodec.createJPEGEncoder(filestream);
}
catch (Exception e)
{
   System.err.println("Error encoding image: " + e.toString());
   encoder = null;
}

// Step 4: Perform the encoding, which writes the file
//---------------------------------------------------------------
try
{
   encoder.encode(b_image);
}
catch (Exception e)
{
   System.err.println("Error encoding image: " + e.toString());
}

// Step 5: Close the file
//---------------------------------------------------------------
try
{
   filestream.flush(); // probably not necessary
   filestream.close();
}
catch (Exception e)
{
   System.err.println("Error closing output file: " + e.toString());
}
```

6. C# Programming Examples

This section presents a variety of C# programming examples for common DICOM integration tasks. See *$DCF_ROOT/devel/cssrc/examples/* for the complete working source code for these and additional examples.

This chapter includes the following sections:

- DICOM Programming Examples section shows how simple DICOM related tasks are performed.
- Common Services Examples section covers use of the DCF framework services.
- Advanced DICOM Programming Examples covers some more complex server concepts.

For additional information, see also

- Chapter 7 – The DCF Development Environment
- Chapter 13 – Deploying a DCF-based application
- Section 6.4 Deploying a Simple C# Standalone Application

6.1. Running Example Servers

Running the DCF tools and/or servers via the DCF Remote Service Interface generally makes running these examples easier. Taking this approach allows easy access to convenient tools for starting and stopping DCF server processes, viewing log files, and controlling trace/debug settings.

Start the Apache web server and open the DCF Remote Service Interface. In a Windows environment this is all handled for you by the startup script:

```
Select "Start"  → "All Programs"  → "DICOM Connectivity Framework" →
"DCF Service Interface"
```

This command runs an Apache web server in its own window and invokes the default browser client to display the DCF home page, called the “DCF Remote Service Interface”.

Alternately, if you prefer a manual approach, type “*run_apache.pl*” from a DCF command window, and then use your favorite web browser to browse to “*localhost:8080*”, which will display the DCF home page.

6.2. Using the DCF with MS Visual Studio .NET 2003/2005 for C#

Notes:

- You must launch Visual Studio from a DCF Command Prompt so that the correct environment is available to Visual Studio (the environment file is *`%DCF_ROOT%\dcf_vars.bat`* if you want to permanently set the environment, but that is not recommended).
  `Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Command Prompt"`
- DCF assemblies are registered in the .NET "Global Assembly Cache" during the toolkit installation. This removes the restriction that the DLL files must by collocated with any executable that uses them.

6.2.1. Opening an Existing C# Example Project

You must launch Visual Studio so that it starts with the DCF environment. This may be done in a few ways:

1. You may launch it from the Start menu:
   `Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "Start Visual Studio with DCF environment"`
2. There may also be a shortcut on the Windows Desktop that does the same thing as the Start Menu option, if you chose to create the shortcut during the DCF installation process.
3. Alternately, you may launch Visual Studio from a DCF Command Prompt:
   `Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Command Prompt"`

   Once you are at a DCF Command Prompt, you now have two choices:

   - You may start Visual Studio by typing *`devenv`* and then navigate to an example project and open it using Visual Studio's GUI menus.

   Or

   - You can change directory to the C# examples directory of interest and run the desired `.csproj` file, which will cause Visual Studio to start and open the selected project.

Assume you choose to work from the DCF Command Prompt and choose the second option and you are interested in the example, `ex_ndcf_dump`. (This example shows how to read a DICOM data set from a file, and write a formatted representation of that data to the console.)

- Open a DCF Command prompt:
  `"Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Command Prompt"`
- Change to the example directory:
  `cd %DCF_ROOT%\devel\cssrc\examples\ex_ndcf_dump`
- To open the example project in Visual Studio, type the name of the project file and hit the `Enter` key:
  `ex_ndcf_dump.csproj`
- Alternately, to run the example and not open Visual Studio, type the name of the example and hit the `Enter` key. For this example the command line invocation requires a file to dump:
  `ex_ndcf_dump %DCF_ROOT%\test\images\test.dcm`

You may select other C# examples to view and/or run by making the appropriate substitutions. All C# examples are found in: `%DCF_ROOT%\devel\cssrc\examples`.

6.2.2. Quick Start - Using ***create_cs_comp.pl*** to generate VS project files and source code

You can run the script *`%DCF_ROOT%\bin\create_cs_comp.pl`*. This interactive script will generate a C# source file template and a cinfo.cfg file. It will then invoke dcfmake.pl which creates .csproj file, and DCF metadata files LOG.cs and CINFO.cs You can then open the .csproj file in Visual Studio 2003/2005 and immediately begin writing your DICOM application.

Launch Visual Studio from a DCF Command Prompt, see note above in Section 6.2.1.

6.2.3. Using ***dcfmake.pl*** to generate a ***.csproj*** file

If you look at the directory *`%DCF_ROOT%\devel\cssrc\examples\ex_necho_scu`*, you will see a file called *`cinfo.cfg`*. See the "HelloWorld" example for information on *`cinfo.cfg`* files. When run, *`dcfmake.pl`* uses the *`cinfo.cfg`* file (short for component information configuration file) to determine what .NET assemblies and DCF configuration information a given application is going to need. With this information *`dcfmake.pl`* can generate a *`.csproj`* file.

There are two basic approaches that you may use to get started:

- Run *perl –S* `dcfmake.pl -g` to generate files (*`LOG.cs`* and *`CINFO.cs`*) only and not run the compiler.
- Run *perl –S* `dcfmake.pl -g -k` to generate the *`LOG.cs`* and *`CINFO.cs`* files and the *`.csproj`* file, but not run the compiler. After this you can use the VS IDE for building and debugging.
- Run *perl –S* `dcfmake.pl` once to generate the *`LOG.cs`* and *`CINFO.cs`* files and a *`.csproj`* file to get you started, then you may edit the project (*`.csproj`*) from the IDE, but *be wary* that (when or if) you need to run *`dcfmake.pl`* again your modified project file *WILL* be over written.

Launch Visual Studio from a DCF Command Prompt, see note above.

6.2.4. Manually Creating C# Projects from MS Visual Studio 2003/2005 IDE

Launch Visual Studio from a DCF Command Prompt, see note above in Section 6.2.1.

To use a DCF assembly in your .cs source file, for example "`LaurelBridge.DCS`", which is the "DICOM Core Services" class library, follow these steps:

- Add "`using LaurelBridge.DCS`" to the top of your .cs source file:
- Then from the Visual Studio.NET toolbar, select *`Project->Add Reference…`*.
- Make sure the .NET tab is selected. Then click "*`Browse`*".
- Go to the directory *`%DCF_ROOT%\lib`* and select the file *`LaurelBridge.DCS.dll`*.

You should follow this same process for any assembly you wish to use directly. Note that currently, all DCF C#.Net assemblies are prefixed with "LaurelBridge.". You should not attempt to reference the standard C++ dll's (prefixed with "DCF_") from C# projects.

6.3. DICOM Programming Examples

6.3.1. DICOM File Access

Reading and writing DICOM format files is easy using the `DicomFileInput` and `DicomFileOutput` classes. Files can be read in Implicit Little Endian, Explicit Big or Little Endian transfer syntax encodings, in either the Part-10 format or the Mallinckrodt (*de facto* non-part-10) format.

The DCF allows the OEM to add additional transfer syntax codec objects that are then available to all DCF based applications, to support additional transfer syntaxes, such as proprietary compression or encryption formats.

Using the DCF Filter plug-in technology users or developers can add one or more data set filters to `DicomFileInput` or `DicomFileOutput` objects. You can choose ready to use DCF filters or create a custom filter. See the section on Filters for more information. (See Appendix D:)

6.3.1.1. Example – Open a DICOM file

Open a DICOM file and display a particular element's value (for speed, don't read the pixel data element).

```
DicomFileReader dfi = new DicomFileReader("myfile.dcm");
DicomDataSet dds = dfi.readDataSetNoPixels();
System.Console.WriteLine("patients name = {0}",  dds.getElementStringValue(
DCM.E_PATIENTS_NAME ) );
```

```
//
// or get the DicomElement and use its ToString() method to show value, length, etc
//
DicomElement e = dds.findElement( DCM.E_PATIENTS_NAME );
System.Console.WriteLine("patients name element = {0}", e );
```

6.3.1.2. Example – Create a DICOM file

Create a new DICOM file containing a minimal image, explicitly setting attributes

```
// create a simple image with a horizontal gradient wedge
int rows = 480;
int cols = 640;
byte[] pixels = new byte[rows*cols];
for (int r = 0; r < rows; r++ )
{
int row_offset = r*cols;
   for ( int c = 0; c<cols; c++ )
   {
      pixels[row_offset+c] = (byte)c;
   }
}

// Create a DicomDataSet
DicomDataSet dds = new DicomDataSet();

//
// Create image header elements using the DicomDataSet convenience methods:
//    insert( AttributeTag, string )
//    insert( AttributeTag, int )
// These will use DicomElementFactory to create the appropriate DicomElement subclass
// and add it to the data set.
//
// Note also the use of DicomDataDictionary to create a new UID.
```

```
//
dds.insert( DCM.E_SOPCLASS_UID, DCM.UID_SOPCLASSUS );
dds.insert( DCM.E_SOPINSTANCE_UID, DicomDataDictionary.makeUID() );
dds.insert( DCM.E_ROWS, rows );
dds.insert( DCM.E_COLUMNS, cols );
dds.insert( DCM.E_PHOTOMETRIC_INTERPRETATION, "MONOCHROME2" );
dds.insert( DCM.E_BITS_ALLOCATED, 8 );
dds.insert( DCM.E_BITS_STORED, 8 );
dds.insert( DCM.E_HIGH_BIT, 7 );
dds.insert( DCM.E_SAMPLES_PER_PIXEL, 1 );
dds.insert( DCM.E_PIXEL_REPRESENTATION, 0 );
//
// for VM>1, we create elements explicitly
//
DicomOBElement e_pixel_data = new DicomOBElement( E_PIXEL_DATA, pixels );
dds.insert( e_pixel_data );

// save to a file
DicomFileOutput dfo = new DicomFileOutput("myfile.dcm");
dfo.writeDataSet( dds );
dfo.close();
```

6.3.1.3. Example – Create a DICOM file from Config Group Data

Create a new DICOM file using text data provided in the form of a CFGGroup

```
// create a CFGGroup object either by using CFGDB to read one from a file
// or using CFGGroup API. In DCF CFG format it might look like:
//
// [ dataset ]
# sop class uid
0008,0016 = 1.2.840.10008.5.1.4.31
# sop instance uid
0008,0018 = 1.2.3.4.1.100
# Accession Number
0008,0050 = 111
# Referring Physician
0008,0090 = Dr. Nick
# referenced study sequence
# sop class uid
0008,1110.0008,1150 = 1.2.3.4.5
# sop instance uid
0008,1110.0008,1155 = 1.2.3.4.100
#patient name
0010,0010 = Simpson^Homer
#patient id
0010,0020 = 112233

CFGGroup g = CFGDB.instance().loadGroup( "dataset.cfg" );
DicomDataSet dds = new DicomDataSet( g );
DicomFileOutput dfo = new DicomFileOutput("myfile.dcm");
dfo.writeDataSet( dds );
dfo.close();
```

6.3.1.4. Example – Open, Modify, and Save a DICOM file

Open a DICOM file, modify the pixel data, and save to a new file.

```
using System;
using LaurelBridge.DCS;

namespace ex_ndcf_ModPixelData
{
/// <summary>
/// Summary description for Class1.
/// </summary>
class Class1
{
   /// <summary>
   /// The main entry point for the application.
   /// </summary>
   [STAThread]
   static void Main(string[] args)
   {
      DicomFileInput dfi = new DicomFileInput();
      dfi.open("C:/temp/in.dcm");
      DicomDataSet dds = dfi.readDataSet();
      dfi.close();
      // get some of the basic image header fields. Use the DicomDataSet
      // convenience methods to get integer values for these.
      int rows = dds.getElementIntValue( DCM.E_ROWS );
      int cols = dds.getElementIntValue( DCM.E_COLUMNS );
      int bits_allocated = dds.getElementIntValue( DCM.E_BITS_ALLOCATED );
      int bits_stored = dds.getElementIntValue( DCM.E_BITS_STORED );
      int pixel_count = rows*cols;
      // get the pixel data element.
      DicomElement e_pixel_data = dds.findElement( DCM.E_PIXEL_DATA );
      // get a C style pointer to the pixels - this locks the
      // element’s data buffer in memory,
      // until “releaseBinaryData” is called.
      // IntPtr’s can be used to create BitMap objects, or can be passed
      // to a COM C++ object.
      System.IntPtr p_raw_pixel_data = e_pixel_data.getBinaryData();

      // alternately, get the pixels as an array
      // Note that, DICOM defines two possible types for pixel data
      // OB (Other byte), and OW  (Other word). We cast our DicomElement
      // to the appropriate derived class.
      //
      if (e_pixel_data.VR == DCM.VR_OB)
      {
         byte[] pixel_data = ((DicomOBElement)e_pixel_data).getOBData();
         byte[] new_pixel_data = new byte[ pixel_count ];
         // error if pixel_count != pixel_data.Length;
         process8bitImageData( pixel_data, new_pixel_data,
                   bits_stored, rows, cols );
         dds.insert( new DicomOBElement(DCM.E_PIXEL_DATA,new_pixel_data ) );
      }
      else if (e_pixel_data.VR == DCM.VR_OW)
      {
         short[] pixel_data = ((DicomOWElement)e_pixel_data).getOWData();
         short[] new_pixel_data = new short[ pixel_count ];
         // error if pixel_count != pixel_data.Length;
         process16bitImageData( pixel_data, new_pixel_data,
                   bits_stored, rows, cols );
         dds.insert( new DicomOWElement( DCM.E_PIXEL_DATA,new_pixel_data) );
      }
      // else error

      // give the object a new UID. There are other attributes
      // that should also be set to
      // indicate that this is a "derived image". That is beyond the scope of
      // this example.
      dds.insert( DCM.E_SOPINSTANCE_UID, DicomDataDictionary.makeUID() );

      // save the data set with the new pixel data
      DicomFileOutput dfo = new DicomFileOutput();
      dfo.open( "C:/temp/out.dcm", UID.TRANSFERLITTLEENDIAN, false );
      dfo.writeDataSet( dds );
      dfo.close();

   }
}
}
```

6.3.2. Using VerificationClient

The most basic network service class in DICOM is the Verification service class. An SCU requests an association with an SCP, sends a C-Echo-Request DIMSE message, and waits to receive a C-Echo-Response DIMSE message.

In addition to the command line application (*dcf_echo_scu*), and the Java and C++ APIs, the `LaurelBridge.DCS.VerificationClient` class provides this functionality to C# developers. (See also the C# console app in *%DCF_ROOT%\devel\cssrc\examples\ex_necho_scu*.)

6.3.2.1. Example – Connect to a Verification SCP

Note the finally block at the end of this code block. If there is an error on connecting to an SCP or during the connection the SCU is not necessarily disconnected. The "finally" block in this example checks that the client is non-null and that the client is connected, then performs an association release (it could also abort if necessary).

```
int status;
VerificationSCU client = null;

try
{
   if( args.Length < 3 )
   {
      throw new DCFException(usage_);
   }

   Framework.FSysModeCFGDB = true;
   Framework.ConsoleModeLogger = true;
   Framework.initDefaultServices(args, CINFO.Instance );

   // enable selected debug/trace messages from the DCS library:
   DCS.CINFO.Instance.setDebugFlags(
               DCS.CINFO.df_DUMP_ACSE |
               DCS.CINFO.df_SHOW_DIMSE_READ |
               DCS.CINFO.df_SHOW_DIMSE_WRITE );

   client = new VerificationSCU("ECHO_SCU", args[0], args[1] + ":" + args[2]);
   client.requestAssociation();
   client.cEcho(10);
   System.Console.Error.WriteLine("test passed");
   status = 0;
}
catch (System.Exception e)
{
   LOG.error(- 1, "Exception caught:\n", e);
   System.Console.Error.WriteLine("test failed");
   status = 1;
}
finally
{
   try
   {
      if ((client != null) && client.Connected)
      {
         client.releaseAssociation();
      }
   }
   catch (DCSException e)
   {
      LOG.error(- 1, "Error releasing Association, aborting", e);
      client.abortAssociation();
      status = 1;
   }
}

Framework.shutdown(status);
System.Environment.Exit(status);
```

6.3.3. Using StoreClient

The `StoreClient` class provides a powerful and easy to use interface to DICOM archives or other storage providers. The user does not need to be concerned with any of the complexities of the DICOM storage service class protocol. Simply create a job describing the images (SOP instances) to be transferred and let the `StoreClient` do the work.

6.3.3.1. Example – Create and submit a store job from files on disk

Create and submit a store job where instance data comes from DICOM files on disk.

```
doStoreJob()
{
   // create a StoreJobDescription with one image

   StoreJobDescription sjd = new StoreJobDescription();
   sjd.ServerName = "localhost:2000:StoreSCP";
   sjd.ClientName = "StoreSCU";
   sjd.addInstance(new StoreJobInstanceInfo( "somefile.dcm", "", "", null ) );


   // create a StoreClient

   StoreClient client = new StoreClient();

// submit the job and then wait for completion - this blocks
// use "submitStoreJob" for non blocking behavior.

   StoreJobStatus sjs = client.runStoreJob( sjd );

   // print the status
   System.Console.WriteLine("status for store job:{0}", sjs );
}
```

6.3.3.2. Example – Create and submit a store job – Handling Events

Create and submit a store job and handle events when job status changes.

```
doStoreJob()
{
   // create a StoreJobDescription with one image

   StoreJobDescription sjd = new StoreJobDescription();
   sjd.ServerName = "localhost:2000:StoreSCP";
   sjd.ClientName = "StoreSCU";
   sjd.addInstance(new StoreJobInstanceInfo( "somefile.dcm", "", "", null ) );


   // create a StoreClient

   StoreClient client = new StoreClient();

// submit the job and then wait for completion - this blocks
// use "submitStoreJob" for non blocking behavior.

   StoreJobStatus sjs = client.runStoreJob( sjd );

}

       /*
       * Implementation of StoreClientListener interface.
       */
       public virtual void  storeObjectComplete(StoreJobInstanceStatus status)
       {

          LOG.info("Received storeObjectComplete event status = " +
status.dimseStatus());
       }

       /*
       * Implementation of StoreClientListener interface.
       */
       public virtual void  storeJobComplete(StoreJobStatus status)
       {
          LOG.info("Received storeJobComplete event " + System.Environment.NewLine +
"status = " + status.status() + System.Environment.NewLine + "statusInfo = " +
status.statusInfo() + System.Environment.NewLine + "status Info Ex = " +
status.statusInfoEx());
       }
```

6.3.4. Using Query/Retrieve classes

A common application of the DICOM protocol is querying an image archive for images or other composite objects. The simplest way to do this with the DCF is to use the `QRSCU` class. `QRSCU` provides a high level mechanism for interoperating with QRSCP's via the C-FIND, C-MOVE and C-GET DIMSE messages.

Here is a code fragment that creates a `QRSCU` object and a `QRIdentifier` (query attributes) then sends it as a C-FIND Request.

```
String host = hostname_box_.Text().Trim();
String port = port_box_.Text().Trim();

AssociationInfo ainfo = new AssociationInfo();
ainfo.calledTitle( called_ae_box_.getText().Trim() );
ainfo.callingTitle( calling_ae_box_.getText().Trim() );
ainfo.calledPresentationAddress( host + ":" + port );
```

```
RequestedPresentationContext ctx = new RequestedPresentationContext( (byte)1,
        UID.SOPPATIENTQUERY_FIND, new String[] {ts_uid_} );
ainfo.addRequestedPresentationContext( ctx );

CFGGroup session_cfg;
DicomSessionSettings session_settings = new DicomSessionSettings();
String session_cfg_name = "/dicom/QRSCU_default_session_settings.cfg";
try
{
    session_cfg = CFGDB.instance().loadGroup( session_cfg_name, true );
    session_settings = new DicomSessionSettings( session_cfg );
}
catch( CDSException e1 )
{
    LOG.error( -1, "Error loading session settings from CFGDB group name = "
            + session_cfg_name, e1 );
}

QRSCU scu_ = new QRSCU( ainfo, session_settings );

Configuration config = parent_panel_.getConfiguration();
scu_.maxReturnedResults( config.maxReturnedResults() );
scu_.queryTimeoutSeconds = config.queryTimeoutSeconds();
scu_.setRequestedSopClassUid = UID.SOPPATIENTQUERY_FIND;


QRIdentifier query = new QRIdentifier();
// Fields to query on:
query.patientsName =  patient_name_box_.Text.Trim();
query.queryretrieveLevel = (String)query_level_combo_box_.getSelectedItem();
query.studyInstanceUid = study_uid_box_.Text.Trim();

// Fields to return:
query.patientId = "";
query.patientsSex = "";
query.modality = "";
query.studyDate = "";
query.numberStudyRelInstances = "";
query.data_set.insert( DCM.E_INSTANCE_AVAILABILITY, "" );

addExtraDicomElementsToQuery( query, config.additionalDicomElements() );

DicomDataSet expected_fields = new DicomDataSet();
setTagsToDisplay( expected_fields, config.tagsToDisplay() );

parent_panel_.setFieldsExpected( expected_fields );

LOG.debug( CINFO.df_SHOW_GENERAL_FLOW, "Query Data set is " + query.data_set() );

query_status_.append( "Query data set:\n" );
query_status_.append( query.data_set().toString() );
query_status_.append( "\n" );

scu_.requestAssociation();
scu_.cFind( query.data_set(), QueryInfoPanel.this, false );
```

By implementing the `QueryListener` interface an object will notified when DIMSE Response messages are received, such as C-FIND-RSP or C-MOVE-RSP or C-GET-RSP.

The `queryEvent()` method is called for each intermediate response.

```
public void queryEvent( DimseMessage rsp )
{
    //output contents of message to log file
    LOG.info( "Received a DIMSE response message " + rsp );
}
```

The `queryComplete()` method is called when a final response is received or if an internal error occurs in the `QRSCU` class.

```
public void queryComplete( int status )
{
    try
    {
        //internal error send an abort message.
        if( status == QueryListenerStatus.QUERY_LISTENER_ERROR )
        {
            LOG.error( -1, "Aborting association\n" );
            scu_.abortAssociation();
        }
        else
        {
            LOG.info( "Releasing association\n" );
            scu_.releaseAssociation();
        }
        scu_ = null;
    }
    catch( DCSException e )
    {
        LOG.error( -1, "Error occurred while disconnecting association", e );
    }
}
```

6.3.4.1. Example – Using Query/Retrieve

See *%DCF_ROOT%\devel\cssrc\examples\ex_nqr_scu\ex_nqr_scu.cs*

and

%DCF_ROOT%\devel\cssrc\examples\ex_ndcf_nquery_scu\ex_ndcf_nquery_scu.cs.

6.3.5. Using PrintClient

The `PrintClient` class provides a powerful and easy to use interface to DICOM printers. The user need not be concerned with any of the complexities of the DICOM print service class protocol, but simply creates a job describing the films to be printed, and lets the `PrintClient` do the work.

To send images from a C# program to a DICOM Printer or Print "Service Class Provider" use the `PrintClient` class. `PrintClient` provides a very high level interface to a DICOM Print SCU. The application developer is removed from the process of negotiating an association, sending DIMSE messages, managing the complex relationships between objects in the normalized service classes, and handling printer and print job status notifications. The sheets of images that are to be printed are defined in an intuitive hierarchical structure. The `PrintClient` object handles the messy details of DICOM Print.

The `PrintJobDescription` object contains basic attributes of the job such as the server address and various job level options. Also included in the `PrintJobDescription` is a single `PrintJobFilmSession` object. This corresponds to the DICOM film-session object. `PrintJobFilmSession` contains one or more `PrintJobFilmBox` objects. A `PrintJobFilmBox` corresponds to the DICOM film-box object, which represents a sheet or film to be printed. `PrintJobFilmBox` contains one or more `PrintJobImageBox` objects. A `PrintJobImageBox` corresponds to a DICOM image-box and represents a single image to be placed somewhere on the film. When the job has completed, a `PrintJobStatus` object is returned which summarizes the results of the print operation.

The `PrintClient` also supports a listener or notification interface using the PrintClientListener interface. If the user installs a method to receive events, then notifications will be sent to that object as DICOM print-job or printer status values change.

```
//
// initialize the PrintJobDescription
//
PrintJobDescription job = new PrintJobDescription(); // describes the job we want to do
   PrintJobFilmSession film_session = new PrintJobFilmSession();
   PrintJobFilmBox film_box = new PrintJobFilmBox();
   PrintJobImageBox image_box = new PrintJobImageBox();

   job.serverAddress(print_server_address);
   job.clientAddress("DEMO");
   job.requestPrintJobSOPClass(true);
   job.pollPrintJob(true);
   job.printJobPollRateSeconds(2);
   job.jobTimeoutSeconds(30);

   film_session.numberOfCopies = ("1");
   film_session.printPriority = ("HIGH");
   film_session.mediumType = ("BLUE FILM");
   film_session.filmDestination = ("MAGAZINE");
   film_session.filmSessionLabel = ("test");
   film_session.memoryAllocation = ("0");
   film_session.ownerId = ("DCF");

   film_box.imageDisplayFormat = ("STANDARD\\1,1");
   film_box.filmOrientation = ("PORTRAIT");
   film_box.filmSizeId = ("14INX17IN");
   film_box.magnificationType = ("NONE");
   film_box.smoothingType = ("NONE");
   film_box.borderDensity = ("0");
   film_box.emptyImageDensity = ("0");
   film_box.minDensity = (0);
   film_box.maxDensity = (280);
   film_box.trim = ("YES");
   film_box.configurationInformation = ("NONE");
   film_box.illumination = (0);
   film_box.reflectedAmbientLight = (0);
   film_box.requestedResolutionId = ("HIGH");

   image_box.imagePosition = (1);
   image_box.polarity = ("NORMAL");
   image_box.magnificationType = ("NONE");
   image_box.smoothingType = ("NONE");
   image_box.configurationInformation = ("NONE");
   image_box.requestedImageSize = ("0");
   image_box.reqdDecimatecropBehavior = ("DECIMATE");
   image_box.imageInstanceInfo(dii);
```

```
    film_box.addImageBox(image_box);
    film_session.addFilmBox(film_box);
    job.filmSession(film_session);

    PrintClient print_client = new PrintClient();

    LOG.info("submitting print job:" + job.ToString());

    PrintJobStatus job_status = new PrintJobStatus(job.jobUID());
    print_client.submitPrintJob(job, null, job_status);
    PrintJobStatus job_status = print_client.runPrintJob( job );
```

This code was taken from the C# `ex_nprint_client` example. The full source for the program can be found in *`$DCF_ROOT\devel\cssrc\examples\ex_nprint_client`*.

6.3.5.1. Example – Create and submit a print job, handling status events

Create and submit a print job where ImageBox data comes from DICOM disk files and handle events when Printer or PrintJob status changes. See $DCF_ROOT/devel/cssrc/examples/ex_nprint_scu for a source code example.

6.3.5.2. Example – Create and submit a print job where ImageBox data comes from DICOM disk files.

See *`$DCF_ROOT/devel/cssrc/examples/ex_nprint_client/ex_nprint_client.cs`* for a complete program.

```
    public virtual int runPrint(System.String[] args)
        {
            DicomInstanceInfo dii = new DicomInstanceInfo( args[0] );

            // Print server
            System.String print_server_address = args[1];

            PrintJobDescription job = new PrintJobDescription(); // describes the job we
want to do
            PrintJobFilmSession film_session = new PrintJobFilmSession();
            PrintJobFilmBox film_box = new PrintJobFilmBox();
            PrintJobImageBox image_box = new PrintJobImageBox();

            job.serverAddress(print_server_address);
            job.clientAddress("DEMO");
            job.requestPrintJobSOPClass(true);
            job.pollPrintJob(true);
            job.printJobPollRateSeconds(2);
            job.jobTimeoutSeconds(30);

            film_session.numberOfCopies = ("1");
            film_session.printPriority = ("HIGH");
            film_session.mediumType = ("BLUE FILM");
            film_session.filmDestination = ("MAGAZINE");
            film_session.filmSessionLabel = ("test");
            film_session.memoryAllocation = ("0");
            film_session.ownerId = ("DCF");

            film_box.imageDisplayFormat = ("STANDARD\\1,1");
            film_box.filmOrientation = ("PORTRAIT");
            film_box.filmSizeId = ("14INX17IN");
            film_box.magnificationType = ("NONE");
            film_box.smoothingType = ("NONE");
            film_box.borderDensity = ("0");
            film_box.emptyImageDensity = ("0");
            film_box.minDensity = (0);
```

```
          film_box.maxDensity = (280);
          film_box.trim = ("YES");
          film_box.configurationInformation = ("NONE");
          film_box.illumination = (0);
          film_box.reflectedAmbientLight = (0);
          film_box.requestedResolutionId = ("HIGH");

          image_box.imagePosition = (1);
          image_box.polarity = ("NORMAL");
          image_box.magnificationType = ("NONE");
          image_box.smoothingType = ("NONE");
          image_box.configurationInformation = ("NONE");
          image_box.requestedImageSize = ("0");
          image_box.reqdDecimatecropBehavior = ("DECIMATE");
          image_box.imageInstanceInfo(dii);

          film_box.addImageBox(image_box);
          film_session.addFilmBox(film_box);
          job.filmSession(film_session);

          PrintClient print_client = new PrintClient();

          LOG.info("submitting print job:" + job.ToString());

          PrintJobStatus job_status = new PrintJobStatus(job.jobUID());
          print_client.submitPrintJob(job, null, job_status);

          LOG.info("Done! Print job status: " + job_status);

          if ( job_status.status().Equals("SUCCESS") )
          {
             return 0;
          }
          else
          {
             return 1;
          }
       }
```

6.3.6. Creating and Populating DICOM Sequences

A sequence is just another type of DICOM element (DicomSQElement). Rather than having an array of Strings, shorts, bytes, or such as its data, DicomSQElement has an array of DicomDataSet's as its data.

6.3.6.1. Example – Explicitly creating DICOM Sequence elements

To create and populate a DICOM dataset that contains a sequence, you typically do the following:

1. Create a single data set, or an array of data sets,
2. Populate those with elements,
3. Create the sequence element with the dataset(s)., and usually
4. Add the sequence element to the top-level data set.

Here's an example of how you might accomplish these steps:

```
        DicomDataSet ds = new DicomDataSet();  //// top level DS

//(1)
        DicomDataSet seq_ds_1 = new DicomDataSet();

//(2)
       seq_ds_1.insert(new DicomUIElement(DCM.E_REFERENCED_SOPCLASS_UID,
"1.2.840.10008.5.1.4.1.1.14"));
```

```
        seq_ds_1.insert(new DicomUIElement(DCM.E_REFERENCED_SOPINSTANCE_UID,
"1.2.840.114089.1.2.3.4"));

//(3)
        DicomSQElement sq_e =  new DicomSQElement(DCM.E_REFERENCED_IMAGE_SEQUENCE,
seq_ds_1);

//(4)
        ds.insert( sq_e );
```

There are no doubt variations on how a given application will do things – refer to the online docs for DicomSQElement, DicomDataSet, etc., for additional details and options.

6.3.6.2. Example – Setting DICOM Sequence elements using a config group

If you have a `CFGGroup` similar to what `dcf_pg` uses, you can give that config group to a `DicomDataSet` constructor, and create a dataset based on the specification in the config group.

For example, if you have a config group stored in a file and load that group as:

```
    CFGGroup g = CFGDB.loadGroup("file:/data_set.cfg");
```

And the file `data_set.cfg` contains:

```
[ header_info ]
# sop class uid
0008,0016 = 1.2.840.10008.5.1.4.31
# sop instance uid
0008,0018 = 1.2.3.4.1.100
# Accession Number
0008,0050 = 111
# Referring Physician
0008,0090 = Dr. Nick
# referenced study sequence
# sop class uid
0008,1110.0008,1150 = 1.2.3.4.5
# sop instance uid
0008,1110.0008,1155 = 1.2.3.4.100
```

then you could say:

```
    DicomDataSet ds = new DicomDataSet( g );
```

to initialize the dataset with the values contained in the config group.

A sequence may be entered in a config group file as a tag by appending it to a numeric tag (the traditional group-element pair) with a period ("`.`"). You may also indicate an item in the sequence with "`#`" and the sequence item ID, followed by the tag indicating the sequence. There may be multiple sequences and sequence IDs as part of one "tag". Examples are shown below:

- Simple tag – `0010,0010`
- Tag with sequence – `0080,0100.0008,0060`
- Tag with sequence ID and sequence – `0080,0100.#0.0008,0060`
- Tag with multiple sequences and IDs – `0080,0100.#1.0080,0100.#0.0008,0060`

If no item number is specified, the first item (#0) is assumed. You can also specify the last element in a sequence by "#L" (upper-case is important!) if you don't know how many items are in a sequence. If you are creating new elements, you can specify the next item in the sequence via "#N" (again, case is important) to append to the sequence. For example: `0080,0100.#L.0010,0010.#N.0008,0060`

Please notice that:

- The sequence IDs (e.g., #1) and the tag-value pairs for the sequences are all separated by periods (“.”).
- The tags for the sequences are simple group-element pairs themselves.

6.3.7. Handling alternate character sets with DCF (C#)

The DCF provides limited support for string elements that are encoded in non-default character sets (i.e., a dataset that contains element 0008,0005 Specific Character Set where that element’s value is set).

Essentially, when reading any string data from DICOM file or DIMSE message, DCF will explicitly specify that the raw bytes should be converted to a string using ISO8859-1 decoding. Since ISO8859-1 is an 8 bit character set, you can get the bytes back from the element and create a new string that decodes the bytes using a different character set. Likewise, when the DCF retrieves the bytes from a string element prior to writing a file or DIMSE message, it will specify that ISO8859-1 encoding should be used. For example:

```
DicomStringElement e = dds.getElement( DCM.E_PATIENTS_NAME );
String s = e.getEncodedStringValue();
byte[] raw_data = System.Text.Encoding.GetEncoding( "iso-8859-1").GetBytes( s );
String s2 = System.Text.Encoding.GetEncoding(
   some_other_charset_name ).GetString(raw_data);
```

The method getEncodedStringValue concatenates all of the values using "\" as the delimiter, and adds any needed pad chars (space or null for UIDs). The resultant value should end up looking the same as the original raw data.

Notice that the number of values (DicomStringElement.vm() or DicomStringElement.values().length) when we force ISO8859-1 decoding may be different from the number of values when using some other character set. That is, a “\” byte value in a string in an alternate char set may actually be something other than a DICOM string VM delimiter.

Also note that if you create DicomStringElements containing text in an alternate character set, you may need to re-encode it as ISO8859-1 before passing it to the constructor: For example:

```
String s = some_string_in_an_altnate_char_set;
byte[] raw_data = System.Text.Encoding.GetEncoding( some_other_char_set ).GetBytes( s );
String s2 = System.Text.Encoding.GetEncoding( "iso-8859-1").GetString(raw_data)
DicomPNElement e = new DicomPNElement( DCM.E_PATIENTS_NAME, s2 );
   // or
DicomElement e = DicomElementFactory.create( DCM.E_PATIENTS_NAME, s2 );
```

Note that Chapter 3, Section C.12.1.1.2 Specific Character Set in the DICOM standard outlines all the various defined terms for DICOM element 0008,0005 Specific Character Set. Another list is provided in Chapter 18, Annex D - IANA Mapping (informative).

The Windows character set names, for example, don’t match up to the values DICOM requires for the 0008,0005 tag. The OEM developer will need to write a mapping method of some kind to translate back and forth between the various naming conventions. For example, if the encoding name for C#.Net is “ISO-8859-2” (ISO Latin 2), then, by these tables, the DICOM value to put in the 0008,0005 attribute would be “ISO_IR 101”.

6.4. Deploying a Simple C# Standalone Application

6.4.1. Deploying a Simple C# Standalone DCF Application

The following procedure shows a simple method of deploying *a DCF C# application* to a Windows host. The application (.exe), its required libraries (.dll), and configuration data can be installed into a single directory on the target system. The application can then be run from the installation directory.

Note that the .Net runtime must be installed on the target system to allow C# applications to run.

We'll show the process of creating the install directory on your DCF developer box (the host with the DCF toolkit installed). Once created, that install directory can then be copied to the target using any number of methods: zip on your DCF developer box, and unzip on the target; or perhaps burn this directory to a CD-ROM and then run directly from the CD on the target.

This example shows deploying the C#.Net `ex_ndcf_filter` example and `ex_ndcf_dump`. The process would be modified somewhat for your own application.

Perform the following steps:

1. Open a DCF command window:
   Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Command Prompt"
2. Create the test install directory:
   Note: You could paste this text into a batch file and run it to automate this process.

```
   REM###
   REM### create install dir
   REM###
   mkdir DCF_test_cs_install
   cd DCF_test_cs_install

   REM###
   REM### copy required library files from %DCF_LIB% (../DCF/lib)
   REM###
   copy %DCF_LIB%\DCF_DCFCore.dll
   copy %DCF_LIB%\DCF_ljpeg12.dll
   copy %DCF_LIB%\DCF_ljpeg16.dll
   copy %DCF_LIB%\DCF_ljpeg8.dll
   copy %DCF_LIB%\LaurelBridge.APC_a.dll
   copy %DCF_LIB%\LaurelBridge.CDS_a.dll
   copy %DCF_LIB%\LaurelBridge.DCF.dll
   copy %DCF_LIB%\LaurelBridge.DCS.dll
   copy %DCF_LIB%\LaurelBridge.DDS.dll
   copy %DCF_LIB%\LaurelBridge.DDS_a.dll
   copy %DCF_LIB%\LaurelBridge.LOG_a.dll
   copy %DCF_LIB%\LaurelBridge.NDCDS.dll
...copy %DCF_LIB%\DCF_TSCW.dll
   copy %DCF_LIB%\DCF_TSCWIJG.dll
   copy %DCF_LIB%\DCF_TSCWJasper.dll
   REM ### The Aware wrapper dll is needed only if using Aware's JPEG libraries.
   REM ### Note the actual Aware JPEG library (awj2k.dll) must be purchased separately
   copy %DCF_LIB%\DCF_TSCWAware.dll

   REM### copy required library files from %DCF_BIN% (../DCF/bin).
   REM### These may exist in other places on the system, but copies
   REM### are put here during DCF toolkit install for convenience,
   REM### (Note omniORB dlls may not be required depending on the
   REM### application and your DCF version)

    REM ### If you are building from a DCF VisualStudio8.x .NET toolkit:
    copy %DCF_BIN%\msvcp80.dll
    copy %DCF_BIN%\msvcr80.dll
```

```
 REM ### Note that the filenames may differ somewhat from what is specified here.
copy %DCF_BIN%\omniORB414_rt.dll
copy %DCF_BIN%\omnithread34_rt.dll

REM###
REM### Copy the application that you want - for example,
REM### include both the C#.Net ex_ndcf_filter example, and the ex_ndcf_dump
REM### example.
REM###
copy %DCF_BIN%\ex_ndcf_dump.exe
copy %DCF_BIN%\ex_ndcf_filter.exe

REM###
REM### Copy the sample filter configuration for ex_ndcf_filter
REM### if desired.
REM###
copy %DCF_ROOT%\devel\cssrc\examples\ex_ndcf_filter\example_filter.cfg

REM###
REM### Create a minimal configuration directory.
REM###
mkdir cfg
mkdir cfg\apps
mkdir cfg\apps\defaults
mkdir cfg\procs

REM###
REM### Copy the license configuration file, and the application configs
REM### for the installed programs.
REM###
copy %DCF_CFG%\systeminfo cfg\systeminfo
copy %DCF_CFG%\apps\defaults\ex_ndcf_filter cfg\apps\defaults
copy %DCF_CFG%\apps\defaults\ex_ndcf_dump cfg\apps\defaults
```

3. Create the media by which you will deliver the install directory to the target machine.

4. On the target machine do the following:
   a) Install .Net framework redistributable (i.e., .NET dlls, etc.)
      Note: this is necessary to run any .Net (C#) application.
   b) Unpack, copy or otherwise make the DCF app install directory available. For example, copy or unzip to `C:\temp\DCF`
   c) From a command window, go to the install directory. For example, use `C:\temp\DCF`.
      `cd C:\temp\DCF`
   d) Set the environment vars and run your apps (you could put this text in a *`run_app.bat`* file).

```
set DCF_CFG=C:\temp\DCF\cfg
set DCF_LIB=C:\temp\DCF
set DCF_TMP=C:\temp\DCF
### display input image (choose a dicom file here)
ex_ndcf_dump \temp\test.dcm

###
### filter that image - the example_filter.cfg
### uses "C:\temp\test.dcm" and "C:\temp\filtered_image.dcm"
### as input and output respectively.
ex_ndcf_filter -f file:/example_filter.cfg

### examine output
ex_ndcf_dump \temp\filtered_image.dcm
```

6.4.2. Deploying a Simple C# Standalone OEM Application

The following procedure shows a simple method of deploying an *OEM C# application that uses some DCF classes* to a Windows host. The application (.exe), its required libraries (.dll), and configuration data can be installed into a single directory on the target system. The application can then be run from the installation directory.

Note that the .Net runtime must be installed on the target system to allow C# applications to run.

We'll show the process of creating the install directory on your DCF developer box (the host with the DCF toolkit installed). Once created, that install directory can then be copied to the target using any number of methods: zip on your DCF developer box, and unzip on the target; or perhaps burn this directory to a CD-ROM and then run directly from the CD on the target.

For this example a new app in cssrc/examples/ex_ndcf_simple_win_app was created. This is a non-DCF app that was created using the Visual Studio designer and the process from this guide (see Section 6.2.4 – Manually Creating C# Projects from MS Visual Studio 2003/2005 IDE). This example is basically a GUI version of dcf_dump. This directory includes a `cinfo.cfg` only so that it can be auto-built, but it has "`gen_build_file = no`" and "`gen_cinfo_code_and_data = no`". The user could delete the `cinfo.cfg` file, and still use the example in Visual Studio.

The deployment is nearly the same as for the example above for the DCF C# app, except that there is no app config file, so instead, you need to include the `cfg\components\cs_lib` files so DCF libraries can load default settings if needed.

To create this application, perform the following steps:

1. Open a DCF command window:
   `Select "Start" → "All Programs" → "DICOM Connectivity Framework" → "DCF Command Prompt"`
2. Create the test install directory:
   Note: You could paste this text into a batch file and run it to automate this process.

```
REM ###
REM ### create install dir
REM ###
mkdir DCF_test_cs_install
cd DCF_test_cs_install

REM ###
REM ### copy required library files from %DCF_LIB% (../DCF/lib)
REM ###
copy %DCF_LIB%\DCF_DCFCore.dll
copy %DCF_LIB%\DCF_ljpeg12.dll
copy %DCF_LIB%\DCF_ljpeg16.dll
copy %DCF_LIB%\DCF_ljpeg8.dll
copy %DCF_LIB%\LaurelBridge.APC_a.dll
copy %DCF_LIB%\LaurelBridge.CDS_a.dll
copy %DCF_LIB%\LaurelBridge.DCF.dll
copy %DCF_LIB%\LaurelBridge.DCS.dll
copy %DCF_LIB%\LaurelBridge.DDS.dll
copy %DCF_LIB%\LaurelBridge.DDS_a.dll
copy %DCF_LIB%\LaurelBridge.LOG_a.dll
copy %DCF_LIB%\LaurelBridge.NDCDS.dll
copy %DCF_LIB%\DCF_TSCW.dll
copy %DCF_LIB%\DCF_TSCWIJG.dll
copy %DCF_LIB%\DCF_TSCWJasper.dll
REM ### The Aware dll is needed only if you are using Aware's JPEG libraries.
copy %DCF_LIB%\DCF_TSCWAware.dll


REM ### copy required library files from %DCF_BIN% (../DCF/bin).
REM ### These may exist in other places on the system, but copies
REM ### are put here during DCF toolkit install for convenience,
REM ### (Note omniORB dlls may not be required depending on the
REM ### application and your DCF version)

REM ### If you are building from a DCF VisualStudio8.x .NET toolkit:
copy %DCF_BIN%\msvcp80.dll
copy %DCF_BIN%\msvcr80.dll

REM ### Note that the filenames may differ somewhat from what is specified here.
copy %DCF_BIN%\omniORB414_rt.dll
copy %DCF_BIN%\omnithread34_rt.dll

REM ###
REM ### Copy the application that you want.
REM ###
REM ### In your C# project directory, the executable might
REM ### be in .\bin\Debug or .\bin\Release
REM ### Here, we assume the exe is in \temp.
copy C:\temp\ex_ndcf_simple_win_app.exe .

REM ###
REM ### Create a minimal configuration directory.
REM ###
mkdir cfg
mkdir cfg\apps
mkdir cfg\apps\defaults
mkdir cfg\procs
mkdir cfg\components
mkdir cfg\components\cs_lib

REM ###
REM ### Copy the license configuration file, and the
REM ### C# library component configuration data. Since
REM ### there is no application configuration, library
REM ### code may look to this data for default configuration.
REM ###
copy %DCF_CFG%\systeminfo cfg\systeminfo
copy %DCF_CFG%\components\cs_lib\* cfg\components\cs_lib
```

3. Create the media by which you will deliver the install directory to the target machine.

4. On the target machine do the following:
   a) Install .Net framework redistributable (i.e., .NET dlls, etc.)
      Note: this is necessary to run any .Net (C#) application.
   b) Unpack, copy or otherwise make the DCF app install directory available. For example, copy or unzip to `C:\temp\DCF`
   c) From a command window, go to the install directory. For example, use `C:\temp\DCF`.
      `cd C:\temp\DCF`
   d) Set the environment vars and run your apps (you could put this text in a *`run_app.bat`* file).

```
set DCF_CFG=C:\temp\DCF\cfg
set DCF_LIB=C:\temp\DCF
set DCF_TMP=C:\temp\DCF

### Run your application
ex_ndcf_simple_win_app
```

Note: If you are separating the installed files into "bin" and "lib" directories – for example, your executable is in the "install/bin" directory and any libraries it needs are in the "install/lib" directory – you need to take some additional steps when deploying your application.

For one thing, you will need to make sure the bin and lib directories are in the PATH environment variable. You will also probably need to add each of the LaurelBridge C# DLLs to the Global Assembly Cache (GAC) with the gacutil command, e.g., `gacutil /i LaurelBridge.DCF.dll`.

6.5. Common Services Programming Examples

6.5.1. C# "hello world" Example Application Using the DCF

To demonstrate some of the capabilities of the DCF, you can create and run the most basic of code examples: the "Hello World" program. The DCF "Hello World" program demo will make use of the DCF development tools, as well as the common services APIs and implementations.

Change to the *devel/cssrc/examples/ex_ndcf_HelloWorld* directory under the DCF install directory, then build and execute the example application:

```
cd %DCF_USER_ROOT%/devel/cssrc/examples/ex_ndcf_HelloWorld
perl -S dcfmake.pl
ex_ndcf_HelloWorld
```

From your web browser, select "View Log Files" from the DCF Remote Service Interface. Select the log file for the ex_ndcf_HelloWorld application, and view the output.

To create the ex_ndcf_HelloWorld application, the following steps were followed:

1. Create a directory for the application
2. Create a component information file for the application
3. Create the source code for the application
4. Build the application
5. Update the configuration data base

1. Create a directory for the new application component. In the DCF, every application or library is a component and has its own source directory.
   ```
   mkdir %DCF_ROOT%\devel\cssrc\examples\ex_ndcf_HelloWorld
   ```
2. Create a component information file in that directory. This file must be called "*cinfo.cfg*". For this example it contains the following:
   ```
   #==========================================================================
   # static information common to all instances of the ex_ndcf_HelloWorld component
   #==========================================================================
   [ component_info ]
   name = ex_ndcf_HelloWorld
   namespace_prefix = LaurelBridge
   guid = 6BB35642-8400-44FA-850E-E7EAA1C03B21
   type = cs_app
   category = examples
   docfile = none
   description = C# example app logs.

   [ build_info ]
   platform = Windows_NT_5_x86_VisualStudio10.x
   platform = Windows_NT_5_x64_VisualStudio10.x
   platform = Windows_NT_5_x86_VisualStudio12.x
   platform = Windows_NT_5_x64_VisualStudio12.x
   ```

```
gen_build_file = yes
gen_cinfo_code_and_data = yes

[ debug_controls ]
debug_flag = df_EXAMPLE,     0x10000, do something special if this debug flag is set

[ required_components ]
component = cs_lib/DCF
component = cs_lib/LOG_a
component = cs_lib/APC_a

#===============================================================================
# per-instance information for the ex_ndcf_HelloWorld component
#===============================================================================
[ ex_ndcf_HelloWorld ]
debug_flags = 0x00000

[ ex_ndcf_HelloWorld/english ]
hello_world = hello world

[ ex_ndcf_HelloWorld/spanish ]
hello_world = hola mundo

[ ex_ndcf_HelloWorld/french ]
hello_world = bonjour le monde

[ ex_ndcf_HelloWorld/german ]
hello_world = hallo welt
```

The file is in the DCF configuration file format, which provides for attributes, groups, and nested groups.

Note: The easiest way to create the cinfo.cfg file for your application or library is to copy one from a similar component, then edit as needed. You will need to generate a new GUID using a program like "uuidgen /c"

The first group `[ component_info ]` describes basic attributes of the component. The component `type` is "cs_app" which indicates a C# application. The `guid` (Globally Unique Identifier) is used by Visual Studio. Each new component you create must have a new `guid`. You can use the program "*`uuidgen`*" with the argument "*`/c`*" to generate a new `guid` in the correct format.

You can use *`dcfmake.pl`* to create applications in any directory, as long as you create a *`cinfo.cfg`* file in that directory. You can also use DCF C# classes for your application as you would any other C# class library.

The `[ required_components ]` group specifies three components needed by this application. The group `[ debug_controls ]` is where the developer can add support for conditional logging or other behavior specific to this component. Debug controls that are defined here can be accessed via the web interface.

The `[ ex_ndcf_HelloWorld ]` group contains the instance configuration for the component. This data is used directly in the example code.

3. Create the application source code

For this example, the file is called "*`ex_ndcf_HelloWorld.cs`*".

```
using System;
using LaurelBridge.DCF;

```

```
namespace LaurelBridge
{
   namespace ex_ndcf_HelloWorld
   {

      /// <summary> The class ex_ndcf_HelloWorld demonstrates the most basic of programs
      /// using the DCF common services interfaces for Application Control (APC),
      /// Configuration Data Services (CDS), and Logging (LOG)
      ///
      /// To make it interesting, messages from different languages
      /// are retrieved from the application configuration.
      /// </summary>
      public class ex_ndcf_HelloWorld
      {

         private static System.String usage_ =
            "use ex_ndcf_HelloWorld [adapter options] [ex_ndcf_HelloWorld options]\n"
            + "adapter options are passed to LOG_a, CDS_a, and APC_a setup methods\n"
            + "[ ex_ndcf_HelloWorld options ]\n" + "-help display this message\n"
            + "-lang [english|spanish|french|german] specifies which message to
display\n";

         [STAThread]
         public static void  Main(System.String[] args)
         {
            int status;
            try
            {
               System.String language = "english";

               for (int i = 0; i < args.Length; i++)
               {
                  if (args[i].Equals("-help"))
                  {
                     System.Console.Error.WriteLine(usage_);
                     System.Environment.Exit(0);
                  }
                  else if (args[i].Equals("-lang"))
                  {
                     if ((i + 1) >= args.Length)
                     {
                        System.Console.Error.WriteLine(usage_);
                        System.Environment.Exit(1);
                     }
                     language = args[i + 1];
                  }
               }


               // Setup the DCF common services adapters

               // we'll use the fully configured logger,
               // but only the File-system mode CFGDB - i.e.,
               // DCDS_Server is not required to be running

               Framework.ConsoleModeLogger = false;
               Framework.FSysModeCFGDB = true;
               Framework.initDefaultServices( args, CINFO.Instance );

               // Get the ex_ndcf_HelloWorld component configuration from within the
               // ex_ndcf_HelloWorld application instance configuration
               //CFGGroup component_cfg = CINFO.Config;
               CFGGroup component_cfg = AppControl.Instance.getProcCfgGroup(
"cs_app/ex_ndcf_HelloWorld" );

               // get the configuration group containing various message strings
               CFGGroup messages = component_cfg.getGroup(language);
```

```
                // write the appropriate hello world message to the logger.
                LOG.info(messages.getAttributeValue("hello_world"));

                // write a debug message
                LOG.debug(CINFO.df_EXAMPLE, "this only prints if the df_EXAMPLE debug
flag is set");

                // print something to stderr
                System.Console.Error.WriteLine("test completed successfully - see log
files for output");

                status = 0;
            }
            catch (Exception e)
            {
                LOG.error(-1, "Exception caught:\n", e );
                status = -1;
            }

            Framework.shutdown( status );

        }
    }
  }
}
```

4. Build the application.

   To build the application, simply type the command

   perl –S dcfmake.pl

   Invoking *dcfmake.pl* will perform the following steps for this example:

   a) Read the *cinfo.cfg* file in the current working directory.
   b) Read the component configuration for each "`required component`" in the *cinfo.cfg*. Component configurations come from the *%DCF_USER_ROOT\devel\cfggen\components* directory. That data was created when *dcfmake.pl* built those components.
   c) Recursively read component configurations for other required components.
   d) Generate the component configuration for this component. This data is written to the file *%DCF_USER_ROOT%\devel\cfggen\components\cs_app\ex_ndcf_HelloWorld*
   e) Generate the application configuration for this component. This data is written to the file *%DCF_USER_ROOT%\devel\cfggen\apps\defaults\ex_ndcf_HelloWorld*
   f) Generate the *CINFO.cs* source file in the current directory. The `CINFO` class contains the debug flag mask constants as well as code to initialize and update the debug flags value from the CDS database. `CINFO` also provides convenience mechanisms for getting the configuration group for the component within a particular application.
   g) Generate the *LOG.cs* source file in the current directory. The `LOG` class (which is internal to the component's assembly) is simply a wrapper for the DCF `LOG` interface. It simplifies checking debug flag settings in `CINFO`, and provides message header fields that remain constant for the component.
   h) Generate the .csproj.
   i) Invoke "*devenv .csproj <args>*". Any arguments given to *dcfmake.pl* are forwarded to *devenv*. After the make completes, the generated *.csproj file* is removed. You can have *dcfmake.pl* leave the generated file by using the "`-keep`" option.

5. Update the configuration data service repository.

   This developer can determine when to deploy any newly created or edited configuration data. This can be useful if you are testing with non-default configurations and do not want the fact that you have rebuilt something to affect your working configuration files. To update the data execute the command:

   ```
   perl –S update_cds.pl
   ```

   This will copy all files from the temporary areas *%DCF_USER_ROOT%\devel\cfggen* and *%DCF_USER_ROOT%\devel\cfgsrc* to the working area: *%DCF_USER_ROOT%\cfg*. As the files are copied various macros are expanded, e.g., the files in the working configuration can have the correct port numbers, path names, etc.

The application is now ready to run!

6.5.2. Using the LOG interface – Logging from C# programs

Each C# component assembly has an internal class named "`LOG`". This class is generated by *dcfmake.pl* in the file *LOG.cs*. Component-specific debug flags are generated in the assembly-internal file *CINFO.cs*.

First, the `LOG` adapter must be initialized. Normally, all of the common services are installed at once, during application initialization. This can be done with the lines:

```
LaurelBridge.LOG_a.LOGClient_a.setup( args );
LaurelBridge.CDS_a.CFGDB_a.setup( args );
LaurelBridge.APC_a.AppControl_a.setup( args, CINFO.instance );
```

Messages are logged using the following methods:

```
LOG.info( "this message will always print" );
LOG.error( -1, "this is an error" );
LOG.error( -1, "the stack trace contained in the exception (e) will print\n", e );
LOG.debug(  CINFO.df_SHOW_GENERAL_FLOW, "this is a conditional debug message" );
```

Generally it is best to keep possibly expensive expressions like

```
("Here is a data set: " + ds.toString() )
```

in conditionals, since in C# (and Java), the args to LOG.debug are evaluated before calling the method, which then may decide to not log anything.

A better approach is to do something like what is illustrated in the following example:

```
if ( CINFO.debug( CINFO.df_SHOW_GENERAL_FLOW )
{
    LOG.debug( "Namespace.StoreSCP.DicomDataService_a.storeObject: dimse-message = " +
c_store_rq);
}
```

By wrapping the debug message in a conditional at least you're not doing extra work when you are in non-debug mode.

6.5.3. Avoiding or Embracing use of the Common Services

The ability to run without any configuration files and without initializing any of the common services is sometimes a desirable option. There are a variety of approaches and possibilities available:

1. Using no common services and no config files:

Under this scenario any services that are used by you or DCF are auto-initialized to a reasonable default configuration.

```
   // just start using DCF classes: e.g.
   using namespace Laurelbridge.DCS;
   DicomFileInput dfi = new DicomFileInput("file.dcm");
   DicomDataSet dds = dfi.readDataSet();
```

In this mode, default configuration data if needed for DCS classes comes from compiled-in data in `LaurelBridge.DCS.CINFO`.

Note: If you do not initialize the logger component to write to a file; then by default you will get the Console Mode logger facility. This may produce unexpected messages or behavior depending on whether or not your application has a console. Even if your application does not write LOG messages, DCF library code may write some info messages to the logger, which will attempt to write to the console. In addition, if you ever need to turn on DICOM debug logging (or any logging, for that matter), this could create a problem for a GUI or Service application that does not have a console.

Note: A word of caution when no configuration files are used. Because configuration data is not getting read from a file, but from burned in default data contained in the DCF assemblies, this means that if you want to change a default value then your code must have a facility to allow users to change those values programmatically. Example of things you might want to change include: Debug flags, timeout values, AE titles, and the like.

2. Using no common services, no application config file, but including component configurations:

   Under this scenario component configurations are created and stored in the directory: "`%DCF_CFG%\components\cs_lib\*`".

   In this mode, default configuration data, if needed for DCS classes, comes from the file: `%DCF_CFG%\components\cs_lib\DCS`. Other DCF libraries would find their config data in the same fashion.

   See Note in example 1 above.

3. Use all the Common Services:

   In this scenario your system is considered fully configured, i.e.,

   a) You have an application configuration, e.g., `%DCF_CFG%\apps\defaults\some_app_name`

You're setting up common services.
You're a server that is started by the system manager and will communicate with it as it initializes (See configurations in `%DCF_CFG%\systems.`)

```
// init common services
LaurelBridge.LOG_a.LOGClient_a.setup( args );
LaurelBridge.CDS_a.CFGDB_a.setup( args );
LaurelBridge.APC_a.AppControl_a.setup( args, CINFO.Instance );

// let sysmgr know you're done init (If you're an SCP using AssociationManager,
// it will do this when run() is called
AppControl.Instance.applicationReady();
```

Note: In this mode (3), we're spelling out what is happening and not using "`Framework.initDefaultServices()`". While this command still works, we're discouraging its use. For the two lines that it saves, there are potential problems or confusing issues. Since the class "`Framework`" is in namespace "`DCF`", which builds before `CDS_a`, `LOG_a`, `APC_a`, it can't literally call those three "`setup`" methods as above. Instead, it uses reflection to load the forward referenced assemblies and call them. Likewise, the methods/properties on `Framework` to set various options are just pass-throughs to the real classes. It seems cleaner to just let the user understand more of what's actually happening as illustrated above.

4. Something in between choosing no services or all services.

   The problem is there are MANY variations possible, so here's one example:

   a) We have an app-config, so we'll setup `AppControl`.

We want to ignore the logger config data in the app-config, and just set a single file name.
We want to use the file system (`fsys`) mode cfg db.
We are not interacting with the system manager.

```
// set up logger with a single file
LaurelBridge.LOG_a.LOGClient_a.LogFileName = "test.log";
LaurelBridge.LOG_a.LOGClient_a.setup( args );

// setup CFGDB adapter in fsys mode.
// Note: this is what happens if you do nothing, and someone calls CFGDB.Instance
LaurelBridge.CDS_a.CFGDB_a.FSysMode = true;
LaurelBridge.CDS_a.CFGDB_a.setup( args );

// setup AppControl and tell it we will not be receiving shutdown messages, which
// implies we don't talk to system manager
LaurelBridge.APC_a.AppControl_a.setHandleExternalShutdownRq( false );
LaurelBridge.APC_a.AppControl_a.setup( args, CINFO.Instance );
```

See Note in example 1 above.

6.5.4. Using the CDS interface

See language specific class documentation for `CDS.CFGGroup`, `CDS.CFGAttribute`, and `CDS.CFGDB`.

6.5.5. Using the APC interface

See language specific on-line documentation for details on using `APC.AppControl`.

6.6. Advanced DICOM Programming Examples

6.6.1. Using StorageCommitmentSCU

The StoreCommitSCU and StoreCommitSCUAgent classes provide the user with an interface to the Storage Commitment Push Model SOP class as a Service Class User. The StoreCommitSCU class allows the user to send a list of DICOM SOP instances to a Storage Commitment SCP for which storage commitment is requested. The StoreCommitSCU class provides the interface for creating an association, creating a transaction UID, and sending the appropriate N-ACTION DIMSE message. The DicomDataService singleton's commitRequestSent method will be called in order to notify the OEM's database that the commit has been requested.

After sending the requests via N-Action, the `StoreCommitSCU` can be configured to hold the outbound association open. Otherwise, the StoreCommitSCUAgent class can be used to wait for inbound associations. In either case, the SCP will send N-Event-Report DIMSE messages back to the SCU (`StoreCommitSCU`). The DicomDataService singleton's commitCompleted method will be called so that the OEM can be updated with the commit completion status from the SCP.

Store related classes are in the `LaurelBridge.DSS` (DICOM Store Services) namespace. These classes are contained in the LaurelBridge.DSS.dll assembly.

6.6.1.1. Example – Send store commit requests and receive StoreCommitClientListener notifications

Send store commit requests and receive N-Event-Report notifications as objects are committed to long term storage.

See the ex_nstorecommit_scu.exe example for a complete program which can optionally start a StoreCommitSCUAgent in a new thread to receive incoming N-Event-Reports on a new association.

```
LaurelBridge.DSS.StoreCommitRequest request = new LaurelBridge.DSS.StoreCommitRequest();

//
// Put together the server_address.
//
String server_address = called_host_ + ":" + called_port_;
LOG.info("Called presentation address = " + server_address);

AssociationInfo ainfo = new AssociationInfo();
RequestedPresentationContext sc_ctx = new RequestedPresentationContext(1,
UID.SOPCLASSSTORECOMMITPUSHMODEL, new String[] { UID.TRANSFERLITTLEENDIANEXPLICIT,
UID.TRANSFERLITTLEENDIAN });

ainfo.calledPresentationAddress(server_address);
ainfo.calledTitle(called_ae_title_);
ainfo.callingTitle(calling_ae_title_);
ainfo.addRequestedPresentationContext(sc_ctx);

scu_ = new StoreCommitSCU(ainfo);

try
{
   // populate the referenced sop sequence from the command line args
   int count = 0;
   ReferencedSopSequence[] ref_sop_sequence = new
ReferencedSopSequence[file_list_.size()];
   request.transactionUid( DCMUID.makeUID());
   while (count < file_list_.size())
   {
      DicomFileInput dfi = new DicomFileInput( (String)file_list_.elementAt(count));
      dfi.open();
      DicomDataSet dds = dfi.readDataSetNoPixels();
      dfi.close();

      // create a sequence item with the uids
      ReferencedSopSequence ref_sop_sequence_item = new ReferencedSopSequence();
      ref_sop_sequence_item.referencedSopclassUid(
dds.getElementStringValue(DCM.E_SOPCLASS_UID));
      ref_sop_sequence_item.referencedSopinstanceUid(
dds.getElementStringValue(DCM.E_SOPINSTANCE_UID));

      // add the sequence item to the vector of items
      ref_sop_sequence[count] = ref_sop_sequence_item;
      count++;
   }

   // add the vector of sequence items to the request, it will be converted to
   // a sequence element containing one data set item for each object in the
   // vector
```

```
       request.referencedSopSequence(ref_sop_sequence);

       scu_.requestAssociation();
       scu_.nAction(request, 10, 10);
       scu_.waitForNEvent( 1 );
       scu_.releaseAssociation();
       exit_status_ = 0;
```

6.6.2. Using MWLClient

The `MWLClient` class provides a powerful and easy-to-use interface for accessing DICOM Modality Worklist providers. The user builds a query using either the basic `DicomDataSet`/`DicomElement` classes or the `ModalityWorklistItem` and related wrapper classes. This query is then sent by the `MWLClient` object. As responses arrive, they are either stored in a collection or delivered back to the client as they arrive.

For an example of writing your own `MWLClient` class, see Section 4.5.6 and `LaurelBridge.DCS.QRSCU` and `LaurelBridge.DCS.DicomSCU` API documentation.

6.6.2.1. Example – Send Worklist Query and wait for all responses before continuing

```
See %DCF_ROOT%\devel\cssrc\examples\ex_ndcf_mwl_scu for an example.

Note with this example the program processes incoming DIMSE Messages.
```

6.6.2.2. Example – Send Worklist Query and handle responses as they arrive

```
See %DCF_ROOT%\devel\cssrc\examples\ex_ndcf_query_scu (Query/Retrieve) for an example of
installing an event handler to be notified (with the query result as the payload) when a
DIMSE Message has been processed.
```

6.6.3. Using MPPSClient

The `MPPSClient` is used to communicate with Modality Performed Procedure Step Service Class providers or servers.

`MPPSClient` creates and updates instances of Modality Performed Procedure Step objects. It sends N-Create and N-Set DIMSE messages to an MPPS SCP or server. The user instructs the `MPPSClient` to connect to the SCP, and uses the `n_set()` and `n_create()` methods to send the appropriate DIMSE messages.

6.6.3.1. Example – MPPSClient Console Application

See %*`DCF_ROOT%\devel\cssrc\examples\ex_nmpps_scu\ex_nmpps_scu.cs`* for a complete console application example.

6.6.3.2. Example – Send DIMSE N-CREATE or N-SET messages to a MPPS Server

The method below will send the appropriate DIMSE N-CREATE or N-SET message to a MPPS Server.

```
public virtual void  runJob()
{
   //
   // Decide on whether to do an n-create or n-set
   //
   if (f_opt_create_ && f_opt_set_)
   {
       throw new DCSException("Cannot n-set and n-create at the same time");
   }

   //
   // Put together the server_address.
   //
   System.String server_address = host_ + ":" + port_;
   LOG.info("Called presentation address = " + server_address);

   AssociationInfo ainfo = new AssociationInfo();
   RequestedPresentationContext mpps_ctx = new RequestedPresentationContext(1,
UID.SOPPERFORMEDPROCEDURESTEP, new System.String[]{ts_uid_});

   ainfo.calledPresentationAddress(server_address);
   ainfo.calledTitle(called_ae_title_);
   ainfo.callingTitle(calling_ae_title_);
   ainfo.addRequestedPresentationContext(mpps_ctx);

   scu_ = new MPPSSCU(ainfo);

   scu_.requestAssociation();
   CFGGroup mpps_cfg = null;
   try
   {
       mpps_cfg = CFGDB.Instance.loadGroup(cfg_file_, true);
   }
   catch (CDSException cds_e)
   {
       LOG.error(- 1, "Error loading mpps cfg file " + usage(), cds_e);
       System.Environment.Exit(- 1);
   }
   LOG.info("MPPS CFGGroup = " + mpps_cfg);
   DicomDataSet ds = new DicomDataSet(mpps_cfg);
   ModalityPerformedProcedureStep procedure = new ModalityPerformedProcedureStep(ds);

   if (f_opt_create_)
   {
       scu_.nCreate(procedure, 10);
   }
   else if (f_opt_set_)
   {
       scu_.nSet(procedure, 10);
   }

   scu_.releaseAssociation();
   exit_status_ = 0;
}
```

6.6.4. C# Store, Q/R, and MWL Server-Related Examples

A common application of the DICOM protocol is in creating an image archive. An OEM may have special requirements for how images and patient information are stored in a database. The DCF provides APIs that are structured such that the OEM can easily customize the handling of image or other DICOM datasets without the need to deal with the mechanics of negotiating associations, handling sockets, PDUs or DIMSE messages.

The `DicomDataService` interface provides the mechanism for customizing the handling of DICOM datasets. Generic DCF protocol handling objects such as `StoreSCP`, `QRSCP` (Query Retrieve), `MWLSCP` (Modality Worklist) invoke `DicomDataService` methods to access the local storage facilities. The reference implementation adapter for the `DicomDataService` interface stores objects in the file system and provides minimal searching capabilities to support testing. Other implementations or adapters can be written that behave differently.

The example `%DCF_ROOT%\devel\examples\ex_nstore_scp` shows a simple storage server that sends incoming DICOM objects to the file system using the default `DicomDataService_a` (DICOM Data Service adapter) in `%DCF_ROOT%\devel\cssrc\DDS_a`. By installing a particular `DicomDataService_a` all incoming DICOM images are passed to the `storeObject()` method defined in that class.

The source file `ex_nstore_scp.cs` contains the function `main()` which installs the `DicomDataService` adapter and enters the loop which waits for incoming DICOM associations.

Note: To support MWL and Query Retrieve, the `findObjects()`, `findObjectsForTransfer()`, and `loadObject()` methods would have to be implemented. See example implementations, which are listed below.

The directory `$DCF_ROOT/devel/cssrc/ex_nqr_scp` shows a simple Query/Retrieve server that searches a "canned" set of DICOM objects in response to C-FIND requests and C-MOVE and C-GET requests and performs the appropriate matching. It either returns the list of found objects for a C-FIND or performs C-STORE operations if a C-MOVE or a C-GET was requested.

See `$DCF_ROOT/devel/cssrc/examples/ex_nqr_scp/ex_nqr_scp.cs`. This program is almost identical to `ex_nstore_scp.cs` with the exception that it uses `QRServer` class instead of `StoreServer` class. The directory `$DCF_ROOT/devel/cssrc/ex_nqr_scp` shows a simple Query/Retrieve server that searches a "canned" set of DICOM objects in response to C-FIND requests and C-MOVE and C-GET requests and performs the appropriate matching. The list of "canned" objects is created from thefiles found in `$DCF_ROOT/test/qr` directory. It either returns the list of found objects for a C-FIND or performs C-STORE operations if a C-MOVE or a C-GET was requested.

The directory `%DCF_ROOT%\devel\cssrc\ex_nmwl_scp` shows a simple Modality Worklist server that searches a "canned" set of DICOM objects in response to C-FIND requests performs the appropriate matching. It returns the list of found objects for a C-FIND. The list of "canned" objects is created from thefiles found in `$DCF_ROOT/test/worklist` directory.

See `%DCF_ROOT%\devel\cssrc\examples\ex_nmwl_scp\ex_nmwl_scp.cs`. This program is almost identical to `ex_nstore_scp.cs` with the exception that it uses `MWLServer` class instead of StoreServer class.

6.6.4.1. Using the MWL Server as an MPPS Server

Both the C# and Java worklist server examples (`ex_nmwl_scp` and `ex_jmwl_scp`) are also MPPS servers by default.

MPPS N-CREATE messages and N-SET messages are stored to the currently installed `DicomDataService` adapter. If you are using the default file system mode `DicomDataService` adapter, you can tell `DicomDataService` to store the both the command and data data-sets from N-CREATE and N-SET messages. This is useful if you want to be able to tell whether the stored object came from an N-CREATE or an N-SET message as this information is sent in the Command Data set of a DIMSE message. This functionality can be turned on by setting the appropriate CFG attribute to "*YES*", e.g.,

```
      /apps/defaults/ex_nmwl_server/java_lib/DDS_a/save_command_data YES
or
      /apps/defaults/ex_nmwl_server/cs_lib/DDS_a/save_command_data YES
```

6.6.4.2. Example – Implementing a custom storeObject() method

The following example illustrates how to create your own `DicomDataService` Adapter and implement the `storeObject()` method.

Note: your `DicomDataService` adapter class can have any name or be part of any assembly. The only requirements on your implementation are that:

- It must implement the abstract class `LaurelBridge.DDS.DicomDataService`; and
- You must "install" that implementation before the first time it will be used: you should add a static method called "setup" to your implementation to "install" it.

The following illustrates what part of your `DicomDataService` adapter would look like:

```
using LaurelBridge.DDS;

namespace OEM.StoreTest

{
using LaurelBridge.DDS;

public class OEMDataServiceAdapter : LaurelBridge.DicomDataService

{
      //protected constructor
      //You can only create one of things be calling the public static setup method.

      protected OEMDataServiceAdapter( String args[] )
      {
            //put initializiation code
      }

      //no default constructors allowed
      protected OEMDataServiceAdapter
      {
            LOG.error( -1, "illegal use of default constructor" );
      }

      //Here's where you add your code to store images to your backend
      public DicomPersistentObjectDescriptor storeObject(
                  AssociationAcceptor association_acceptor,
                  DimseMessage c_store_rq )

            throws DDSException
      {
            //Your implementation goes here
      }

      //Do something like the following for the other abstract methods
      //in DicomDataService base class.

      public abstract DicomDataSet loadObject(
            DicomPersistentObjectDescriptor dpod,
            boolean f_read_pixel_data )
                  throws DDSException
      {
            throw new DDSException( "loadObject unimplemented")
      }

      //Add a public setup method to install your implementation
      public static void setup( String args[] )
            throws DDSException
      {
            OEMDataServiceAdapter instance = new OEMDataServiceAdapter( args );
            //base class method that will set a new installed implementation
            setInstance( instance );
      }

}  //end of class OEMDataSetAdapter
```

In your application's `main()` method, before you begin accepting incoming associations, call your `setup()` method, for example:

```
public static void Main( string[] args )
{

...

      OEM.StoreTest.OEMDataServiceAdapter.setup( args );

      //begin accepting associations

...

}
```

You could, in fact, simply replace the following line in the example program *`ex_nstore_scp.cs`*

```
    DicomDataService_a.setup( args );
```

with this line:

```
    OEM.StoreTest.OEMDataServiceAdapter.setup( args );
```

and then once this change is made *`ex_nstore_scp.cs`* will use your new `DicomDataService` adapter's `storeObject()` method whenever an incoming DICOM Image is stored.

The other methods in the `DicomDataService` interface support operations like finding previously stored objects (`findObjects()` method) or loading previously stored objects (`loadObject()` method); this functionality is used to support the Query/Retrieve or Worklist SOP classes.

6.6.4.3. Example – How DicomDataService (DDS) gets called:

`DicomDataService` is a singleton that is created at init time when `setup()` is called on a `DicomDataService` subclass (the adapter).

In your store SCP app you do something like:

```
AssociationManager amgr = new AssociationManager();
StoreServer store_server = new StoreServer( amgr );
amgr.run();
```

When `AssociationMgr.run()` detects a new connection, it does the following:

Create a new thread for the association, so association manager can go back to waiting for incoming assocs.

In the new per-association thread the following are done:

- Create one `AssociationAcceptor`.
- Call the `configPolicyMgr` that was registered with amgr to get session settings for the association.
- Call `beginAssociation()` on each `AssociationListener` that is registered with amgr.
  One of the listeners is store_server which does the following:

create `StoreSCP` which will be devoted to this association. `StoreSCP` registers with the `AssociationAcceptor` as a `PresentationContextAcceptor` for any supported SOP classes.

- `AssociationAcceptor` performs association negotiation during which it may associate `StoreSCP` with one or more of the accepted presentation-contexts.
- `AssociationAcceptor` now enters a message loop, reading and dispatching messages.
  In addition to implementing `PresentationContextAcceptor` which is used during connection setup, `StoreSCP` also implements `DimseMessageHandler` which is used as messages are received by the main loop in `AssociationAcceptor`.
- When `AssociationAcceptor` receives a C-Store-Request dimse message, it dispatches it to the handler for the indicated presentation-context (`StoreSCP` handles all accepted store contexts).
- `StoreSCP` calls `DicomDataService.Instance.storeObject()` which lands in your `storeObject()` code.

6.6.4.4. Example – Adding OEM specific data to DicomSessionSettings:

When implementing a custom `StoreSCP` it may become desirable to add custom processing for certain clients, for example, you may customize your actions based on the calling AE-Title of the client. When adding OEM specific data to `DicomSessionSettings` the main issue to be aware of is to assure that you don't use a group or element name that `DCS.DicomSessionSettings` is already using.

To add a setting you could do something like the following:

```
MyStoreSCPApp
{
   DicomSessionSettings getSessionSettings( AssociationAcceptor acceptor )
   {
      // create default settings
        DicomSessionSettings ss = new DicomSessionSettings();

      // create custom config info
        CFGGroup my_data = new CFGGroup("OEM.DicomDataService_a");
        my_data.setAttributeValue("output_dir", "/somedir/" +
         acceptor.AssociationInfo.CalledTitle );

      // add to CFGGroup contained by DicomSessionSettings
        ss.Cfg.addGroup( my_data );
        return ss;
   }
}

DicomDataService_a
{
   storeObject( AssociationAcceptor acceptor, ... )
   {
      string dir =
acceptor.SessionSettings.Cfg.getAttributeValue("OEM.DicomDataService_a/output_dir");
      ...
   }
}
```

6.6.4.5. Example – Receiving or Logging Retired SOP classes:

Suppose you receive an image that has a SOP class UID of 1.2.840.10008.5.1.4.1.1.6, which is the retired Ultra Sound Image Storage UID. You can make your StoreSCP accept it by adding this UID to the list of StoreSCP supported_sop_classes in your apps configuration file (e.g., in cs_lib\DSS, under

DSS/StoreSCP/default_session_cfg/supported_sop_classes) or by adding this UID to session settings (see 6.6.4.4). You will need to restart your DCF system after making the change, since you need to restart the DCF if you are using the CFGDB.

You can find the Retired SOP classes by looking at the UID's appendix of Chapter 6 of the DICOM standard.

With the ex_nstore_scp.exe example, one way to check for this kind of condition is to override the endAssociation() method of AssociationListener (see the ex_nstore_scp example). You can get the AssociationInfo object from the AssociationAcceptor and check the list of rejected presentation contexts. With that information you may choose to take some action or simply log the fact that a presentation context was rejected.

You can add the following code to the ex_nstore_scp example to log a rejection:

```
...

/// <summary> Optional implementation of AssociationListener interface.
/// Indicates that an association has ended.
/// </summary>
/// <param name="assoc"> the object handling the association.
/// </param>
public virtual void  endAssociation(AssociationAcceptor assoc)
{
    LOG.info("Association has ended.");
    System.Collections.ArrayList ctx_list =
        assoc.AssociationInfo.rejectedPresentationContextList();

    //If there were any rejected presentation contexts do something.
    for( int i=0; i<ctx_list.Count; i++ )
    {
       RejectedPresentationContext ctx =
           (RejectedPresentationContext) ctx_list[i];
       LOG.info( "RejectedPresentationContext = " + ctx.ToString() );
    }
}

...
```

6.6.4.6. Writing a Custom DICOM SCP

You can extend *%DCF_ROOT%\devel\cssrc\examples\ex_n_oem_scp\ex_n_oem_scp.cs.*

6.7. DICOM compression transfer syntax support for C#

DCF C# applications can handle DICOM datasets in any transfer syntax for non-pixel data operations provided that compression pass through mode is turned on (except for DICOM Deflated Little Endian Syntax and JPIP Transfer syntaxes).

DCF C# applications can compress and decompress data sets in these encapsulated transfer syntaxes:

• 1.2.840.10008.1.2.4.5	RLE Lossless
• 1.2.840.10008.1.2.4.50	JPEG 8 bit lossy
• 1.2.840.10008.1.2.4.51	JPEG 12 bit lossy
• 1.2.840.10008.1.2.4.57	JPEG lossless
• 1.2.840.10008.1.2.4.70	JPEG lossless (predictor selection=1)
• 1.2.840.10008.1.2.4.90	JPEG-2000 lossless
• 1.2.840.10008.1.2.4.91	JPEG-2000 lossy

RLE Lossless transfer syntax is supported for compression of single frame data sets. RLE Lossless transfer syntax is supported for the decompression of single frame and multi-frame data sets.

Look at the settings under the DCS section of an application configuration file or in a DCS component configuration file to see options that can be configured for compression.

Note: If you are using the Aware, Inc., JPEG library, that this does not support .57.

7. Using DCF System Manager to control processes

The application `dcf_sysmgr` provides a convenient mechanism to manage startup and shutdown of a collection of related server processes. The system manager can start C++, Java or C# or other applications either as foreground (utility) processes or background (server) processes. Using either CORBA or COM IPC, an application can communicate with the system manager to request system startup or shutdown, or to query for the status of the system. Likewise the system manager will communicate with child processes to determine their status or to request a shutdown. The system manager may be started as needed, as part of launching a set of applications, or automatically at initialization time (e.g., in an /etc/rc file on Unix, or by the Service Control Manager on Windows).

Note that it is not required that any DCF application use the system manager. Use of the system manager is controlled by the attribute **APC_a/handle_external_shutdown_rq** in the `cpp_lib` (C++), `java_lib` (Java), or `cs_lib` (C#) configuration sub-group, depending on the implementation language. If this attribute is true, then the dcf_sysmgr app is expected to send shutdown messages, and the application will attempt to send "registerApplication" and "applicationReady" messages to the system manager.

7.1. Installing and Starting the System Manager

7.1.1. Installing and starting as a service on Windows

Register the system manager as a Windows service using:

```
dcf_sysmgr /Service
```

The service will be named:

<product_name>.<DCF version>.dcfsysmgr

e.g.,

DCF.3.1.3a.dcfsysmgr

Make sure that the PATH contains the location of `dcf_sysmgr.exe` and any dll's that it will need.

You can use the "Services" administrator tool to adjust how the service is started, or to stop and start the service. The "sc" command can also be used to start the system manager. For example:

```
sc start DCF.3.1.3a.dcfsysmgr
```

See Section 7.3.1 below for information about attributes that control the service's behavior.

7.1.2. Installing and starting as a normal server process on Windows

Register the system manager as a LocalServer32 using:

```
dcf_sysmgr /RegServer
```

Note that the DCF toolkit installer will optionally perform this step on a development box.

Start the `dcf_sysmgr` process either by running it from the command line, in a batch file, or by using the `dcfstart.pl` or `apc_client.exe` programs, which will start `dcf_sysmgr` if it is not already running.

7.1.3. Installing and starting on Unix

Start the dcf_sysmgr process like any other Unix server process or daemon. For example:

```
dcf_sysmgr &
```

See Section 7.3.1 below for information about attributes that control the service's behavior.

7.2. System Manager Related Interfaces

The system manager implements and uses various CORBA and COM interfaces. CORBA interfaces are defined by the DAPC (Distributed Application Control) idl library, while the COM interfaces are defined as part of the dcf_sysmgr C++ application. These interfaces are:

Interface	**Description**
DAPC::SystemManager	CORBA interface to start/stop/get-status of a system
DAPC::ApplicationControl	CORBA interface by which dcf_sysmgr requests child server (C++, Java) applications to shutdown
DAPC::SystemStatusListener	CORBA interface used to deliver status events to system management tools
DAPC::ProcessStatusListener	CORBA interface used to deliver status events to system management tools
ICOMAppControl	COM interface by which dcf_sysmgr requests child server (C#) applications to shutdown
ICOMSystemManager	COM interface to start/stop/get-status of a system
_ICOMSystemManagerEvents	COM interface used to deliver process and system status events to management tools

The application **apc_client** is used to communicate with the system manager using various interfaces from the DAPC component. The scripts `dcfstart.pl`, `dcfstop.pl` and `dcfsysstatus.pl` are simple wrappers that invoke apc_client.

A GUI based example is available for C#.Net. This application can be found in DCF_ROOT/devel/cssrc/examples/NDCFSystemMonitor. System manager must be running prior to starting NDCFSystemMonitor. A system configuration can be selected, and the system manager can be requested to start or stop that system. The status of the system and each process is updated dynamically. Note that if "exit_after_system_stopped" is set in the dcf_sysmgr app configuration, then the system manager process will exit after the first "Stop-System" request is completed.

As of DCF version 3.1.4b, both COM and CORBA clients locate the system manager using a stringified object reference which is stored in the directory given by the DCF_TMP environment variable. For CORBA clients, this is a standard IIOP stringified object reference. For COM, a string representation of an ObjRefMoniker object is used. On Windows systems, a single instance of the COMSystemManager class is created and registered in the Running Object Table. All COM clients interact with this object.

7.3. System Manager Configuration

The system manager uses two sources of configuration data. First, as it is itself a DCF application, it has an *application configuration*. When a request to start a system is given, a *system configuration* is provided which describes the programs that will be started as well as the startup and shutdown order, etc.

7.3.1. System Manager Application Configuration

By default, the file $DCF_CFG/apps/defaults/dcf_sysmgr provides the application configuration. Below are *some* of the attributes from that file:

```
#=============================================================================
# per-instance information for the dcf_sysmgr component
#=============================================================================
[ cpp_ipc_app/dcf_sysmgr ]
debug_flags = 0x00000

#
# If true, dcf_sysmgr will exit after system shutdown is complete.
# Will also exit if the first startsystem request fails.
# This means a call to "stopsystem" will behave the same as "shutdown"
# If this is false, then "stopsystem" will shutdown child processes,
# but dcf_sysmgr will remain running.
#
exit_after_system_stopped = YES

#
# Name of system configuration to auto-start
# e.g. file:C:/Program Files/DCF_3.1.3a/cfg/systems/store_server_win32.cfg
# or /systems/ndcds_server_win32.cfg
# In the latter case, CFGDB will look under the directory indicated by the
# DCF_CFG environment variable.
#
#auto_start_system_cfg =
```

There are five main configuration attributes that are used to control the system manager's behavior when starting and stopping servers or applications.

- `auto_start_system_cfg` – This tells the system manager the name of the configuration of processes to start automatically. You can manually indicate a system configuration to start via a command-line argument to dcfrestart.pl or to apc_client; setting this attribute tells the system manager to start this system *without* having to specify the configuration name on the command line. (In fact, if this value is set, it will always be used when the system manager is run, even if you do specify a different configuration name on the command line.)
- `exit_after_system_stopped` – This tells the system manager that it should not itself exit after all of the servers / applications in the system configuration have stopped. This is

useful when you are running the system manager as a Windows service – you may desire for the various DICOM servers to stop but you don't want the system manager to stop.

- `exit_after_system_error` – This is similar to `exit_after_system_stopped`, except that it tells the system manager whether or not to exit if an error occurs in one of the servers / applications that causes that server / application to exit with an error.
- `restart_after_system_error` – This tells the system manager if it should restart the `auto_start_system_cfg` if one of its processes failed unexpectedly. This can be useful if your system encounters an error and exits – the system manager can then restart the configuration so that there is only a brief disruption in handling requests and data flow.
- `max_auto_restarts` – This is how many times the system should auto-restart after a system failure occurs. The default value is 3; if your system fails more than 3 times, there may be a serious error that should be investigated and resolved before resuming normal operations. You can set this to "-1" to have it restart indefinitely.

Note that some of these values are applicable only if `auto_start_system_cfg` is set.

When you install the dcf_sysmgr as a service, you will probably want to set the above attributes to the following values. This will configure the system manager so that it will continue running your system configuration even if an error occurs.

- `auto_start_system_cfg = <name of the system configuration>`
- `exit_after_system_stopped = NO`
- `exit_after_system_error = NO`
- `restart_after_system_error = YES`

7.3.2. System Manager System Configuration

The format of a system configuration file is shown below. All of the example system configurations which can be run from the remote service interface are started using the system manager. You can see the configuration files for these systems in $DCF_CFG/systems.

```
#
# System information group: general information about this configuration
#
[ system_info ]
name = < name of system >
description = < description of this system >
#
# If the platform attribute is present, then the
# current platform (given by $DCF_PLATFORM)
# must match one of the attribute values. This is
# currently only used by the DCF service web pages
# to filter which system configurations are shown.
# see DCF_ROOT/platforms.cfg for a complete list
# of defined platforms.
#
platform = <first supported platform>
platform = <next supported platform>
…
#
# Any environment variables in the "environment" group
# are inherited by processes started by dcf_sysmgr.
# example attribute in environment group:
```

```
#       DCF_ROOT = C:\LBS\DCF
[ environment ]
<env_var_name_1> = <value 1>
…
#
# List of processes that are run as part of the system startup sequence.
# Each value for the "process" attribute is a group name for
# a process configuration group below.
#
# The processes are started in the same order as the attribute values.
#
[ startup ]
process = <name of first startup process>
process = <name of next startup process>
…
#
# List of processes to run (for type=utility), or to stop (for
# server procs that from the [startup] group) as part of the
# system shutdown sequence.
#
# If one of these entries is a utility process, it is run in the
# foreground. If it is a server process, it must correspond to
# a server process from the startup group, in which case it
# is issued a shutdown or terminate request as appropriate.
#
# Processes are run/stopped in the same order as the attribute
# values.
#
[ shutdown ]
process = <name of first shutdown process>
process = <name of next shutdown process>
…

#======================================================
# Per Process configuration settings:
#======================================================
# Each Process group referenced by either the "startup" or
# "shutdown" groups has the following format:
#
# The group name is the string in the startup or shutdown group
# e.g. "dcf_store_scp.001"
#
#======================================================
[ <process_name> ]

#
# Process type:
#     utility : runs in the foreground. System Manager will execute
#            this process and wait for its termination.
#
#    dcf_server : A DCF server, i.e., one that is expected to send
#            register_application and application_ready messages, and can
#            accept shutdown messages.
#            Server processes are run in the background.
#            The next process will not be started until this process
#            completes its start up sequence - i.e., sending
#            register-app and then app-ready.
#
#    server2 : A non-DCF server, i.e., some other app that is not
#            DCF system manager aware.
#            server2 processes are run in the background. The next
#            process is started immediately after starting one of these.
#
#    server : synonym for dcf_server. ("server" is used for compatibility
#            with old versions of dcf_runsystem.pl.)
#
# Required attribute
```

```
#
type = utility | server | server2 | dcf_server

#
# For a utility process, the exit code that is expected.
# If you do not care about the exit code, use the special
# value "IGNORE_RETURN".
# Otherwise, if the exit code does not match the expected, the
# system will shutdown. If this occurs during shutdown,
# an error is logged, and shutdown continues.
#
# Default value = 0
#
expected_exit_code = <decimal number> | IGNORE_RETURN

#
# For a utility process, the number of seconds after starting
# to wait for termination.
# If this time is exceeded during startup, the system will shutdown.
# If this time is exceeded during shutdown, the utility is
# terminated, and shutdown continues.
# This timeout is used when utility processes are run either
# as part of the startup or shutdown sequence.
#
# Default value = 30
#
wait_timeout_seconds = <seconds>

#
# For a server or dcf_server process, the number of seconds
# to wait after starting the process for the register_application
# message.
# If this time is exceeded, the system will shutdown.
#
# Default value = 15
#
register_app_timeout_seconds = <seconds>


#
# For a server or dcf_server process, the number of seconds
# to wait after receiving the register_application
# message for the application_ready message.
# If this time is exceeded, the system will shutdown.
#
# Default value = 15
#
app_ready_timeout_seconds = <seconds>


#
# Number of seconds after starting a server2 process or after
# receiving application_ready from a server or dcf_server process
# before the next process is started.
#
# Default value = 0
#
post_start_delay_seconds = <seconds>


#
# Number of seconds to wait for process termination after a
# shutdown request is issued to a dcf_server process.
# If this time is exceeded, the process is terminated and
# shutdown continues.
#
# Default value = 30
```

```
#
shutdown_timeout_seconds = <seconds>

#
# The program and command line arguments to run for this process.
#
# Required attribute
#
command = <command line>

#
# Name of file to which stdout and stderr for this process will
# be redirected.
#
# Default value is "$DCF_LOG/<process_name>.out.log"
#
# A future release may provide a special value that redirects
# the process' stdout/stderr to the dcf_sysmgr's stdout/stderr.
# We may also allow stdout and stderr to go to different files.
# Note this has nothing to do with output for the DCF logger or
# other logging interfaces.
#
stdout_name =  <filename>

#
# What to do if the process terminates while the system is starting
# or running:
#
# shutdown_system : the system will be shutdown. This is a required process.
# ignore : do nothing. This process is optional.
# restart : (Not currently implemented - in a future release, the
#             process will be restarted)
#
# Default value = shutdown_system
#
terminate_action = shutdown_system | ignore | restart
```

7.4. System startup for a DCF server application

Following is the initialization sequence for a typical DCF server application. (Note that in this example, use of the system manager is enabled, as is use of the server mode Configuration Data Service.)

1. dcf_sysmgr starts application “server_xyz”
2. server_xyz performs preliminary initialization, e.g., set up IPC capabilities.
3. server_xyz initializes the Configuration Data Service adapter (CDS_a.CFGDB_a) in either file-system or server mode.
4. server_xyz initializes the Application Control adapter (APC_a.AppControl_a). The application configuration is read and used to initialize the process configuration. Optionally, the process configuration is saved to the CDS CFGDB. Optionally, server_xyz registers as an observer of the process configuration. At the end of the AppControl adapter setup, server_xyz sends “registerApplication” message to dcf_sysmgr. This indicates that server_xyz is not fully initialized, but is at least able to receive a shutdownApplication message from dcf_sysmgr if the startup is aborted.
5. server_xyz completes initialization – this may include setting up a DICOM server socket, performing other application specific startup tasks, etc.

6. server_xyz sends “applicationReady” message to dcf_sysmgr. At this point, dcf_sysmgr will proceed to start the next process defined by the system configuration. If this is the last process, then the system state is changed from “STARTING” to “RUNNING”.

7.5. System shutdown for a DCF server application

System shutdown is very simple. dcf_sysmgr stops processes in the order defined by the “`[ shutdown ]`” configuration group in the system configuration. Note that if a utility process is contained in the shutdown group, that process is run in the foreground; if a server-process that was also contained in the startup group is referenced, then that server is issued a shutdown request. For DCF servers (example server_xyz) the following takes place:

1. dcf_sysmgr sends shutdownApplication message to server_xyz. This message is delivered to the AppControl adapter.
2. AppControl notifies all shutdown listeners that phase-1 shutdown has started. All but critical services (logging, etc.) are stopped. For a DICOM application, for instance, the AssociationManager object will stop accepting associations at this point.
3. The process configuration, if saved in CDS, is deleted (optionally). Any observers that are registered with CDS are unregistered.
4. AppControl notifies all shutdown listeners that phase-2 shutdown has started. Any components that have not yet cleaned up do so now.
5. The AppControl event loop is stopped. If the application has called AppControl.Instance.runEventLoop(), then that method returns, and the application is free to exit.

8. The DCF Development Environment

8.1. Using the dcfmake.pl utility

The utility script, `dcfmake.pl`, is a program that builds DCF components or DCF based OEM components. You can use *dcfmake.pl* to build your applications, libraries, etc., or you can use your favorite make program, IDE, etc., provided you reference the DCF include, library, classes, directories appropriately.

The `dcfmake.pl` script functions in two primary modes – **index file mode** or **component build mode**. If the file `dcfmake.cfg` exists in the current directory, then index file mode is assumed. Otherwise component build mode is assumed.

In index file mode, `dcfmake.pl` opens the file `dcfmake.cfg` and reads a list of subdirectories (one per line). It sequentially descends into each of these directories and invokes `dcfmake.pl` recursively. This provides control over the build order of multiple components.

In Component build mode, `dcfmake.pl` performs the following steps

1. Read the component information file (`cinfo.cfg`)
2. Execute optional user defined "pre_gen" command or script
3. Generate Logging/Debugging/Configuration related instrumentation code as applicable for the component type
4. Generate Configuration meta data files as applicable for the component type
5. Generate a UNIX makefile or Windows project file
6. Execute optional user defined "post_gen" command or script
7. Execute the "make" command or the visual studio command line build application with the generated makefile or project file.

See the `ex_hello_world` example for a step-by-step example of running `dcfmake.pl` to create an application that uses the DCF development environment.

(See Appendix G: Using Perl with the DCF for information on simplifying the invocation of the Perl interpreter on Windows.)

8.1.1. Command line options for dcfmake.pl

The `dcfmake.pl` script supports the following command line options:

```
dcfmake.pl [options] <-- make options>
  -verbose : print useful information while building
  -cfgfile <name> : overrides the default "cinfo.cfg"
  -indexfile <name> : overrides the default "dcfmake.cfg"
  -build-config <debug|release> : overrides the default "debug"
  -action <build|rebuild|clean> : overrides the default "build"
  -force : regenerate metadata and instrumentation even if config file is not newer
  -keep : do not delete temporary files
  -no-execute: do not execute any commands or create any files
  -generate-code-only : create metadata and instrumentation, but do not run make/devenv
  -ctype-to-build <cpp_app|cpp_lib|java_lib|....> : only build components of this type
```

(Note ctype-to-build is matched as a regex, so for example "lib" matches cpp_lib, cpp_lib_pkg, java_lib,...)

Options may be abbreviated as long as there is no ambiguity – i.e., "–v" is OK and selects "–verbose", but "–c" is not OK, it must be either "–cf" or "–ct" to distinguish which option to select.

Any options following the optional '--' are passed directly to the make or Visual Studio's command line interface.

8.1.2. The cinfo.cfg file

The file `cinfo.cfg` provides information about your component (library, application, etc.). It normally exists in the same directory as your component. Each component must be contained in its own directory or folder.

The format of that file is defined by the following annotated example.

```
#==============================================================================
# static information common to all instances of the ex_hello_world component
#==============================================================================
[ component_info ]
# name for this component (required)
name = ex_hello_world
# type for this component (required)
type = cpp_app
# category for this component
category = examples
# file containing doc comments for auto-gen'ed docs
docfile = ex_hello_world.cpp
# description of this component (required)
description = Example first application program
version = 0.1

#================================================================================
# The build_info group is optional and contains settings that control dcfmake.pl
#================================================================================
[ build_info ]
# 1 if application configuration file should be created (default = 1)
# only valid for application type components
gen_app_cfg = 1
#
# 1 if generated Component Information instrumentation and config data
# should be generated (default=1)
#
gen_cinfo_code_and_data = 1
#
# 1 if makefile or VS project file should be created. Set to 0 for
# custom built (or built with IDE or other tool) build files.
#
gen_build_file = 1
#
# name of generated application configuration file
# (default is $DCF_USER_ROOT/devel/cfggen/apps/defaults/<name>)
# only valid for application type components
#
app_cfg_name =
#
# set if idl module name is different from component name
# only valid for idl_lib components
#
module_name =
#
# additional compiler options added to generated makefiles
#
cpp_options =
#
# additional linker options added to generated makefiles
```

```
#
link_options =
#
# additional include directories (for VS8+)
win_xml_inc_dirs =
#
# additional libraries to link with (for VS8+)
win_xml_libs =
#
# additional debug libraries to link with (for VS8+)
#
win_xml_debug_libs =
#
# additional preprocessor defines (for VS8+)
#
Win_xml_preproc_defines =
#
# extension for generated C++ source files (default = .cpp)
#
cinfo_cc_file_ext =
#
# command to execute prior to generating files (default = none)
#
pre_gen =
#
# command to execute after generating files (before invoking make) (default = none)
#
post_gen =
#
# command to execute after invoking make (default = none)
#
post_make =
#
# create executables in this directory(default = $DCF_BINDIR)
# only valid for application type components
#
bin_dir =
#
# create public CInfo header files in this directory
# (default = ${DCF_USER_ROOT}/include/$cinfo->{component_name})
inc_dir
#
# create libraries in this directory(default = $DCF_LIBDIR)
# only valid for library type components
#
lib_dir );
= .

#
# This group defines debug settings.
#
[ debug_controls ]
# debug_flag = <short_name>, <bit_value>, <description>
debug_flag = df_TEST1,  0x10000, place holder for test 1 debug setting
debug_flag = df_TEST2,  0x20000, Do something cool

# list of other DCF components required. This will affect makefile or dsp files,
# as well as the generated application configuration file if any.
[ required_components ]
component = cpp_lib_pkg/DCFCore
component = cpp_lib/DCFUtil
component = cpp_lib/LOG_a
component = cpp_lib/APC_a
component = cpp_lib/CDS_a
component = idl_lib/DCDS

#===============================================================================
# per-instance information for the ex_hello_world component
#===============================================================================
[ ex_hello_world ]
debug_flags = 0x00000

#===============================================================================
# The following sections allow the customization of the generated default
# application configuration.
# After the application configuration is created,
# selected library component configuration settings can be overridden.
# Note that this affects the settings for that library only within the context
# of this application.
#===============================================================================
[ lib_cfg_overrides ]

# this example changes the “use_log_server” attribute in the [LOG_a] group
# in the generated application configuration file.
[ lib_cfg_overrides/LOG_a ]
use_log_server = FALSE
```

8.1.3. Generated files for various component types

Prior to executing the actual "make" command, *`dcfmake.pl`* creates various files.

For each of the component physical types, the files generated are listed in the following table. Note that this list does not include the actual binary output of the various compiler or linker programs – i.e., .EXE, .SO, .DLL, .CLASS files, etc…

Type	**Generated files**
cpp_lib	<name>CInfo.cpp <name>CInfoL.h include/<name>/<name>CInfoP.h include/<name>/<name>CInfo.h devel/cfggen/components/cpp_lib/<name> makefile.dcf (unix platforms) <name>.dsp or <name.vcproj>(Windows platforms) <name>.dswf or <name>.sln (Windows platforms)
cpp_lib_src	<name>CInfo.cpp <name>CInfoL.h include/<name>/<name>CInfoP.h include/<name>/<name>CInfo.h devel/cfggen/components/cpp_lib_src/<name>
cpp_lib_pkg	devel/cfggen/components/cpp_lib_pkg/<name> makefile.dcf (unix platforms) <name>.dsp or <name.vcproj>(Windows platforms) <name>.dswf or <name>.sln (Windows platforms)
java_lib	CINFO.java LOG.java devel/cfggen/components/java_lib/<name> makefile.dcf <name>.dsp or <name.vcproj>(Windows platforms) <name>.dswf or <name>.sln (Windows platforms)
cpp_app	<name>CInfo.cpp <name>CInfoL.h name>CInfo.h devel/cfggen/components/cpp_app/<name> devel/cfggen/apps/defaults/<name> makefile.dcf (unix platforms) <name>.dsp or <name.vcproj>(Windows platforms) <name>.dswf or <name>.sln (Windows platforms)

Type	Generated files
cs_lib	CINFO.cs LOG.cs devel/cfggen/components/cs_lib/<name> <name>.csproj
cs_app	(For Console applications) CINFO.cs LOG.cs devel/cfggen/components/cs_app/<name> devel/cfggen/apps/defaults/<name> <name>.csproj
cs_win_app	(For GUI applications) CINFO.cs LOG.cs devel/cfggen/components/cs_win_app/<name> devel/cfggen/apps/defaults/<name> <name>.csproj
cpp_jni_lib	devel/cfggen/components/cpp_jni_lib/<name> makefile.dcf (unix platforms) <name>.dsp (Windows platforms) <name>.dswf (Windows platforms)
java_app	CINFO.java LOG.java devel/cfggen/components/java_app/<name> devel/cfggen/apps/defaults/<name> makefile.dcf <name>.dsp (Windows platforms) <name>.dswf (Windows platforms)
idl_lib	devel/cfggen/components/idl_lib/<name> makefile.dcf <name>.dsp (Windows platforms) <name>.dswf (Windows platforms)

An explanation of each of the generated files is contained in the following table:

Generated file name	Description
<name>CInfo.cpp	Component Information C++ file. Defines the <name>CInfo class. Holds component global information at runtime. (Generated by substituting text in devel/lib/templates/CInfo.cpp)

Generated file name	**Description**
<name>CInfoL.h	Component Information “Local” header file. Defines Logging and other macros used only within the component. (Generated by substituting text in devel/lib/templates/CInfoL.h)
include/<name>/<name>CInfoP.h	Component Information “Public” header file. Defines namespace and performs Windows DLL Import magic. Included by other source files which use this component. (Generated by substituting text in devel/lib/templates/CInfoP.h)
include/<name>/<name>CInfo.h	Component Information header file. Included by CInfoL.h and external source files which need to adjust debug settings for this component. (Generated by substituting text in devel/lib/templates/CInfo.h)
CINFO.java	Component Information Java file.
LOG.java	For Java, defines Logging macros used only within the component.
CINFO.cs	Component Information C# file.
LOG.cs	For C#, defines Logging macros used only with the component.
devel/cfggen/components/<type>/<name>	Configuration data common to all instances of this component. This includes build information, as well as the “[debug_controls]” and “[<name>]” groups from the original cinfo.cfg
devel/cfggen/apps/defaults/<name>	Default application configuration file. This contains a generated “[application_info]” group, plus a copy of the “devel/cfggen/components/<type>/<name>/<name>” group from each required library component.
makefile.dcf	Standard Unix or Visual Studio makefile (Generated by substituting text in devel/lib/<type>_gnumakefile or <type>_nmakefile
<name>.dsp	Visual Studio 6 project file – Each component maps to a VC++ project. (Generated by substituting text in devel/lib/<type>_dsp
<name>.dswf	Visual Studio 6 workspace “fragment” file. Files of this type are concatenated by the “makedsw.pl” utility to create a Visual studio workspace (.dsw) file.
<name>.vcproj	Visual Studio 2003/2005 C++ project file – Each component maps to a VS project. (Generated by substituting text in devel/lib/<type>_vcproj

Generated file name	Description
<name>.csproj	Visual Studio 2003/2005 C# project file – Each component maps to a VS project. (Generated by substituting text in devel/lib/<type>_vsproj
<name>.sln	Visual Studio 2003/2005 solution file.

8.2. Example: Creating a DCF library component

The following diagram shows the steps that are taken when a DCF library component is built. The example shows a C++ library component. Similar steps are taken for idl_lib, java_lib, and cs_lib component types.

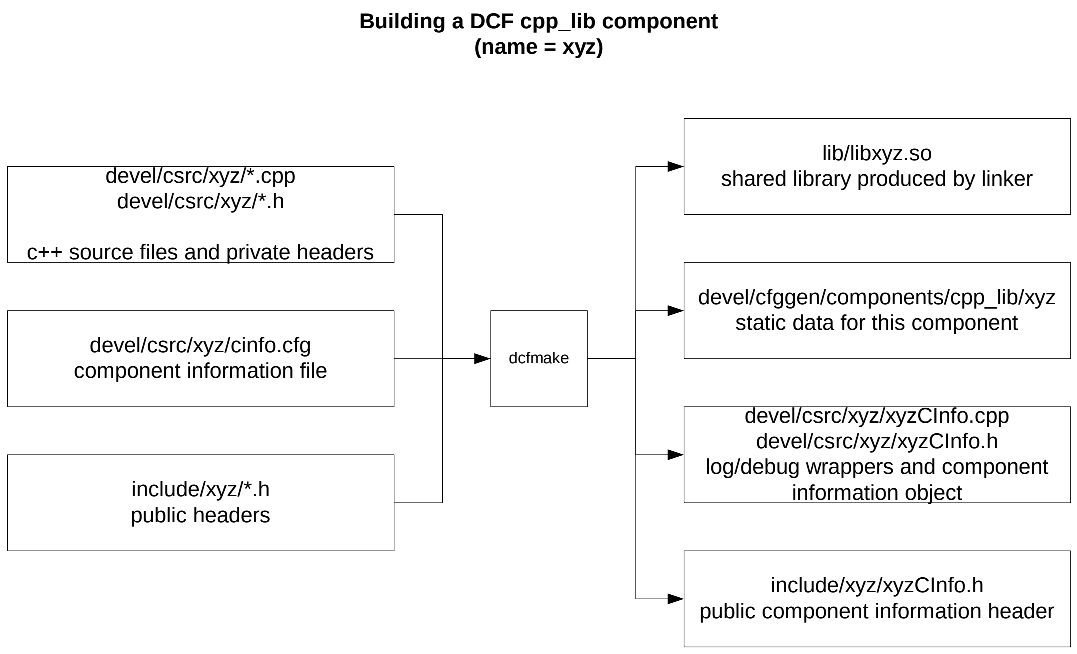

Building a DCF cpp_lib component
(name = xyz)
devel/csrc/xyz/*.cpp
devel/csrc/xyz/*.h
c++ source files and private headers
devel/csrc/xyz/cinfo.cfg
component information file
include/xyz/*.h
public headers
dcfmake
lib/libxyz.so
shared library produced by linker
devel/cfggen/components/cpp_lib/xyz
static data for this component
devel/csrc/xyz/xyzCInfo.cpp
devel/csrc/xyz/xyzCInfo.h
log/debug wrappers and component information object
include/xyz/xyzCInfo.h
public component information header

Figure 14: Creating a DCF Library Component

8.3. Example: Creating a DCF application component

The following diagram shows the steps taken to build a C++ application component. The procedures for creating a Java or C# application are similar.

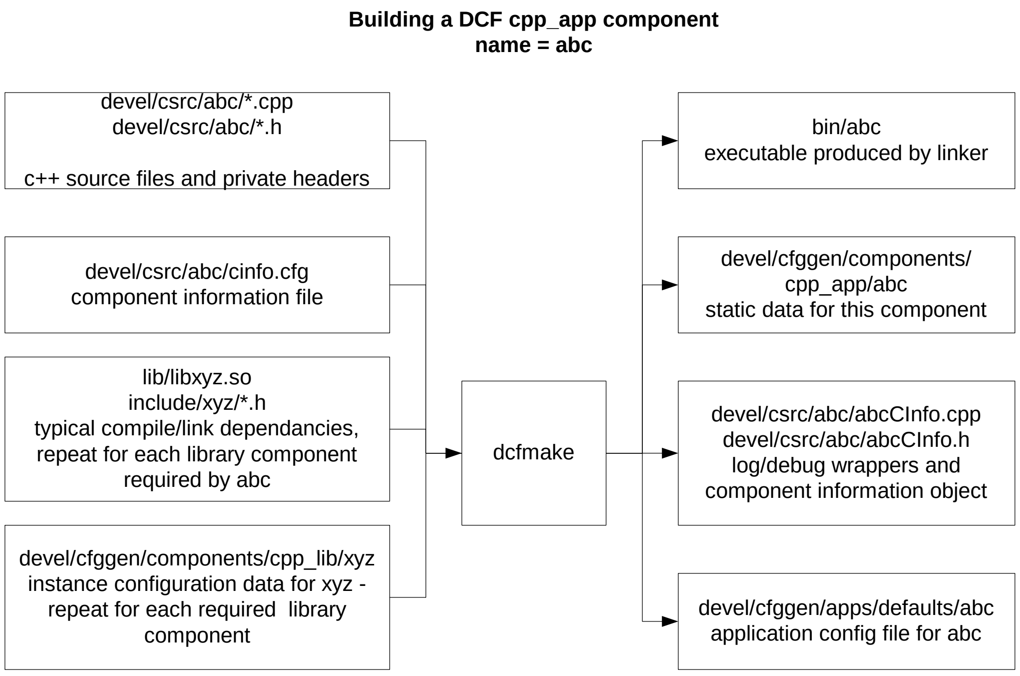

Building a DCF cpp_app component
name = abc
devel/csrc/abc/*.cpp
devel/csrc/abc/*.h
c++ source files and private headers
devel/csrc/abc/cinfo.cfg
component information file
lib/libxyz.so
include/xyz/*.h
typical compile/link dependancies, repeat for each library component required by abc
devel/cfggen/components/cpp_lib/xyz
instance configuration data for xyz - repeat for each required library component
dcfmake
bin/abc
executable produced by linker
devel/cfggen/components/cpp_app/abc
static data for this component
devel/csrc/abc/abcCInfo.cpp
devel/csrc/abc/abcCInfo.h
log/debug wrappers and component information object
devel/cfggen/apps/defaults/abc
application config file for abc

Figure 15: Creating a DCF Application Component

8.4. Using iodgen to create DICOM dataset wrappers to represent an IOD

The DCF comes with an advanced code generation tool – *iodgen.pl*. A DICOM IOD or Information Object Definition is a data set specification for an image or some other radiology related entity. For instance, each image from the various supported modality types is defined as an IOD. The IOD specification in the standard defines a collection of modules that make up the object. The module specification in the standard defines a collection of elements that make up the module.

iodgen will read configuration data for an IOD and generate classes that provide access methods to each element in that class. Iodgen can generate C++, Java and C#.NET output.

iodgen uses the DICOM data dictionary. For each element, the VR (value representation), VM (value multiplicity), name, and description are looked up. This information is used to generate code to get or set that element in the dataset contained by the IOD object.

To create the code for the DicomImage C++ class, we ran

```
perl –S iodgen.pl DicomImage.iod
```

The configuration file *DicomImage.iod* contains the following:

```
#

# DicomImage iod represents a generic image SOP instance
# All of the attributes from the General Image and Image Pixel
# modules are there, as well as selected additional attributes
#
name = DicomImage
uidname = "1.2.3"
specref = chapter 3
module = GeneralImage
module = ImagePixel
module = ModalityLUT
module = VOILUT
```

The name field “`DicomImage`” indicates the name of the class. For each module, the file of the same name with an extension “.mod” in the current directory is read. As a module example, the file *ModalityLUT.mod* contains the following:

```
#

# Modality LUT
#
element = 0028 3000
>element = 0028 3002
>element = 0028 3003
>element = 0028 3004
>element = 0028 3006
element = 0028 1052
element = 0028 1053
```

A module file contains a list of element tags. Similar to the notation in the actual DICOM standard, if an element is a “sequence” or container type, then the elements below it are indented with a “>” character. For a sequence within a sequence, the elements are preceded by “>>”, and so on.

For each top-level element, for example 0028, 1052 (rescale intercept) in the Modality LUT module, the following member functions are added to the `DicomImage` class definitions.

```
/**

*  rescaleIntercept()
*  get the Rescale Intercept element value
*  from the DicomImage's data set.
*  dicom tag = (0028, 1052)
*  Throws DCSException if that element has not been set (not in dataset).
*  Throws DCSException if element is not multi-valued.
*  @return the data for this element
*/
const string &rescaleIntercept( void ) const
   throw( DCSException );

/**
*  rescaleIntercept()
*  set a new value for the Rescale Intercept element.
*  dicom tag = (0028, 1052)
*  A copy of the input string will be made by the object. (using string's copy
*  on write logic).
*  @param data - string value
*  @return nothing
*/
void rescaleIntercept( const string &data );
```

For elements that are sequences, a new class is generated to represent the sequence. Member functions to get or set that sequence are also added to the containing class. For example, for the element 0028, 3000 (modality LUT sequence) in the Modality LUT module, the following member functions are added to the `DicomImage` class definitions:

```
/**

*  getModalityLutSequenceCount()
*  return the number of items (data sets) in the
*  ModalityLutSequence sequence
*  dicom tag = (0028, 3000)
*  @return the number of items in the given sequence or -1 if the
*  element is not present in the data set
*/
INT32 getModalityLutSequenceCount() const;

/**
*  modalityLutSequence()
*  get the requested item (data set) from the
*  Modality LUT Sequence sequence
*  in the DicomImage's data set.
*  dicom tag = (0028, 3000)
*  The default item index is 0.
*  The returned DicomDataSet can be assigned directly to
*  a ModalityLutSequence object (which will make a copy of that data set).
*
*  The data that is returned is still owned by this object
*  and may not be deleted.
*  If this object goes out of scope, or is otherwise deleted, the pointer
*  will become invalid.
*
*  @param n the index of the sequence item (data set)to retrieve
*  @return pointer to data set at the given index in the sequence, or NULL
*  if the requested index is zero, and the sequence element exists, but is
*  zero length.
*  Throws DCSException if that element has not been set (not in dataset).
*  Throws DCSException if the requested item is not present in the sequence
*  (excepting case of requesting item 0, for a zero length sequence)
*
*/
DicomDataSet* modalityLutSequence( INT32 n=0 ) const
   throw( DCSException );

/**
```

```
*  modalityLutSequence()
*  set a new value for the Modality LUT Sequence element.
*  dicom tag = (0028, 3000)
*  overwrites any existing element with tag E_MODALITY_LUT_SEQUENCE
*
*  The sequence is created with one data set which is copied from
*  the argument's data.
*  @param data the ModalityLutSequence object from which the sequence data set will
*  be copied.
*  @return nothing
*/
void modalityLutSequence( ModalityLutSequence& data );

/**
*  modalityLutSequence()
*  set a new value for the Modality LUT Sequence element.
*  dicom tag = (0028, 3000)
*  overwrites any existing element with tag E_MODALITY_LUT_SEQUENCE
*
*  The sequence is created with the data sets which are copied from
*  each element of the argument vector.
*  @param data the vector  of ModalityLutSequence objects from which the sequence
*  data set(s) will be copied.
*  @return nothing
*/
void modalityLutSequence( std::vector<ModalityLutSequence>& data );
```

See the program *`$DCF_ROOT/devel/csrc/examples/ex_iod`* for an example of using this generated class to access fields in a DICOM image.

Note that you do not need to use IOD wrapper objects, since `DicomDataSet`, `DicomElement`, `DicomSequence`, and other classes can be used directly, which in many cases may be a more convenient approach.

9. Configuring DCF Applications

Applications using the DCF can be designed to run with no externally provided configuration files or data. All settings can be controlled at runtime using available API's. Often however, the quickest way to produce commercial quality applications is to leverage the configuration data architecture available to the DCF. To complicate matters further, various combinations of "handle it all yourself" and "use DCF configuration facilities and data files" can be built.

9.1. Configuration Files and the CDS interface

DCF C++, Java and C# applications use the CDS (Configuration Data Service) component to read and write configuration data. CDS defines several key classes or interfaces: `CFGAttribute` – essentially a name/value string pair; `CFGGroup` – a named collection of `CFGAttribute` and nested `CFGGroup` objects; and `CFGDB` – which provides methods to load and store data to persistent storage. The standard implementation of the `CDS::CFGDB` interface stores data in text files, either directly or via the *`DCDS_Server`* application (Distributed CDS Server). *`DCDS_Server`* also stores data in text files, but provides higher-level functionality such as multi-process safe data access, observer notifications when objects of interest are changed, and the ability to see the repository as a simple hierarchy of addressable groups and attributes. The application NDCDS_Server provides configuration DB server capabilities for C#.Net clients.

The following is an example configuration file:

```
#  File comment

# group comment
[ group_name ]
# attribute comment
attrname = value
multival_attr_name = val_1
multival_attr_name = val_2
multival_attr_name = val_3
multi_line_attr = aaaaaaaaaaaaaaaaaaaaaaaaa \
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb\
ccccccccccccccccccccccccccccccccccccccccccc

[ group_name/sub_group_name ]
attr = value
```

Notice that there is no `CFGFile` object. A file is represented as a `CFGGroup`. If the *`DCDS_Server`* is used, file system directories are also represented as `CFGGroup` objects.

The full name of any `CFGGroup` or `CFGAttribute` includes the names of all of its parent objects, up to some root, separated with the "/" character.

The *`DCDS_Service`* allows an arbitrary collection of directories and configuration files to appear as a single hierarchy of `CFGGroup` and `CFGAttribute` objects. From the programmers' perspective there is a simple hierarchical object database. Since the persistent storage for this data is simply directories and text files, maintenance procedures are simple and flexible.

An alternate implementation might use some other persistent representation (e.g., XML files, SQL DB tables, Windows Registry, etc). As long as the CFGDB interface can provide the notion of a hierarchy of groups and attributes, the underlying format is unimportant.

Configuration files or `CFGGroup` objects are used throughout the DCF. Configuration data is normally stored under the directory indicated by the DCF_CFG environment variable (usually

$DCF_ROOT/cfg*). A configuration group name `/apps/defaults/dcf_store_scu` indicates the file *$DCF_CFG/apps/defaults/dcf_store_scu. To describe a file that is outside of the DCF_CFG directory (and/or that will not be handled by the DCDS_Server), add the "file:/" prefix to the name. For example:

```
dcf_pg -f file://tmp/image.cfg
```

indicates the file */tmp/image.cfg*.

```
dcf_pg -f file:/image.cfg
```

indicates the file *./image.cfg*.

```
dcf_filter -f file:/C:/temp/filter1.cfg
```

indicates the file C:\temp\filter1.cfg.

Perl scripts can access CDS data using the *CFG*.pm* modules located in *$DCF_ROOT/lib/perl5*.

9.1.1. Using cds_client to access data in the configuration database

If the *DCDS_Server* is running, *cds_client* can be used to read or write from the configuration database. For example:

```
cds_client loadgroup /procs
```

will list all server processes that have saved their process configuration to the `CFGDB`.

```
cds_client loadgroup /components/cpp_lib/DCS/debug_controls
```

will list all defined debug settings for the DCS C++ library component.

```
cds_client saveattr
/apps/defaults/dcf_store_scp/DCS/association_manager/tcp_port 104
```

will change the port that will be used to accept DICOM associations by the store SCP application.

Note: Use ncds_client.exe to interact with the C# NDCDS_Server.

9.1.2. Receiving notifications of updated data

If the *DCDS_Server* is running, applications may be notified when data is changed. This allows an application to react to changes in the data. For example, most DCF applications "listen" to their own `debug_flags` attribute in their process configuration; this allows them to change dynamically the amount of debugging information that is output when a user (or another application) changes the value of the `debug_flags` attribute.

In order to be notified when an object changes, a process must register as an observer of the object. (Note that a process can be an observer of several objects, not just one, and that multiple processes may be observing the same object.) This is done via the `CFGDB` method `register()`. When the object changes, the process's `notify()` method is called with the name of the object that changed. The process may then read the updated value and proceed accordingly.

If the process is observing changes in a `CFGAttribute`, it will be notified when that attribute changes. Processes may also observe changes in `CFGGroup`s. In this case, they will be notified if a change occurs in the group or in any of the group's sub-groups and attributes, including in sub-groups of those sub-groups, and so on.

For example, suppose we have the following `CFGGroup`:

```
[ tmp ]
[ tmp/status ]
device_mno_status = ERROR

[ tmp/status/status_info ]
device_xyz_status = NORMAL
device_abc_status = NORMAL
```

The C++ example *ex_notify* listens to changes in the CFGGroup */tmp/status/status_info*. If the value of the attribute device_abc_status in that CFGGroup is changed to "ERROR", *ex_notify* will be alerted to the change and can react to the change (in this case, *ex_notify* will display the new values of the attributes in the group). If another process is registered as an observer of the attribute device_abc_status, it will also be notified of the change in value. It will not be notified, however, if the value of device_xyz_status changes. If the CFGGroup */tmp/status* is changed, *ex_notify* will *not* be notified of the change unless that change affects the status_info sub-group – for example, deleting the */tmp/status* group will cause *ex_notify* to be notified; changing the value of device_mno_status will *not* cause *ex_notify* to be notified. Adding a new attribute device_pqr_status to the status_info group would cause any listeners to the tmp, status, and status_info groups to be notified – this includes the *ex_notify* example.

9.2. Application and Process Configurations

The APC::AppControl interface uses the CDS to manage two special configuration objects (CFGGroups). The application configuration (app config for short) provides initial settings for the program and is optional. The process configuration ("proc config" for short) reflects the current settings for the program. (Note: the process configuration was called "application instance configuration" in earlier DCF releases.) At runtime, most configuration data is read from the Process configuration. Changes made to the application configuration normally take effect the next time the application is run.

(Note that in C#, the AppControl and CFGDB common service interfaces are contained in the LaurelBridge.DCF assembly.)

Log/Debug tracing control is a special case of using application or process configuration data.

9.2.1. Application Configuration Settings

The application configuration contains the initial settings for a program that is run. The same program may be invoked multiple times with different application configurations. A program may be run with no application configuration.

Each LBS DCF program has a default application configuration. These are typically found in the group /apps/defaults. The application configuration used for a program can be specified on the command line using the -appcfg option, e.g.,

```
-appcfg /apps/defaults/some_cfg
```

or

```
-appcfg file:/C:/temp/my_custom_cfg
```

Application configurations can be created by hand, or automatically generated by *dcfmake.pl*. The *genappcfg.pl* utility can generate an application configuration by specifying changes to the default or any other existing application configuration.

Note that some programs may require additional configuration data. For example, ex_nprint_client or ex_nstore_client can take a job configuration on their command line.

9.2.1.1. Structure of an application or process configuration

The application or process configuration contains one sub-group for each library component that is used by the application and one sub-group for the application itself. Configuration files follow the same format for each supported language. For example, a C# application "`my_scp_program`" that uses the C# `ABC`, `DEF`, and `XYZ` libraries will contain the following groups:

```
[ application_info ]
name = my_scp_program
description = example application
app_component_name = cs_app/my_scp_program
execution_state = STOPPED

[ required_components ]
component = cs_lib/ABC
component = cs_lib/DEF
component = cs_lib/XYZ

[ cs_app ]
[ cs_app/my_scp_program ]
debug_flags = 0
my_cfg_1 = xxxxxx
my_cfg_2 = xxxxxx

[ cs_lib ]
[ cs_lib/ABC ]
debug_flags = 0
abc_cfg_1 = xxxxxx

[ cs_lib/DEF ]
debug_flags = 0
def_cfg_1 = xxxxxx

[ cs_lib/XYZ ]
debug_flags = 0
xyz_cfg_1 = xxxxxx
```

9.3. Process Configuration Settings

The process configuration contains the current settings for a program that is running.

9.3.1. Process configuration with AppControl setup

If AppControl_a.setup() is invoked to initialize the application control component, an application configuration is required. The process configuration is created initially from a copy of the application configuration. After creating the initial process configuration, it may be modified by command line arguments that were passed to the AppControl_a.setup() method. For example:

```
-apc "/cs_lib/DCS/AssociationManager/server_tcp_port=1234"
```

Will override the value for the attribute `server_tcp_port` in the group `cs_lib/DCS/AssociationManager`.

The process configuration can optionally be written to the CFGDB data base. The default CFGDB name for this will be "`/procs/<base-name-of-program-exe>.<process-id>`", e.g.,

```
"/procs/dcf_print_scp.4355"
```

The proc cfg name can be specified on the command line using the -proccfg option, e.g.,

```
-proccfg /procs/dcf_store_scp.001
```

9.3.1.1. Monitoring the Process Configuration

An external application can monitor the process configuration in the `CFGDB` for changes that are made as the application runs. Likewise an application can opt to be notified if the process configuration data in `CFGDB` is changed. If this mode is selected, `AppControl` automatically reloads the process configuration data before sending notifications to user code. By using this technique, applications can easily support reconfiguration without the need to shutdown and restart. (See Section 9.1.2 below for more information.)

9.3.2. Process configuration without AppControl setup

If no application configuration is available, the process configuration will be constructed dynamically as sub groups are referenced.

If the application configuration does not exist, data for the process configuration will be read from the /components group in the CDS repository. For example, if library code asks AppControl for the group "java_lib/DCS" and that group is not present in the proc config, the group "DCS" in the file components/java_lib/DCS will be loaded. If that data is not available, then compiled-in data from the component's (assembly/package/dll) CINFO class is loaded.

9.3.3. Creating a custom application configuration

Every DCF application is given an application configuration at runtime. This configuration contains data used by the application component, and each library component linked by the application. By default the generated configuration apps/defaults/<application_name> is used. This can be overridden with the –appcfg <cfgname> command line option, which is processed by the `APC_a::AppControl_a` class. The alternate configuration can be a hand modified copy of the default, or can be automatically generated by making modifications to another configuration.

Custom Application Config files are stored in the *`$DCF_ROOT/devel/cfgsrc`* directory. If the file has the extension .cac, then it is processed by the *`genappcfg.pl`* utility. A ".cac" file is in the form of other configuration files, however certain attributes contain instructions to the genappcfg.pl program. The *`update_cds.pl`* script will invoke *`genappcfg.pl`* as needed to created custom application config files when the CDS repository is being updated.

The following example describes the process of creating a custom application configuration from a default application configuration.

Here is a .cac file example. The file is named:

```
$DCF_ROOT/devel/cfgsrc/apps/PrintSCP/PrintSCP1.cac
```

```
 [ config_control ]
base = /apps/defaults/print_server
target = /apps/PrintSCP/PrintSCP1
remove_group = DCS/supported_sop_classes

[ DCS ]

[ DCS/scp_options ]
timeout = 30


[ DCS/supported_sop_classes ]
sopclass = 1.2.3.4
sopclass = 1.2.3.5

[ LOG_a ]
+output = /tmp/additional_log_output
```

The `config_control` group contains only instructions to the *`genappcfg.pl`* program.

`base` defines the starting configuration. We load a copy of that configuration from the file system, relative to *`$CFGGENDIR`*. This becomes the working configuration.

`target` is the output or destination configuration that also will be written relative to $CFGGENDIR.

`remove_group` means that the group having the given name in the working configuration is to be deleted. A likely reason to remove a group is so that it can be entirely replaced with contents from this file.

After any groups have been removed from the working configuration, the remainder of the custom app config is processed.

For each additional group, if that group does not exist in the working configuration, it is created.

For each attribute in the custom app config group, one of two things will happen.

- If the attribute name begins with '+', then the attribute having the same name in the working config is located. The values in the custom app config are then added to the existing attribute's values. If no such attribute exists, it is created with only the new values.
- If an attribute name does not start with '+' then any attribute in the working config with that name is first deleted, before the new attribute is added.

If a group is being modified, then the comments from the base configuration for that group will remain in the target configuration.

If a group is being added (or replaced), the comments from the custom app config will accompany the group.

Attributes or attribute values that are added or replaced are always accompanied by the corresponding comments from the custom app config

Using the following as the base configuration,

```
[ DCS ]
debug_flags = 0

[ DCS/scp_options ]
port = 3004
timeout = 10
max_concurrent_associations = 8


[ DCS/supported_sop_classes ]
sopclass = 1.2.3.4
sopclass = 1.2.3.5
sopclass = 1.2.3.6
sopclass = 1.2.3.7

[ LOG_a ]
debug_flags = 0
output = /tmp/logfile
```

the resulting output would be:

```
[ DCS ]
debug_flags = 0

[ DCS/scp_options ]
port = 3004
timeout = 30
max_concurrent_associations = 8


[ DCS/supported_sop_classes ]
sopclass = 1.2.3.4
sopclass = 1.2.3.5

[ LOG_a ]
debug_flags = 0
output = /tmp/logfile
output = /tmp/additional_log_output
```

The group `DCS/supported_sop_classes` which originally contained 4 attribute values was removed, and replaced with the group containing 2 values.

The attribute `timeout` in the group `DCS/scp_options` was removed and replaced.

The attribute `output` in the group `LOG_a` was modified by adding a value. The original value was preserved.

9.4. Log/Debug tracing control using "`debug_flags`"

Each DCF library or application component in a running process maintains a current "`debug_flags`" setting. As code in a given component is executed, the bits that are set in debug_flags determine which messages get logged. Other behavior may also be controlled by `debug_flags`.

The `debug_flags` for each application or library component are initialized from the current process configuration. This means of course that their initial value comes from the application configuration. If the `CFDB` is configured in server mode, those `debug_flags` attributes in the process configuration are monitored for changes. If another application changes the `debug_flags` in the process configuration, the DCF is notified and the new setting can be retrieved from the `CFGDB`.

API's exist that allow programmers to directly get or set debug flags for components in their process. Also service tools that provide convenient access to this special class of configuration data exist.

As applications become larger, and multiple developers create complex sub-systems, it is useful to distinguish log/debug settings for each of those sub-systems. The DCF per-component `debug_flags` provide an effective way to do this.

In practice, users may find that only certain debug flags are of interest to them. For instance, the DIMSE and PDU logging settings controlled by the "`cs_lib/DCS/debug_flags`" attribute.

Custom user interfaces can easily be designed to manipulate only those debug settings appropriate for a particular use case.

Refer to DCF.ComponentInfo constructor documentation as a reference.

9.4.1. Example – Setting Debug Flags for an Example App

Suppose you want to enable some debug flags for the java_lib/DCS component/package to trace network communications for the ex_jqr_scu example application..

Depending on how you are set up, there are a few ways to set debug flags.

1. If you have a configuration file for the app.
   In this case you would look at `$DCF_CFG/apps/defaults/ex_jqr_scu`, then you could set the attribute "`debug_flags`" under the section "`[ java_lib/DCS ]`" to the value `0x360000`.

   For example, you would change the default text show below to match the suggestion above:

```
#==========================================================================
# per-instance information for the DCS component
#==========================================================================
[ java_lib/DCS ]
# turn on df_SHOW_WARNINGS by default
debug_flags = 0x00040
```

2. Setting the operating system environment.
   You can set that same debug flag in the environment that for the app. In Linux, at a command prompt you would say:

   `$ export java_lib_DCS_DF=0x360000`

3. Programatically.
   In your Java code, you can set debug flags by saying:

```
com.lbs.DCS.CINFO.instance().setDebugFlags(
      com.lbs.DCS.CINFO.df_DUMP_ACSE
      | com.lbs.DCS.CINFO.df_DUMP_PDATA
      | com.lbs.DCS.CINFO.df_SHOW_DIMSE_READ
      | com.lbs.DCS.CINFO.df_SHOW_DIMSE_WRITE );
```

4. Using the DCF Operations Web Interface.
   If you want to know how to configure application debugging via the DCF Operations web interface, refer to Section 2.4.3.8, Set Debug Flags. Basically, this convenience interface allows a knowledgeable user to navigate to the config file of interest and set the debug flags of interest by clicking a checkbox.

9.4.2. Defined Debug Flags

All of the debug flags for a given component are defined in the `CINFO` class in the corresponding package.

One place to see that list of definitions is in the generated configuration data for a package that has a `cinfo.cfg` and that was processed with *`dcfmake.pl.`* The list for an app is found in the appropriate `$DCF_CFG/components` sub-directory.

For example, the file `$DCF_CFG/components/java_lib/DCS` shows each of the debug settings defined for the DCS component.

Show below is the default list for this example:

```
[ debug_controls ]
debug_flag = df_SHOW_CONSTRUCTORS,   0x0001, show object constructors
debug_flag = df_SHOW_DESTRUCTORS,    0x0002, show object destructors
debug_flag = df_SHOW_GENERAL_FLOW,    0x0004, show general flow
debug_flag = df_SIMULATE_HARDWARE,    0x0008, simulate external devices
debug_flag = df_SHOW_CFG_INFO,        0x0010, show configuration information
debug_flag = df_SHOW_EXC_THROW,       0x0020, show exceptions before throwing
debug_flag = df_SHOW_WARNINGS,        0x0040, show warning message text
debug_flag = df_DUMP_ACSE,          0x020000, show ACSE pdu data
debug_flag = df_DUMP_PDATA,         0x040000, show PDATA pdu summary
debug_flag = df_DUMP_PDATA_VERBOSE, 0x080000, show PDATA pdu data (can be very verbose)
debug_flag = df_SHOW_DIMSE_READ,    0x100000, show DIMSE message reads
debug_flag = df_SHOW_DIMSE_WRITE,   0x200000, show DIMSE message writes
debug_flag = df_TCP_NETWORK,        0x400000, show tcp/ip network related debugging
debug_flag = df_COMPRESSION,        0x800000, show compression transfer syntax codec
                                              debugging
debug_flag = df_VERBOSE,           0x1000000, show various verbose debug messages
debug_flag = df_IDLE_TIMERS,       0x2000000, trace association idle timeout logic
debug_flag = df_FILTER_SUMMARY,    0x4000000, show summary of filters applied
debug_flag = df_FILTERS,           0x8000000, trace data-set/dimse-message filtering
debug_flag = df_VERBOSE_DICOM_IO, 0x10000000, show detailed information while reading or
                                              writing data sets
debug_flag = df_REJECT_TRANSIENT, 0x20000000, Force association rejection with transient
                                              status for testing
debug_flag = df_REJECT_PERMANENT, 0x40000000, Force association rejection with permanent
                                              status for testing
```

9.5. C#-related Configuration Notes

9.5.1. Description of DCF setup code

This block is copied from the example *`ex_nmwl_scp`*’s `main()` method:

```
LaurelBridge.CDS_a.CFGDB_a.setup( args );

LaurelBridge.APC_a.AppControl_a.setup(args, CINFO.Instance );

LaurelBridge.LOG_a.LOGClient_a.setup();
```

Each of these commands is explained further below. For the method calls which have an `args` argument, `args` is a string array. For a command line program you would pass through the string array passed into main from the command line to pass in parameters to custom implementations of our common services. If you do not require any special parameters you can pass in some bogus array, e.g.,

```
string[] args = new string[0];
args[0] = stuff;
```

9.5.2. Common services setup description:

1. `LaurelBridge.CDS_a.CFGDB_a.setup( args )`

This call sets up the configuration database via .NET Remoting to the *`NDCDS_Server`* `CFGDB` Server. The *`NDCDS_Server`* by default uses a filesystem database.

2. `LaurelBridge.APC_a.AppControl_a.setup(args, CINFO.Instance )`

This call sets up the Application Control component. `AppControl` manages and provides an API to accessing an application’s configuration data at runtime, by default through the configuration database. The second argument `CINFO.Instance` sets the application’s name and checks that any required components for this application are located.

3. `LaurelBridge.LOG_a.LOGClient_a.setup();`

This call sets up the reference C# implementation of our `LOG` interface. `LOG` writes log files to text files. It can also “rotate” log files based on size. You can create your own implementation of the `LOG` interface to control logging, e.g., write to the event log, etc. In your app you would replace this call line with something like:

```
OEMname.LOG_a.SystemLogger.setup();
```

Usually you at least want the Configuration Database (`CFGDB`) server running. You can launch it by running *`ndcds_server`* from a command line. You could remove this requirement by adding the line:

```
LaurelBridge.CDS_a.CFGDB_a.FSysMode = true;
```

before the line

```
LaurelBridge.CDS_a.CFGDB_a.setup( args )
```

However, by making this choice, you will lose the ability to change logging levels and other instrumentation while the process is running. You will have to restart the process to make changes to these settings, which may not be practical in a production environment.

10. Configuring DICOM features

10.1. Java and C# DICOM configuration

In addition to application or process global configuration settings, Java and C# applications can configure many functions on a per association basis.

An application that is running may have multiple active DICOM associations or, more generally, multiple I/O sessions. It is useful to allow these to be configured independently. An example of an I/O session that is not a DICOM association is an image viewer application that is reading a DICOM file from mass storage. Settings which may vary from one DICOM association or I/O session to another are contained in the `DicomSessionSettings` object.

The `DicomSessionSettings` class is a convenient container for settings used by numerous DCF I/O related classes.

10.1.1. Example Session settings

Session settings are normally established at the beginning of each association or I/O session and are used throughout the life of that session. There are various mechanisms via the DCF API's that developers can provide their own choices for session settings. In the simplest case, default values are read from the process configuration group:

`cs_lib/DCS/default_session_cfg`.

(Note: Configuration files are the same for all languages; "cs_lib" indicates a C# library component, and "DCS" is "DICOM Core Services", which is where all low level DICOM I/O support classes live. For java applications the defaults are in java_lib/DCS/default_session_cfg)

For example, for the C# store SCP example program, the file:

```
%DCF_CFG%\apps\defaults\ex_nstore_scp
```

contains the default session settings. Comments in the sample configuration shown below explain many of the configuration parameters.

```
[ cs_lib/DCS/default_session_cfg ]
#
# Per session debug_flags. Currently, a subset of the DCS library
# debug flags can be controlled on a per-session basis.
#
# This can be very useful if you wish to enable verbose logging
# when only certain applications connect.
#
# df_SHOW_DIMSE_READ
# df_SHOW_DIMSE_WRITE
# df_SHOW_ACSE_PDU
# df_SHOW_PDATA_PDU_SUMMARY
# df_SHOW_PDATA_PDU_DATA
#
debug_flags = 0

#
# The maximum PDU size we wish to read. This is communicated to the
# peer device during association setup.
# (Network sessions only)
max_read_pdu_size = 32768

#
# The maximum PDU size we will write.
# (Network sessions only)
#
max_write_pdu_size = 32768

#
# For testing, max-length-negotiation can be ignored, allowing
# apps to write larger PDU’s than the remote AE expects.
# (Network sessions only)
#
ignore_max_length_negotiation = NO

#
# The number of seconds DCF will wait for a single PDU to be
# written to a socket. -1 means wait forever. Note other timeouts
# may prevent an application from hanging.
# (Network sessions only)
#
pdu_write_timeout_seconds = -1

#
# If true, then pixel data is not fully buffered. Instead, blocks of
# data are read from the source and written to the destination of
# a transfer as needed. This enables very large data sets to be
# transferred without allocating large amounts of main memory.
# Disabling this provides simpler access to applications that
# ultimately want all of the pixel data in memory.
#
enable_streaming_mode = YES

#
# Size in bytes of buffers that are used for streaming mode transfers.
#
stream_mode_buffer_size = 16384

#
# Number of seconds that DicomSocket will delay before writing each
# outbound PDU. This can be useful for testing, to force a timeout condition,
# or to otherwise slow down data flow.
# (Network sessions only)
#
pdu_write_delay_seconds = 0
#
# Number of seconds that DicomSocket will delay before returning each
# inbound PDU. This can be useful for testing, to force a timeout condition,
# or to otherwise slow down data flow.
# (Network sessions only)
#
pdu_read_delay_seconds = 0

#
# debug support for forcing delays between PDU fragments
# (Network sessions only)
#
socket_write_delay_seconds = 0

# debug support for breaking PDU writes into multiple fragments
# (Network sessions only)
max_bytes_per_socket_write = 0
# name of extended data dictionary config group
extended_data_dictionary = /dicom/ext_data_dict
```

```
#
# Set flag to true if sequences should always be output with
# undefined length.
#
always_write_undef_length_seqs = YES

#
# Set flag to true if sequence items should always be output
# with undefined length.
#
always_write_undef_length_seq_items = YES

#
# Debug support for forcing end delimiters even for fixed length
# sequences. (some implementations incorrectly do this, and may expect it).
#
always_write_seq_end_delims = NO

#
# Debug support for forcing end item delimiters even for fixed length
# sequence items. (some implementations incorrectly do this, and may expect
it).
#
always_write_seq_item_end_delims = NO

#
# Number of seconds that AssociationAcceptor will wait for a message
# before sending an A-Abort-PDU to the Requester and ending the
# association. -1 means wait forever.
#
association_idle_timeout_seconds = 600

#
# cmd line of program to be run at start of association
#
# The following variables will be added to the environment
# to be inherited by both the pre and post association scripts:
#
# CALLING_AE_TITLE                 (client DICOM AE name)
# CALLED_AE_TITLE                  (server DICOM AE name)
# CALLING_PRESENTATION_ADDRESS     (client network addr from AssociationInfo)
# CALLED_PRESENTATION_ADDRESS      (server network addr from AssociationInfo)
# DCF_ROOT                         (value of env var if any)
# DCF_USER_ROOT                    (value of env var if any)
# DCF_CFG                          (value of env var if any)
# DCF_TMP                          (value of env var if any)
# DCF_LOG                          (value of env var if any)
# PROC_CFG_NAME                    (name of proc configuration object if any)
#
pre_association_script =

#
# cmd line of program to be run at end of association
#
post_association_script =

#
# If set, we will not send out multiple pdv’s within a
# single pdu. Some implementations can not handle pdu’s
# containing multiple pdv’s.
#
disable_multi_pdv_pdus = YES

#
# Name of input filter configuration group or file.
# For example “file:/C:/temp/some_filter_set.cfg”
# or “/dicom/filter_sets/example_set1” (which will be located
# in CDS CFGDB or beneath the $DCF_CFG directory.)
# Incoming dimse messages are processed by input filters.
#
# If this value is set, then the input_filters sub-group
# shown below is ignored.
#
input_filter_cfg_name =

#
# Name of output filter configuration group or file.
# Outgoing dimse messages are processed by output filters.
#
# If this value is set, then the output_filters sub-group
# shown below is ignored.
#
output_filter_cfg_name =

#
# Input filter configuration. Add one sub-group per filter.
#
[ DCS/default_session_cfg/input_filters ]

#
# Output filter configuration. Add one sub-group per filter.
#
[ DCS/default_session_cfg/output_filters ]

#
# Transfer syntaxes that will be accepted by an SCP.
# The first transfer_syntax in this group that also exists in the
# proposed list will be selected.
#
[ DCS/default_session_cfg/supported_transfer_syntaxes ]
# implicit-little-endian
transfer_syntax = 1.2.840.10008.1.2
# explicit-little-endian
transfer_syntax = 1.2.840.10008.1.2.1
# explicit-big-endian
transfer_syntax = 1.2.840.10008.1.2.2
```

10.2. C++ DICOM configuration

There are a multiple options available for how a SCP selects a configuration for a new connection.

1. Have your SCP implement the `LBS::DCS::AssociationCfgPolicyManager` interface
   Your SCP picks the configuration for each incoming association when asked via the `selectConfigurationName()` method. You do this by creating some object that inherits from `LBS::DCS::AssociationConfigPolicyManager` and implements the virtual function `selectConfigurationName()`. You inform the `AssociationManager` of your desire to control this part of the connection setup by invoking the `LBS::DCS::AssociationManager::registerConfigPolicyManager()` method

   (which is currently the method used by all DCF SCP applications).

   All of those servers implement `AssociationCfgPolicyManager` similarly:

   ```
   config-name-for-this-association = <value of cfg_name_base attribute> + "/" +
   <called-ae-title>
   ```

   The `DCS/AssociationManager/default_association_config_name` attribute is not used.

2. Let `AssociationManager` select a configuration for each new association.
   AssociationManager uses the config attribute `DCS/AssociationManager/default_association_config_name` to select a configuration for each new association.

   a) Do not set `default_association_config_name`
   The new association will use the same configuration that the server is using. If you are happy with a single set of settings for some SCP, this is the simplest option. Whatever configuration is given to the server at startup (or /apps/defaults/<server_name>) will be used for the daemon, and all associations.

   This is the default in the DCS component configuration, which will be included in the generated application configuration for any application that uses DCS.

   b) Set `default_association_config_name` to something
   The new association will use the specified configuration, after variables are expanded.

   By allowing certain macros to be expanded in this string, the same behavior that previously required a custom `AssociationConfigPolicyManager` can often be provided by the default implementation. For example, the `AssociationConfigPolicyManager` code could be removed from print_server, and the following attribute added to the configuration for that program:

   ```
   default_association_config_name = /apps/PrintSCP/${CALLED_AE_TITLE}
   ```

The available macros are explained in the config file comment:

```
[ DCS/AssociationManager]
... text removed ...
#
# If no other AssociationConfigPolicyHandler is installed, this string
# will be used to generate the configuration name for an incoming association.
#
# In addition to the DCF_VAR and DCF_FUNC text expansions that may occur during
# the update_cds process, the following macros will be expanded after the
# A-Associate-Request PDU is received from the SCU:
#
# MACRO             EXPANDS TO
# ===============   =============================
```

```
# ${CALLED_TITLE}  called ae title from pdu
# ${CALLING_TITLE} calling ae title from pdu
# ${CALLING_HOST}  remote device's host address
# ${CALLED_HOST}   local host address for connected socket
# ${CALLED_PORT}   local port number for connected socket
#
# If the string is empty or this parameter does not exist, then the
# new association will use the configuration of the parent server.
#
default_association_config_name =

# set to true if config data should be cached in SCP between associations
cache_association_configurations = true
```

10.3. Customizing DicomDataDictionary

The `DicomDataDictionary` class provides lookup services for DICOM attribute tags. `DicomDataDictionary` also provides the UID (unique identifier) factory service. Both of these services may be customized by the OEM.

There are two ways to add support for private DICOM tags, or to override the definitions for standard tags. The easiest is to add a custom data dictionary configuration file. Any application that uses `DicomDataDictionary` (e.g., all standard DCF apps), will now recognize the tags that are defined in that file. By default, each application will look for a file named *$DCF_CFG/dicom/ext_data_dictionary*. If this file exists, it is used to extend the standard data dictionary. The name of the extended dictionary file can be changed for any application. An example of the text in that file is:

```
#
# The following is an example extended data dictionary file.
# If this file is named "ext_data_dictionary", it will be
# used whenever DCF libraries or applications look up information
# for a DICOM tag. The name of the extended data dictionary file
# is contained in the [ DCS] configuration group of DICOM related
# applications.
#
[ elements ]
0029,0010 = CS,1,Example Private Attribute 1
0039,0020 = US,1,Another Example Private Attribute 2
0049,1001 = DS,1,Private DS attribute 3
0049,1002 = UL,1,Private UL attribute 4
0049,1003 = SL,1,Private SL attribute 5
0049,1004 = UI,1,Private UI attribute 6
```

Alternately, the `DicomDataDictionary` class can be extended programmatically. `DicomDataDictionary` uses the singleton pattern. Only one instance of that class exists in a process. If the OEM creates a subclass of `DicomDataDictionary`, it will become the singleton instance, and all DCF code will use it. See the online documentation for `DicomDataDictionary` for more information about this.

To change the base for UIDs that are generated by the DCF system edit the file: $*DCF_CFG/dicom/uid*. An example of the text in that file is:

```
[ oem_info ]
uid_prefix = 1.2.3.4
uid_system_prefix = 192.131.14.4
```

The configuration attribute `uid_prefix` can be used to provide an organization wide prefix for UIDs, containing perhaps your organization's ANSI identifier. If this attribute is not present, the DCF will establish the prefix.

The attribute `uid_system_prefix` can be used to provide a per-host portion of each UID. If this attribute is not present, the DCF will use a string based on the IP address of the system.

NOTE THAT IF YOU ARE USING PRIVATE IP ADDRESSES – e.g., 192.168.0. – THE DEFAULT IMPLEMENTATION MAY RESULT IN NON-UNIQUE UIDS.*

When the `DicomDataDictionary::makeUID()` method is called, a uid is produced by combining the results of `DicomDataDictionary::getUIDPrefix()` and `DicomDataDictionary::getUIDSuffix()`. `GetUIDPrefix()` normally returns the concatenation of the `uid_prefix` and `uid_system_prefix` strings. `GetUIDSuffix()` normally generates a suffix based on an incrementing sequence number, combined with the current process ID.

By installing a custom `DicomDataDictionary`, any of the functions `getUIDSuffix()`, `getUIDPrefix()` or `initUIDPrefix()` may be overridden by the user if an entirely different UID generation algorithm is desired. For example, there may be a central UID factory implemented as a COM or CORBA server. See the online documentation for `DicomDataDictionary` for more information about this.

See the notes on UID generation in Appendix E: DCF MakeUID Function.

11. DICOM Image Compression

The discussion of the use of JPEG encoding in medical imaging applications is the subject of many other dedicated articles and publications, which should be referred to for additional details. Some references are included below and in Appendix B: Section 3.

In general, careful attention must be given to the appropriate application of compression transfer syntaxes. Some image formats cannot or may not be compressed by certain CODECs. The user has the responsibility to determine the appropriateness of any compression selected; no compression should be indiscriminately applied to all images. For additional details on transfer syntax encoding rules, see the DICOM Standard, PS 3.5, Section 10 and Appendix A.

By default, DCF provides built-in libraries for compression. The Aware, Inc. JPEG library can also be plugged in as an alternate implementation. See Section 11.4 below for details.

11.1. Lossy Compression Quality Issues & Concerns

The user of the lossy compression codecs has an obligation to assure that the choices made for compression parameters result in usable images. Because the selection of lossy compression options depends significantly upon the content of the images being compressed, the toolkit user must assume responsibility for choosing lossy compression parameters appropriate to their data. There are some images that should not be lossy compressed; the toolkit user must also determine the appropriateness of lossy compression for their data – such a choice cannot be left to the codec.

There has been much research and much written on the use of lossy compression for medical images. We recommend that users of lossy compression review the literature, especially as it relates to their image types. One recent publication that summarizes some current thinking is:

CAR Standards for Irreversible Compression in Digital Diagnostic Imaging within Radiology, were published by the Canadian Association of Radiologists in 2008 & revised in 2011 are available for download from:

```
http://www.car.ca/uploads/standards%20guidelines/201106_EN_Standard_Lossy_Compression.pdf
```

Other information sources are listed in Appendix B: Section 3- Sources for Compression Related Information.

11.2. JPEG Encoding Notes

11.2.1. JPEG Lossless (.57, .70) Encoding Notes

By default, DCF uses the C language IJG library for this compression transfer syntax.

Also note that the Aware, Inc. JPEG library can be plugged in as an alternate implementation. See Section 11.4 below for details.

The name “JPEG Lossless, Non-Hierarchical (Process 14)” usually refers to DICOM transfer syntax 1.2.840.10008.1.2.4.57; we say “usually”, because the names used for various JPEG encodings vary, depending on where you read them.

The default transfer syntax commonly used by DICOM for JPEG lossless is 1.2.840.10008.1.2.4.70; it has a similar name, but usually incorporates the additional text “selector value 1”, or “predictor selection 1”.

Both syntaxes use the same JPEG lossless encoding process, but the “.70” specifies the choice for one of the codec parameters (predictor-selection-value = 1), whereas the “.57” does not specify that value (in some implementations, the default for predictor-selection-value in “.57” is set to “6”).

From the DICOM standard in part 5, we see that ".70" is the default lossless syntax:

> *10.2 TRANSFER SYNTAX FOR A DICOM DEFAULT OF LOSSLESS JPEG COMPRESSION*
>
> *DICOM defines a default for lossless JPEG Image Compression, which uses a subset of coding Process 14 with a first-order prediction (Selection Value 1). It is identified by Transfer Syntax UID = "1.2.840.10008.1.2.4.70" and shall be supported by every DICOM implementation that chooses to support one or more of the lossless JPEG compression processes. …*

The application configuration file contains the complete configuration data for the JPEG lossless codec. For example, from *%DCF_ROOT%\cfg\components\cs_lib\DCS*:

```
[ DCS/default_session_cfg/DicomJPEGCodec/jpeg_lossless ]
# If true, then 12 bit operations will use the 16 bit IJG library
no_12bit_lib = true

# Set the jpeg predictor selection value for the .57 syntax.
# If the transfer syntax is 1.2.840.10008.1.2.4.70,
# this attribute is ignored and predictor selection value
# is set to 1.
jpeg_predictor_selection_value = 6

# If true, then derived image fields are added for monochrome
# images. (Some implementations add derived fields, create
# a new sop-instance-uid, etc. even for lossless compressed
# images.)
add_derived_image_fields_for_mono = false

# If true, then derived image fields are added for color
# images. (Some implementations add derived fields, create
# a new sop-instance-uid, etc. even for lossless compressed
# images.)
add_derived_image_fields_for_color = false

# If true, signed pixel data (pixel-representation = 1 ) will
# be allowed.
allow_signed_data = false

# If true, color pixel data will be allowed. Some implementations
# don't implement lossless jpeg for color, since the RGB to YBR
# color space conversion may result in some information loss.
allow_color = true

# For codecs that support creating multiple threads for a single
# compress or decompress operation.
max_threads = 1

# If true, the TSCWIJG codec will scan the jpeg header for the
# encoded bit depth and may override the bit depth defined by DICOM.
prescan_jpeg_header = true

# If true, the header prescan will stop once the start of frame
# tag has been processed.  If false, and df_COMPRESSION is set,
# all jpeg header items will be logged to the log stream.
stop_scanning_after_sof = true

# Sanity check the rows, columns and samples per pixel in the
# jpeg header, and throw an exception if these values are not
# consistent with the values defined by the DICOM header.
check_jpeg_dimensions = true
```

11.2.2. JPEG Lossy (.50, .51) encoding notes

By default, DCF uses the C language IJG library for this compression transfer syntax.

Note that the Aware, Inc. JPEG library can be plugged in as an alternate implementation. See Section 11.4 below for details.

Note that it is only legal to use JPEG.50 to encode 8-bits-allocated/8-bits-stored images, and only legal to use JPEG.51 to encode 16-bits-allocated/12-bits-stored images. Attempting to use these transfer syntaxes to encode images with something other than the appropriate bits allocated value should generate a message stating: "`invalid non 8-bit data for jpeg.50`" or "`invalid non 12-bit data for jpeg.51`". See DICOM Standard, PS 3.5-2011, Section 10 for additional details.

When you compress images to JPEG Baseline Process 1 (1.2.840.10008.1.2.4.50) or any other lossy compression, the result is a new image (being lossy compressed, the result image will not be exactly identical to the source image); a new image requires a new SOP Instance UID, according to the DICOM standard. In such a case, the DCF will automatically create a new SOP Instance UID for the image and add to the dataset any necessary derived fields to show that the image was compressed at some point in time. (One possible result of this is that, as you are testing a DCF-based application by having it compress the same image again and again, you may appear to be getting duplicate images – each new image will have a new SOP Instance UID.)

One way to avoid the creation of new UIDs is to use JPEG .70 compression (1.2.840.10008.1.2.4.70 – JPEG Lossless First Order Prediction). JPEG .70 is a lossless compression – since the resulting *image* will be identical to the source *image*, a new UID is not required.

DCF will automatically change the photometric interpretation of an RGB uncompressed dataset to YBR_FULL_422 for the compressed dataset as specified by DICOM. DCF will automatically change the photometric interpretation of a YBR_FULL uncompressed dataset to YBR_FULL_422 for the compressed dataset as specified by DICOM.

See Section 11.1, Lossy Compression Quality Issues & Concerns for additional comments.

The application configuration file contains the complete configuration data for the JPEG lossy codec. For example, from *`%DCF_ROOT%\cfg\components\cs_lib\DCS`*:

```
[ DCS/default_session_cfg/DicomJPEGCodec/jpeg_lossy ]
# lossy compression quality : 0 to 100
compression_quality = 75

# If true, then 12 bit operations will use the 16 bit IJG library
no_12bit_lib = false

# If true, derived image elements are added to data sets
# as they are written. This includes changing Image-Type,
# and adding Source-Image-Sequence and Derivation-Code-Sequence.
# A new sop-instance-uid will be created for the output data set.
add_derived_image_fields = true

# If true, signed pixel data (pixel-representation = 1 ) will
# be allowed.
allow_signed_data = false

# For codecs that support creating multiple threads for a single
# compress or decompress operation.
max_threads = 1

# If true, the TSCWIJG codec will scan the jpeg header for the
# encoded bit depth and may override the bit depth defined by DICOM.
prescan_jpeg_header = true

# If true, the header prescan will stop once the start of frame
# tag has been processed.  If false, and df_COMPRESSION is set,
# all jpeg header items will be logged to the log stream.
stop_scanning_after_sof = true

# Sanity check the rows, columns and samples per pixel in the
# jpeg header, and throw an exception if these values are not
# consistent with the values defined by the DICOM header.
check_jpeg_dimensions = true
```

11.2.3. JPEG 2000 Lossless (.90) encoding notes

By default, DCF uses the C language JasPer library for this compression transfer syntax.

Also note that the Aware, Inc. JPEG library can be plugged in as an alternate implementation. See Section 11.4 below for details.

DCF will automatically change the photometric interpretation of an RGB uncompressed dataset to YBR_RCT for the compressed dataset as specified by DICOM. An image with photometric interpretation YBR_FULL will be compressed without the JPEG2000 multi-component-transform being applied and the photometric interpretation will be left as YBR_FULL.

The application configuration file contains the complete configuration data for the JPEG 2000 lossless codec. For example, from *%DCF_ROOT%\cfg\components\cs_lib\DCS*:

```
[ DCS/default_session_cfg/DicomJPEGCodec/jpeg2000_lossless ]
# override all jasper options by using this attribute
# use "\" to end lines for a multi-lined attribute value.
# Do not use with Aware plugin
jpeg2000_codec_options =

# If true, then derived image fields are added for monochrome
# images. (Some implementations add derived fields, create
# a new sop-instance-uid, etc. even for lossless compressed
# images.)
add_derived_image_fields_for_mono = false

# If true, then derived image fields are added for color
# images. (Some implementations add derived fields, create
# a new sop-instance-uid, etc. even for lossless compressed
# images.)
add_derived_image_fields_for_color = false

# set max tile width: 0 means full frame size
max_tile_width = 0

# set max tile height: 0 means full frame size
max_tile_height = 1024

# For codecs that support creating multiple threads for a single
# compress or decompress operation.
max_threads = 1
```

11.2.4. JPEG 2000 Lossy (.91) encoding notes

By default, DCF uses the C language JasPer library for this compression transfer syntax.

Also note that the Aware, Inc. JPEG library can be plugged in as an alternate implementation. See Section 11.4 below for details.

When you compress images to JPEG 2000 Lossy (1.2.840.10008.1.2.4.91) or any other lossy compression, the result is a new image (being lossy compressed, the result image will not be exactly identical to the source image); a new image requires a new SOP Instance UID, according to the DICOM standard. In such a case, the DCF will automatically create a new SOP Instance UID for the image and add to the dataset any necessary derived fields to show that the image was compressed at some point in time. (One possible result of this is that, as you are testing a DCF-based application by having it compress the same image again and again, while you may appear to be getting duplicate images – each new image will actually have a new SOP Instance UID.)

One way to avoid the creation of new UIDs is to use JPEG .90 compression (1.2.840.10008.1.2.4.90 – JPEG 2000 Lossless). JPEG .90 is a lossless compression – since the resulting *image* will be identical to the source *image*, a new UID is not required.

DCF will automatically change the photometric interpretation of an RGB uncompressed dataset to YBR_ICT for the compressed dataset as specified by DICOM. An image with photometric interpretation YBR_FULL will be compressed without the JPEG2000 multi-component-transform being applied and the photometric interpretation will be left as YBR_FULL.

Our testing also has shown that, occasionally, for certain 16-bit input images that use very little of the dynamic range available to them, J2K-lossy compression artifacts can arise. These artifacts change as a function of the number of `compression_levels`, but only in cases where the input image uses very little of its dynamic range.

The following block shows an example of the quality related configuration data for the J2K-lossy codec from an application configuration file:

```
[ DCS/default_session_cfg/DicomJPEGCodec/jpeg2000_lossy ]
# override all options by using this attribute
# use "\" to end lines for a multi-lined attribute value
jpeg2000_codec_options =

# Specify the compression ratio
compression_ratio = 15

# Specify the number of jpeg2000 compression levels
compression_levels = 4
```

The number of `compression_levels` corresponds to the number of levels in the dyadic decomposition of the Discrete Wavelet Transform performed within the J2K compression algorithm. You would never want to use “1” as the value for this configuration parameter – our testing has shown unacceptable artifacts whenever 1 was used for the number of `compression_levels`.

We recommend setting the `compression_levels` in the configuration for JPEG Lossy (.91) to “4” when using the default JasPer library implementation, only because we have yet to see the compression show significant “troubles” at that level for any images we have tested. Empirically, a value of “4” has resulted in near-optimal PSNR (Peak Signal-to-Noise Ratio) without compression artifacts over a wide range of images (different SOP classes, different bit depths, etc.). But ultimately, of course, that is no solution to the general problem. If using the Aware, Inc., library, we recommend setting the compression_levels in the configuration for JPEG Lossy (.91) to “3”, because the Aware, Inc., implementation uses this value as the *additional* number of levels, so the actual levels used will be “4”.

See Section 11.1, Lossy Compression Quality Issues & Concerns for additional comments.

The application configuration file contains the complete configuration data for the JPEG 2000 lossy codec. For example, from *`%DCF_ROOT%\cfg\components\cs_lib\DCS`*:

```
[ DCS/default_session_cfg/DicomJPEGCodec/jpeg2000_lossy ]
# override all jasper options by using this attribute
# use "\" to end lines for a multi-lined attribute value.
# Do not use with Aware plugin
jpeg2000_codec_options =

# Specify the compression ratio
compression_ratio = 2

# Specify the number of jpeg2000 compression levels
compression_levels = 4

# If true, derived image elements are added to data sets
# as they are written. This includes changing Image-Type,
# and adding Source-Image-Sequence and Derivation-Code-Sequence.
# A new sop-instance-uid will be created for the output data set.
add_derived_image_fields = true

# set max tile width: 0 means full frame size
max_tile_width = 0

# set max tile height: 0 means full frame size
max_tile_height = 1024

# For codecs that support creating multiple threads for a single
# compress or decompress operation.
max_threads = 1
```

11.3. JPEG Decoding Notes

This section attempts to collect, identify, and explain some of the special cases likely to be encountered when attempting to decode various JPEG or J2K compressed images. This section will likely always be incomplete as there seems to be no end to the number of ways that images can be wrongly compressed.

11.3.1. Photometric Interpretation Problems

When encoding color images, the compression process may involve changing the color-space of the image to achieve better compression results. The DICOM standard defines how the image header should be modified to indicate what type of data is contained in the compressed code stream. When the image is decoded or un-compressed, the photometric interpretation may be changed back to match the original format. Much confusion exists in this area, and vendors often encode incorrect values for the photometric interpretation. Monochrome images (photometric interpretation MONOCHROME1 or MONOCHROME2) aren't typically a problem since no color-space transforms are applied. The DCF will detect and correct many problems in this area, but occasionally visual confirmation may be required to identify a problem, and a filter or a custom transfer syntax codec may be needed to correct an issue identified in this way.

11.3.2. JPEG Lossless (.57, .70) Decoding Notes

DCF will not change the photometric interpretation of a compressed dataset when decompressing it. So, if the photometric interpretation of the compressed dataset is YBR_FULL it will still be YBR_FULL after decompression.

11.3.3. JPEG Lossy (.50, .51) Decoding Notes

DCF will automatically change the photometric interpretation of an YBR_FULL_422 compressed dataset to RGB for the uncompressed dataset as specified by DICOM.

11.3.4. JPEG 2000 Lossless (.90) Decoding Notes

DCF will automatically change the photometric interpretation of an YBR_RCT compressed dataset to RGB for the uncompressed dataset as specified by DICOM. If the photometric interpretation in the compressed dataset is YBR_FULL, it will remain YBR_FULL after decoding.

11.3.5. JPEG 2000 Lossy (.91) Decoding Notes

DCF will automatically change the photometric interpretation of an YBR_ICT compressed dataset to RGB for the uncompressed dataset as specified by DICOM. If the photometric interpretation in the compressed dataset is YBR_FULL, it will remain YBR_FULL after decoding.

11.4. Using Aware, Inc JPEG Compression libraries

DCF based applications can be configured to use the Aware, Inc. JPEG codecs. The Aware, Inc. JPEG libraries must be purchased from Aware, Inc. and are not included with the DCF SDK.

JPEG (.50,.51), JPEG Lossless(.70), JPEG 2000 (.91), and JPEG 2000 Lossless (.90) can be configured separately to use the default JasPer implementation or the Aware, Inc. implementation. The Aware, Inc. JPEG codec (awj2k.dll) must be located in the library search path of your application. The transfer syntax JPEG Lossless (.57) is not supported by Aware, Inc. See section 11.2.1 for an explanation of the differences between (.57) and (.70) transfer syntaxes.

The DCS (DICOM Core Services) component in a DCF based application's configuration file is where this can be configured.

11.4.1. C# configuration

Each JPEG transfer syntax has a configuration group that controls the JPEG library used for that syntax.

```
[ cs_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.50 ]
plugin_name = TSCWIJG
[ cs_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.51 ]
plugin_name = TSCWIJG
```

The "plugin_name" attribute is set to the deault library of TSCWIJG. To use Aware, Inc., change the plugin_name attribute to TSCWAware. In this example DCF will use Aware, Inc. for the .50 transfer syntax and IJG for the .51 transfer syntax.

```
[ cs_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.50 ]
plugin_name = TSCWAware
[ cs_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.51 ]
plugin_name = TSCWIJG
```

11.4.2. Java Configuration

Each JPEG transfer syntax has a configuration group that controls the JPEG library used for that syntax.

```
[ java_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.50 ]
plugin_name = TSCWIJG
[ java_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.51 ]
plugin_name = TSCWIJG
```

The “plugin_name” attribute is set to the deault library of TSCWIJG. To use Aware, Inc., change the “plugin_name” attribute to TSCWAware . In this example DCF will use Aware, Inc. for the .50 transfer syntax and IJG for the .51 transfer syntax.

```
[ java_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.50 ]
plugin_name = TSCWAware
[ java_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.51 ]
plugin_name = TSCWIJG
```

11.4.3. C++ Configuration

Each JPEG transfer syntax has a configuration group that controls the JPEG library used for that syntax.

```
[ cpp_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.50 ]
plugin_name = TSCWIJG
[ cpp_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.51 ]
plugin_name = TSCWIJG
```

The “plugin_name” attribute is set to the deault library of TSCWIJG. To use Aware, Inc., change the “plugin_name” attribute to TSCWAware. In this example DCF will use Aware, Inc. for the .50 transfer syntax and IJG for the .51 transfer syntax.

```
[ cpp_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.50 ]
plugin_name = TSCWAware
[ cpp_lib/DCS/DicomTSCWCodec/plugin_mappings/1.2.840.10008.1.2.4.51 ]
plugin_name = TSCWIJG
```

12. I/O Statistics for Java and C#

DCF components manage various I/O counters during network associations or other activities. These counters are in the `IOStatistics` class.

The following counters are contained in the `IOStatistics` class:

- `bytes_in`
- `bytes_out`
- `PDUs_in`
- `PDUs_out`
- `dimse_messages_in`
- `dimse_messages_out`
- `total_associations`
- `active_associations`
- `dimse_errors_in`
- `dimse_errors_out`

The `IOStatisticsManager` is responsible for providing `IOStatistics` instances within a process, and for making these statistics available to other entities for monitoring, etc.

The default `IOStatisticsManager` provides a single `IOStatistics` object for each process. That object can optionally be written to the `CFGDB` on a periodic (configurable) basis. An observer app can easily be written to monitor the appropriate file or group in the `CFGDB` for changes and can then report the current counter values.

An OEM can easily provide a custom `IOStatisticsManager`; for example, one that uses an alternate persistence mechanism. Also, statistics may be kept on a per-association, per session, or even per system boundary rather than a per-process boundary.

13. Deploying a DCF-based application

13.1. Deployment Guidelines

To deploy an OEM application, an OEM must include various components of the DCF Toolkit. Normally this means the run-time components: dlls, libraries, config files, etc. More details are found below.

The DCF Toolkit is a separate product, sold to developers for the purpose of creating a derived application. An OEM (developer) may not redistribute the entire DCF Toolkit as part of the OEM application installation.

13.2. License Key for a Deployed Application

Every DCF-based application requires a License Key to run. How this works depends, in part, on the terms of the licensing contract between the OEM and Laurel Bridge Software. There are two basic situations that may occur:

13.2.1. OEM applications that ship with an installed license

For OEM applications that ship with an installed license, the OEM must do the following:

1. Copy the contents of the appropriate license key into the `systeminfo` file located in the *`%DCF_ROOT%\cfg`* (or *`$DCF_ROOT/cfg`*) directory.
2. If you do not have a `cfg` directory in your deployment, then you should set the `%DCF_CFG%` (or `$DCF_CFG`) environment variable to point to the location where you will install the license and then place the `systeminfo` file in that location.
3. Track deployed licenses as required by the DCF License Contract agreement.

13.2.2. OEM applications that do not ship with an installed license

OEM applications that do not ship with an installed license typically require the end user to install a license before the DCF-based portion of the OEM application will run. In this case, the OEM must create or provide an application, utility, or some other method that allows the end user to select and install the required DCF license key in the appropriate location in the OEM’s software installation hierarchy.

An example license installation/upgrade utility is provided with the DCF Toolkit and all the sources are included. This example may serve as a guide for developing an OEM license installer and may be found in: *`%DCF_ROOT%\win_install\upgrade_license.nsi`*. This is, in fact, the script that is used to install/upgrade the license used by a DCF product.

Basically this example license install/upgrade utility copies the contents of the appropriate license key into the `systeminfo` file, which is normally located in the *`%DCF_ROOT%\cfg`* (or *`$DCF_ROOT/cfg`*) directory. Note: if you do not have a `cfg` directory in your deployment, then you should set the `%DCF_CFG%` (or `$DCF_CFG`) environment variable to point to the location where you will install the license and then place the `systeminfo` file in that location.

A **typical OEM deployment plan** for applications that do not ship with an installed license might look something like the following:

1. Install the OEM application – this installation would include all appropriate DCF files, but no DCF license key file (i.e., no `systeminfo` file).
2. An attempt to use a DCF component without a valid license present causes the DCF system to write an error to the system console and exit. The OEM application should check for the presence of a DCF license file and post some friendly message telling the end user how they can purchase and install the appropriate license file to enable the desired DCF-based functionality.
3. Note: You may run the *`dcf_info`* utility command to check the validity of the installed license. Running *`dcf_info -c`* produces "VALID" or "INVALID"; running *`dcf_info -C`* produces a more verbose message. See the *`dcf_info`* usage statement for more information.
4. The end user orders and receives a license file, then installs it on their target system using the license installation mechanism provided by the OEM.
5. The next time the end user runs the OEM application, the DCF license key is found and the DCF-based functionality is available.

13.2.3. OEM applications that use an activatable license

It is possible to ship an OEM application with an activatable license – see Appendix J: Product Licensing and Activation for information on this option.

13.3. Limiting the DCF libraries required for Deployment

To deploy applications or code that incorporate features, libraries, or utilities based on the DCF Toolkit, an OEM must include various components and applications based on how the OEM application is designed, built, and deployed.

One approach is to deploy all of the DCF runtimes, as well as all potential third-party applications and runtimes. This may be the simpler approach, but may include more than is desired. This section describes how to restrict what is deployed to the minimal set of runtimes that are required by the OEM application.

Some third-party software may be required by DCF-based applications. For example, when the DCF service tools and web-based service interface is deployed, then the following may need to be installed on the target platform:

- Perl
- Java
  - Java Advanced Imaging (JAI)
  - Java Advanced Imaging I/O (JAI I/O)

  These allow certain Java applications to read and write DICOM files with JPEG data. They are not required for the DCF but may be necessary depending on the JPEG compression options you elect to use. The class com.lbs.DCS.JAIUtil still uses JAI and perhaps some of JAI I/O. If you're not using that class, and you are not using DicomEncapsulatedCodecJAI, which is not used by default, then you don't need to include JAI or JAI I/O.
- Apache

There are specific DCF related components that must be deployed to the target platform. These may include portions of the following:

- **Configuration information** – the DCF configuration (CFG) database tree and its associated text files (for more information about the DCF configuration database, see Section 9, Configuring DCF Applications).
- **O/S environment settings** – appropriate O/S environment settings must be made. This may involve setting environment variables that are read by the DCF libraries. See Section13.4.2, which is an installer example that shows the setting of some environment variables that an application may require.
- **O/S-specific dependent dll's or libraries** – ensure that the target contains all libraries required by the O/S (e.g., a Windows application might require that the target platform has the .NET framework installed, while a Linux application might require some GNU libraries).
- **OEM application dependent dll's or libraries** – Running a dependency check on the OEM executable to find the dependent libraries should show the minimal set of required libraries (e.g., on Windows, run `dumpbin /dependents <filename>`; on Linux run `ldd <filename>`).

These DCF related components are described more fully below:

Configuration information – DCF applications and components store their configuration information in text files in the `$DCF_CFG` directory (usually `$DCF_ROOT/cfg`). Applications put their files in the `apps/defaults` subdirectory, while components will have theirs in the appropriate subdirectory under the `components` subdirectory: cpp_lib for C++ libraries, java_lib for Java libraries, cs_lib for C# assemblies, etc. Applications that use DCF libraries and classes may need to include the corresponding configuration files in a tree-structured directory hierarchy. *It is important to note that the DCF run-time license file must be included in the CFG database***.** Copy your run-time license file to `$DCF_ROOT/cfg/systeminfo` and be sure your deployment includes this file.

O/S environment settings – The DCF often uses environment variables to specify the paths to components. For example, the DCF_CFG variable may be set to indicate where the configuration files are located – in fact, this environment variable usually *must* be set for proper operation of a DCF-based application. Your installer may need to set this or other environment variables or settings, or tell the user to set these configurations.

O/S-specific dependent dll's or libraries – All applications use some libraries that are part of the operating system and that did not come with the DCF toolkit. When deploying your application onto a new system, you may have to include some of these libraries along with your application – sometimes this is because that DLL (or library) does not exist on the target system, other times your application may require a different version of the DLL than is already present on the system. These dependencies can be determined by knowing what files your application and its components are linking with. On Windows, this can be determined with the "`dumpbin /dependents <filename>`" command; for Linux, you would use the "`ldd <filename>`" command. A Java app could require that the target system have Java, JAI, and JAI_imageio installed and in its PATH.

Applications built with Microsoft Visual Studio 8 may need additional libraries to be installed in order for them to operate correctly. Microsoft has provided the "VC 8.0 Redistributable" to install updated versions of some C++ libraries, including the MSVCRT libraries (C run-time libraries). This should be installed for VC8-based DCF applications. Additionally, applications built with Visual Studio 7 or Visual Studio 8 or higher may need to have the appropriate .NET Frameworks installed so that DCF libraries can, if necessary, be installed into the Global Assembly Cache (GAC).

OEM application dependent dll's or libraries – There are also the DCF libraries that the application uses, and this must be included. The dependencies can be determined by the "dumpbin" or "ldd" commands (as described above; see below for an example) for C++ applications.

For Java applications, the DCF dependencies are those libraries and classes that are imported. At a minimum, that includes `DCF_DCFCore.dll` and its dependencies. An example showing this is included in Section 5.4:

For C# applications, the DCF dependencies are those libraries and classes that are imported or included for use by the application, including DCF_TSCW.dll and its dependencies.

See also Section 13.4.1 - Language Specific Standalone Installation below.

Note: the OEM must also include any libraries developed specifically for the OEM application to use.

13.3.1. Find Application Dependencies - Windows

The OEM must determine the dependencies for the components they choose to deploy. Shown below is an example for one Windows application.

Windows example – check for dependencies in *`dcf_store_scp.exe`*:

```
C:\DCF-3.3.360c\bin>dumpbin /dependents dcf_store_scp.exe
Microsoft (R) COFF/PE Dumper Version 9.00.30729.1
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file dcf_store_scp.exe

File Type: EXECUTABLE IMAGE

  Image has the following dependencies:

    MSVCR90.dll
    MSVCP89.dll
    omniORB414_vc9_rt.dll
    omnithread34_vc9_rt.dll
    omniDynamic414_vc9_rt.dll
    DCF_DCFCore.DLL
    DCF_LOG_a.DLL
    DCF_CDS_a.DLL
    DCF_APC_a.DLL
    DCF_DDS_a.DLL
    DCF_DCS.DLL
    DCF_DSS.DLL
    KERNEL32.dll

  Summary

        1000 .data
        1000 .pdata
        4000 .rdata
        1000 .reloc
        1000 .rsrc
        5000 .text

C:\DCF-3.3.36c\bin>
```

Alternately, you might find the same information by using a program like “Dependency Walker” (*`http://www.dependencywalker.com`*).

Troubleshooting application dependencies may be aided by the use of a freeware tool like `FileMon` for Windows, which may be downloaded from:
`http://www.sysinternals.com/utilities/filemon.html`.

In their own words: *FileMon monitors and displays file system activity on a system in real-time. Its advanced capabilities make it a powerful tool for exploring the way Windows works, seeing how applications use the files and DLLs, or tracking down problems in system or application file configurations.*

13.3.2. Find Application Dependencies - Linux

The OEM must determine the dependencies for the components they choose to deploy. Shown below is an example for one Linux application.

Linux example – check for dependencies for `dcf_store_scp`:

```
$ cd $DCF_ROOT/DCF

$ . ./dcf.env
[snip]

$ cd $DCF_ROOT/bin

$ ldd dcf_store_scp
  libDCF_dcfcore.so => /home/dcfbuild/DCF/lib/libDCF_dcfcore.so (0x40018000)
  libDCF_dcfutil.so => /home/dcfbuild/DCF/lib/libDCF_dcfutil.so (0x4005d000)
  libDCF_log_a.so => /home/dcfbuild/DCF/lib/libDCF_log_a.so (0x40063000)
  libDCF_dlog.so => /home/dcfbuild/DCF/lib/libDCF_dlog.so (0x40079000)
  libDCF_apc_a.so => /home/dcfbuild/DCF/lib/libDCF_apc_a.so (0x40084000)
  libDCF_cds_a.so => /home/dcfbuild/DCF/lib/libDCF_cds_a.so (0x400b4000)
  libDCF_dcds.so => /home/dcfbuild/DCF/lib/libDCF_dcds.so (0x400d3000)
  libDCF_dds_a.so => /home/dcfbuild/DCF/lib/libDCF_dds_a.so (0x400ee000)
  libDCF_dds.so => /home/dcfbuild/DCF/lib/libDCF_dds.so (0x400ff000)
  libDCF_dcs.so => /home/dcfbuild/DCF/lib/libDCF_dcs.so (0x40104000)
  libDCF_ddcs.so => /home/dcfbuild/DCF/lib/libDCF_ddcs.so (0x40286000)
  libDCF_boost_regex.so => /home/dcfbuild/DCF/lib/libDCF_boost_regex.so (0x402dd000)
  libDCF_dss.so => /home/dcfbuild/DCF/lib/libDCF_dss.so (0x4038e000)
  libomniORB4.so.0 => /opt/omniORB-4.1.4/lib/libomniORB4.so.0 (0x00f75000)
  libomnithread.so.3 => /opt/omniORB-4.1.4/lib/libomnithread.so.3 (0x0038b000)
  libpthread.so.0 => /lib/i686/libpthread.so.0 (0x405e3000)
  libstdc++.so.5 => /opt/gcc-3.2.2/lib/libstdc++.so.5 (0x405f8000)
  libm.so.6 => /lib/i686/libm.so.6 (0x406b3000)
  libgcc_s.so.1 => /opt/gcc-3.2.2/lib/libgcc_s.so.1 (0x406d6000)
  libc.so.6 => /lib/i686/libc.so.6 (0x406df000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)

$
```

13.3.3. Find Application Dependencies - Java

The OEM must determine the dependencies for the components they choose to deploy. There are numerous open source tools that may be used to assist in finding the dependencies, or it may be done as a manual process.

For Java applications, the DCF dependencies are those libraries and classes that are imported by the application. At a minimum, that includes `DCF_DCFCore.dll` and its dependencies. An example showing these and other dependencies for a sample Java application is included in Section 5.4

Java applications that read and write JPEG data might require the JAI and JAI I/O packages, depending on the compression options that you desire.

13.4. Deploying standalone applications containing DCF code

Information is provided in the language specific sections of this guide that show examples of how to deploy simple standalone DCF-based applications. An example of a more full-featured installer is shown below.

13.4.1. Language Specific Standalone Installation

For a C++ application – see Section 4.3: Deploying a Simple Standalone DCF C++ Application.

For a C# application – see Section 6.4: Deploying a Simple C# Standalone Application

For a Java application – see Section 5.4: Deploying a Simple Standalone DCF Java Application

13.4.2. An Example Application Installer

An example installer for a standalone Windows application (dcf_filter) is provided with the DCF Toolkit distribution. Look at the file *%DCF_ROOT%\win_install\ex_one_app.nsi*. This is an example configuration file for generating a NullSoft Installer application for Windows platforms.

(See NSIS – Nullsoft Scriptable Install System, http://nsis.sourceforge.net/.)

The contents of this file may be used as a guideline for selecting the required components for other target platforms as well. If you look closely at this file, you can see that it includes the main application (dcf_filter), the DCF libraries that the application uses, any additional libraries for it, and the application's configuration file – *it shows what files need to be installed and where to put them***.** The installer example also shows the setting of some environment variables that the application will be using. This example installer also does other things and uses the NSIS syntax, but it gives a general idea of how one goes about deploying a DCF application.

Example Windows NullSoft installer configuration file – *ex_one_app.nsi*:

```
;----------------------------------------------------------------------
; ex_one_app.nsi
;
; This is an example installer for how to install a single DCF-based
; application.  This is a very simple installer: it shows how to
; install the files and set up the bare minimum of environment variables
; necessary, and also how to uninstall the files and the env vars.
;
; In this case, it installs the dcf_filter application and the libraries
; and configuration files needed for it to run.  It also shows how to
; uninstall the dcf_filter and its files.
;
;----------------------------------------------------------------------

;----------------------------------------------------------------------
# Write environment changes for the current user only
# !define ALL_USERS ; uncomment this if changes should be for all users
!include WriteEnvStr.nsh
;----------------------------------------------------------------------

!include ReplaceInFile.nsh

# The files are referenced from the DCF_ROOT directory
!cd ".."

; The name of the installer
Name "DCF Filter"

; The file to write
OutFile "ex_one_app.exe"
```

```
; The default installation directory
InstallDir "$PROGRAMFILES\DCF Filter"

!define DCF_ROOT $%DCF_ROOT%

;--------------------------------------------------------------------
; Indicate where additional libraries should be gotten from

…

;--------------------------------------------------------------------
; Version information
;     (all these may be overriden on compilation with /D switches)

…
;--------------------------------------------------------------------
; Indicate the version of the product in the installer

…

;===================================================================
; Pages
;===================================================================

Page directory         # Let the user choose where to install the app
Page instfiles         # Install the files
UninstPage instfiles   # on uninstall, remove the files

;--------------------------------------------------------------------
; The files to install
Section "" ;No components page, name is not important

   ; Set output path to the installation directory.
   SetOutPath $INSTDIR

   ; Put these files there:
   ; The main executable
   File bin\dcf_filter.exe

   ; the DCF libraries
   File lib\DCF_APC_a.dll
   File lib\DCF_CDS_a.dll
   File lib\DCF_DCFCore.dll
   File lib\DCF_DCFUtil.dll
   File lib\DCF_LOG_a.dll
   File lib\DCF_DCS.dll
   File lib\DCF_DLOG.dll
   File lib\DCF_boost_regex.dll

   ; the support libraries (names may be slightly different from that shown here)
   File ${DEVEL_OMNI_BIN}\omniORB414_rt.dll
   File ${DEVEL_OMNI_BIN}\omnithread34_rt.dll
   File ${DEVEL_OMNI_BIN}\omniDynamic414_rt.dll

   ; Include a sample config file for the users
   File devel\csrc\dcf_filter\sample.cfg

   ; Systeminfo is needed for the license
   SetOutPath $INSTDIR\cfg
   File cfg\systeminfo

   ; The appcfg must be included – this is the configuration file for the app
   SetOutPath $INSTDIR\cfg\apps\defaults
   File cfg\apps\defaults\dcf_filter
   ; Set the DCF_CFG environment variable so the app
   ; knows where to find the config files
   Push "DCF_CFG"
   Push "$INSTDIR\cfg"
   Call WriteEnvStr

   ; Optional: Set the DCF_ROOT environment variable and include
   ; the dcfversion file - these are useful if the user specifies
   ; '-l' on the app's command line.
   ;    Push "DCF_ROOT"
   ;    Push "$INSTDIR"
   ;    Call WriteEnvStr
   ;    SetOutPath $INSTDIR
   ;    File dcfversion

   ; Replace references to the development box directory structure
   ; (this is optional)
   Push "${DCF_ROOT}"  # First change any backslashes to slashes
   Push "\"
   Call StrSlash
   Pop $0

   Push "$INSTDIR"     # First change any backslashes to slashes
   Push "\"
   Call StrSlash
   Pop $1

   Push $0      # Now change all references: $0 becomes $1
   Push $1      # (i.e., $DCF_ROOT --> $INSTDIR)
   Push all
   Push all
   Push "$INSTDIR\cfg\apps\defaults\dcf_filter"
   Call AdvReplaceInFile

   ; Indicate that the log server should not be used by default
   ; (this is optional as well)
   Push "use_log_server = TRUE"
   Push "use_log_server = FALSE"
   Push all
   Push all
   Push "$INSTDIR\cfg\apps\defaults\dcf_filter"
   Call AdvReplaceInFile

   ; Create the log directory (for when the user specifies -l
   ; on the app's command line)
   CreateDirectory "$INSTDIR\tmp\log"

   ; Create the uninstaller
   WriteUninstaller "$INSTDIR\uninst-one-app.exe"

   ; This is a very basic installer, so we don't bother with
   ; Start menu options.
SectionEnd ; end the section

;----------------------------------------------------------------------
; This shows how to uninstall the application.
Section Uninstall

…
 [remainder of file ommitted]
```

13.5. DICOM Ports

The DCF can use almost any port that you specify for its communication with other DICOM SCUs and SCPs. The default DICOM ports that should be used are 104 and 11112. The DCF will automatically open up a "hole" through the Windows Firewall for port 104 when you install it; if you wish to use port 11112, you should create a similar "hole" for it in the Firewall.

Note that port 104 may be used by any of the DCF's SCPs – but they do *not* use it by default as you develop with the DCF. When you are deploying your application, you should select (or allow the user to select) the port that will be used for DICOM communication. You should write your installer to open the "hole" for the necessary port(s) that you will be using.

It is also possible to set the DICOM port that is being used – one way to do this is to use the configuration GUIs that are provided by the DCF (see Appendix H: , Section 3: CDS Configuration Shortcuts for more information about this option). It is also possible to set the DICOM port manually or during the installation, but it is up to you to configure the Firewall as necessary.

Appendix A: Glossary of Terms

See also `%DCF_ROOT%\Docs\DCF-DICOM-Glossary.pdf`

A

Abstract Syntax: A DICOM term which is identical to a DICOM SOP Class; it identifies a set of SOPs which, when taken together, represent a logical grouping. An Abstract Syntax identifies one SOP Class or Meta SOP Class.

ACR: American College of Radiology.

Annotation Box: A DICOM name for annotation text printed on the film or other media.

ANSI: American National Standards Institute.

Application Entity (AE): A DICOM term for defining a particular user at an IP address.

Association: A DICOM term for a communication context which is used by two Application Entities that communicate to one another.

Association Negotiation: The software handshaking that occurs between two DICOM Application Entities to set up an Association.

Attribute: Each DICOM information object has its own set of characteristics or attributes. Each attribute has a name and may have a value (see IOD), depending on its category.

B

Big Endian: A term for encoding data where the most-significant byte appears first and remaining bytes follow in descending order of significance; sometimes known as "Motorola" format (see Little Endian). (The term is used because of an analogy with the story Gulliver's Travels, in which Jonathan Swift imagined a never-ending fight between the kingdoms of the Big-Endians and the Little-Endians, whose only difference is in where they crack open a hard-boiled egg.)

C

Calling (Requesting) AE Title: The name used by the receiver in a DICOM Association to indicate which Application Entity it received the data from. It is the AE Title of the AE that is initiating the transfer.

Called (Receiving) AE Title: The name used by the sender in a DICOM Association to indicate which Application Entity it wants to transmit its data to. It is the AE Title of the AE that is receiving the transfer.

Command Element: An encoding of a parameter of a command which conveys this parameter's value.

Command Stream: The result of encoding a set of DICOM Command Elements using the DICOM encoding scheme.

Composite Information Object: A DICOM information object (see IOD) whose attributes contain multiple real world objects.

Conformance: Conformance in the DICOM sense means to be in compliance with the parts of the DICOM Standard.

Conformance Statement: A document whose organization and content are mandated by the DICOM Standard, which allows users to communicate how they have chosen to comply with the Standard in their implementations (see Section 8).

Combined Print Image: a pixel matrix created by superimposing an image and an overlay, the size of which is defined by the smallest rectangle enclosing the superimposed image and overlay.

D

Data Dictionary: A registry of DICOM Data Elements which assigns a unique tag, a name, value characteristics, and semantics to each Data Element (see the DICOM Data Element Dictionary in DICOM PS 3.6-2004).

Data Element: A unit of information as defined by a single entry in the data dictionary. An encoded Information Object Definition (IOD) Attribute that is composed of, at a minimum, three fields: a Data Element Tag, a Value Length, and a Value Field. For some specific Transfer Syntaxes, a Data Element also contains a VR Field where the Value Representation of that Data Element is specified explicitly.

Data Set: Exchanged information consisting of a structured set of Attribute values directly or indirectly related to Information Objects. The value of each Attribute in a Data Set is expressed as a Data Element.

Data Stream: The result of encoding a Data Set using the DICOM encoding scheme (Data Element Numbers and representations as specified by the Data Dictionary).

DICOM: Digital Imaging and Communications in Medicine.

DICOMDIR File: A unique and mandatory DICOM File within a File-set which contains the Media Storage Directory SOP Class. This File is given a single component File ID, DICOMDIR.

DICOM File: A DICOM File is a file with a content formatted according to the requirements of DICOM PS 3.10-2004. In particular such files shall contain, the File Meta Information and a properly formatted Data Set.

DICOM File Format: The DICOM File Format provides a means to encapsulate in a File the Data Set representing a SOP Instance related to a DICOM Information Object.

DICOM File Service: The DICOM File Service specifies a minimum abstract view of files to be provided by the Media Format Layer. Constraining access to the content of files by the Application Entities through such a DICOM File Service boundary ensures Media Format and Physical Media independence.

DIMSE: DICOM Message Service Element. This represents an abstraction of a common set of things that a user would do to a data element, would likely use over and over, and would appear in various different contexts.

DIMSE-C: DICOM Message Service Element—Composite.

DIMSE-C services: A subset of the DIMSE services which supports operations on Composite SOP Instances related to composite Information Object Definitions with peer DIMSE-service-users.

DIMSE-N: DICOM Message Service Element—Normalized.

DIMSE-N services: A subset of the DIMSE services which supports operations and notifications on Normalized SOP Instances related to Normalized Information Object Definitions with peer DIMSE-service-users.

E, F

File: A File is an ordered string of zero or more bytes, where the first byte is at the beginning of the file and the last byte at the end of the File. Files are identified by a unique File ID and may by written, read and/or deleted.

File ID: Files are identified by a File ID which is unique within the context of the File-set they belong to. A set of ordered File ID Components (up to a maximum of eight) forms a File ID.

File ID Component: A string of one to eight characters of a defined character set.

File Meta Information: The File Meta Information includes identifying information on the encapsulated Data Set. It is a mandatory header at the beginning of every DICOM File.

File-set: A File-set is a collection of DICOM Files (and possibly non- DICOM Files) that share a common naming space within which File IDs are unique.

File-set Creator: An Application Entity that creates the DICOMDIR File (see DICOM PS 3.10, section 8.6) and zero or more DICOM Files.

File-set Reader: An Application Entity that accesses one or more files in a File-set.

File-set Updater: An Application Entity that accesses Files, creates additional Files, or deletes existing Files in a File-set. A File-set Updater makes the appropriate alterations to the DICOMDIR file reflecting the additions or deletions.

Film Box: A Normalized Information Object which is the DICOM name for the equivalent of a sheet of physical film.

Film Session: A Normalized Information Object which is the DICOM name for the equivalent of a typical “study” or “series”.

G, H, I

Image Box: A Normalized Information Object which is the DICOM name for the equivalent of a typical “frame” or “image”.

Information Object Class or

Information Object [Definition] (IOD): A software representation of a real object (e.g., CT Image, Study, etc.). An Information Object is generally a list of characteristics (Attributes) which completely describe the object as far as the software is concerned. The formal description of an Information Object generally includes a description of its purpose and the Attributes it possesses.

Information Object Instance or

Instance (of an IOD): A software representation of a specific occurrence of a real object or entity, including values for the Attributes of the Information Object Class to which the entity belongs..

J, K, L

Little Endian: A term for encoding data where the least-significant byte appears first and remaining bytes follow in ascending order of significance; sometimes known as "Intel" format (see Big Endian).

LUT: Lookup Table.

M

Media Storage Application Profile: A Media Storage Application Profile defines a selection of choices at the various layers of the DICOM Media Storage Model which are applicable to a specific need or context in which the media interchange is intended to be performed.

Media Format: Data structures and associated policies which organize the bit streams defined by the Physical Media format into data file structures and associated file directories.

Media Storage Model: The DICOM Media Storage Model pertains to the data structures used at different layers to achieve interoperability through media interchange.

Media Storage Services: DICOM Media Storage Services define a set of operations with media that facilitate storage to and retrieval from the media of DICOM SOP Instances.

Message: A data unit of the Message Exchange Protocol exchanged between two cooperating DICOM Application Entities. A Message is composed of a Command Stream followed by an optional Data Stream.

Meta Service-Object Pair (SOP) Class: a pre-defined set of SOP Classes that may be associated under a single SOP for the purpose of negotiating the use of the set with a single item.

Meta SOP Class: A collection or group of related SOP Classes identified by a single Abstract Syntax UID, which, when taken together, represent a logical grouping and which are used together to provide a high-level functionality, e.g., for the purpose of negotiating the use of the set with a single item.

Module: A logical group of the valid attributes of DICOM information objects.

N

NEMA: National Electrical Manufacturers Association.

Normalized Information Object: A DICOM Information Object (see IOD) whose attributes contain a single real world object. *Note: the differentiation of normalized versus composite information object definitions is not strongly enforced in DICOM 3.0.*

O, P

Physical Media: A piece of material with recording capabilities for streams of bits. Characteristics of a Physical Media include form factor, mechanical characteristics, recording properties and rules for recording and organizing bit streams in accessible structures

Presentation Context: A Presentation Context consists of an Abstract Syntax plus a list of acceptable Transfer Syntaxes. The Presentation Context defines both what data will be sent (Abstract Syntax) and how the data are encoded to be sent (Transfer Syntax).

Print Job SOP Class: A DICOM representation of a Print Job which consists of a set of IODs which describe a Print Job and a set of services which can be performed on those IODs.

Print Management Service Class or

Print Service Class (PSC): A DICOM term for a logical grouping of Service Classes which all involve printing, also referred to as Print Management Service Class (an example of a Meta SOP Class).

Printer SOP Class: A DICOM representation of a Printer which consists of a set of IODs which describe a Printer and a set of services which can be performed on those IODs.

Protocol Data Unit (PDU): A data object which is exchanged by software protocol devices (entities, machines) within a given layer of the protocol stack.

Q, R

Real-World Activity: Something which exists in the real world and which pertains to specific area of information processing within the area of interest of the DICOM Standard. A Real-World Activity may be represented by one or more SOP Classes.

Real-World Object: Something which exists in the real world and upon which operations may be performed which are within the area of interest of the DICOM Standard. A Real-World Object may be represented through a SOP Instance.

S

Secure DICOM File: A DICOM File that is encapsulated with the Cryptographic Message Syntax specified in RFC 2630.

Secure File-set: A File-set in which all DICOM Files are Secure DICOM Files.

Secure Media Storage Application Profile: A DICOM Media Storage Application Profile that requires a Secure File-set.

Service Class: A group of operations that a user might want to perform on particular Information Objects. Formally, a structured description of a service which is supported by cooperating DICOM Application Entities using specific DICOM Commands acting on a specific class of Information Object.

Service Class Provider (SCP, Provider, Server): A device (DICOM Application Entity (DIMSE-Service-User)) which provides the services of a DICOM Service Class or Classes which are utilized by another device (SCU) and which performs operations and invokes notifications on a specific Association.

Service Class User (SCU, User, Client): A device (DICOM Application Entity (DIMSE-Service-User)) which utilizes the DICOM Service Class or Classes which are provided by another device (SCP) and which invokes operations and performs notifications on a specific Association.

Service-Object Pair (SOP) Class: the union of a specific set of DIMSE Services and one related Information Object Definition (as specified by a Service Class Definition) which completely defines a precise context for communication of operations on such an object or notifications about its state.

Service-Object Pair (SOP) Instance: a concrete occurrence of an Information Object that is managed by a DICOM Application Entity and may be operated upon in a communication context defined by a specific set of DIMSE Services (on a network or interchange media). A SOP Instance is persistent beyond the context of its communication.

SOP Class: A DICOM term which is identical to an Abstract Syntax; it identifies a set of SOPs which, when taken together, represent a logical grouping (see Meta SOP Class).

Storage Service Class (SSC): A DICOM term for a logical grouping of Service Classes which all involve storage of images.

T

Tag: A unique identifier for an element of information composed of an ordered pair of numbers (a Group Number followed by an Element Number), which is used to identify Attributes and corresponding Data Elements.

TCP/IP: Transmission Control Protocol / Internet Protocol.

Transfer Syntax: A part of the DICOM Presentation Context which specifies a set of encoding rules that allow Application Entities to unambiguously negotiate the encoding techniques (e.g., Data Element structure, byte ordering, compression) they are able to support, thereby allowing these Application Entities to communicate.

U

Unique Identifier (UID): A globally unique identifier (based on the structure defined by ISO 8824 for OSI Object Identifiers) which is assigned to every DICOM information object as specified by the DICOM Standard (see Section 2.1.1.4) and which guarantees global unique identification for objects across multiple countries, sites, vendors and equipment.

V

Value Representation (VR): A VR is the defined format of a particular data element.

W, X, Y, Z

- End of Glossary -

Appendix B: Bibliography

1. The DICOM Standard

ACR (the American College of Radiology) and NEMA (the National Electrical Manufacturers Association) formed a joint committee to develop a Standard for Digital Imaging and Communications in Medicine (DICOM). The resulting DICOM standard is published by:

National Electrical Manufacturers Association
1300 N. 17th Street
Rosslyn, Virginia 22209 USA

See ACR-NEMA DICOM 3.0 Standard, Parts 1 through 18 (PS 3.1–PS 3.22); ©2020.

Copies of the chapters are available for http access at:

`http://medical.nema.org/dicom`

and for ftp download at:

`ftp://medical.nema.org/MEDICAL/Dicom`

Access to other NEMA related topics is via:

`http://medical.nema.org/`

2. Integrating the Healthcare Enterprise (IHE)

IHE is an initiative by healthcare professionals and industry to improve the way computer systems in healthcare share information. IHE promotes the coordinated use of established standards such as DICOM and HL7 to address specific clinical needs in support of optimal patient care. Systems developed in accordance with IHE communicate with one another better, are easier to implement, and enable care providers to use information more effectively. Physicians, medical specialists, nurses, administrators and other care providers envision a day when vital information can be passed seamlessly from system to system within and across departments and made readily available at the point of care. IHE is designed to make their vision a reality by improving the state of systems integration and removing barriers to optimal patient care.

(http://www.ihe.net/About/index.cfm)

The IHE Technical Frameworks are the detailed reference documents for implementing standards to achieve successful data integration.

The IHE Technical Frameworks, available for download below, are a resource for users, developers and implementers of healthcare imaging and information systems. They define specific implementations of established standards to achieve effective systems integration, facilitate appropriate sharing of medical information and support optimal patient care. They are expanded annually, after a period of public review, and maintained regularly by the IHE Technical Committees through the identification and correction of errata.

Copies are available at: `http://www.ihe.net/Technical_Framework/index.cfm`.

3. Sources for Compression Related Information

Many sources of information are available regarding both lossless and lossy compression as it applies to DICOM and medical imaging. Users of lossy compression have an obligation to ensure the appropriateness of such compression for their images should they choose to apply it. References listed below are in no way complete, they simply represent a starting point for additional reading.

Additional information on image compression in the DCF Toolkit is found in Chapter 11.

CAR Standards for Irreversible Compression in Digital Diagnostic Imaging within Radiology, were published by the Canadian Association of Radiologists in 2008 & revised in 2011.

> *This standard validates the use of irreversible compression under certain defined circumstances and for specified examination types. The specific recommendations appear in section V. Modification of this standard is anticipated when validation of the use of irreversible compression for thin slice CT is completed.*

A copy is available for download from:

`http://www.car.ca/uploads/standards%20guidelines/201106_EN_Standard_Lossy_Compression.pdf`

The DICOM Committee has a working group dedicated to compression related issues as they apply to the DICOM Standard. Reviewing their information, beyond what is already published in the standard, is also advisable.

The DICOM Strategic Document is available from `http://medical.nema.org/` and contains a description of the various working groups and their purposes.

Working Group 4 (Compression) minutes are available for review from:
`http://medical.nema.org/Dicom/minutes/WG-04/`

Certain studies, reports, and sample images are available from
`ftp://medical.nema.org/MEDICAL/Dicom/DataSets/WG04`

Appendix C: Storing Images from Print SCP

1. Collecting image and association info from Print_SCP

After each association with the *dcf_print_scp* has finished, you can run a program or script that gets the images received by the *dcf_print_scp* and that also gets the pertinent association information from the environment.

Listed below in section 1.2 is an example Windows batch (.bat) script.

- The script writes a file with a directory listing of the contents of the *%DCF_TMP%\job_images* directory. This file will contain the names of the images received.
- Then the script moves the files into the SAME directory where the file was written.
- Then it writes the pertinent association information into that same file.

Please note: this is only a simple example. You should write your own script or program (batch, Perl, C#, etc.) that will create a unique directory name for the images for a given association. Insert your custom script into step 4 of the instructions below instead of the example batch script provided in section 1.2.

1.1. Preparation and Configuration Steps

1. Set the *dcf_print_scp* configuration file to its default settings.
2. In the configuration file *%DCF_ROOT%\cfg\apps\defaults\dcf_print_scp* set the attribute "`max_concurrent_associations`" to 1. This will ensure that only one association is writing to the job_images directory at any given time.

   That section of the file should look something like this:

```
# maximum number of associations that can be simultaneously active (1-1024) -
# the practical maximum depends on system resource availability.
max_concurrent_associations = 1
```

3. In the file *%DCF_ROOT%\cfg\apps\defaults\PrinterServer* set the attribute for `OEMPrinter debug_flags` to `0x20000`. By default the `OEMPrinter` component deletes images from the *scp_images* directory after 20 seconds. This debug flag setting sets the `OEMPrinter` component to NEVER delete images from the *scp_images* directory.

   That section of the file should look something like this.

```
[ OEMPrinter ]
debug_flags = 0x20000
printer_script =
printjob_script =
printer_name = DCF Test Printer
manufacturer = Laurel Bridge Software
manufacturer_model_name = DCF_OEMPrinter_Simulator
device_serial_number = 1.2.3.4
software_version = 2.7.6b
date_of_last_calibration = 20000101
time_of_last_calibration = 162841
job_simulate_sleep_secs = 20
```

4. In the file *%DCF_ROOT%\cfg\apps\PrintSCP\PrintSCP1* set the attribute of `post_association_script` to the location of the attached batch file.

   This section of that file would look something like this:

```
pre_association_script =
#
# cmd line of program to be run at end of association
#
post_association_script = C:\tmp\post.bat
```

1.2. Example Batch Script

Listed below is an example Windows batch (.bat) script.

- The script writes a file with a directory listing of the contents of the *%DCF_TMP%\job_images* directory. This file will contain the names of the images received.
- Then the script moves the files into the SAME directory where the file was written.
- Then it writes the pertinent association information into that same file.

```
@ECHO OFF

dir %DCF_TMP%\job_images\* >> C:\tmp\images\info.txt

move %DCF_TMP%\job_images\* C:\tmp\images

date /T >> C:\tmp\images\info.txt
time /T >> C:\tmp\images\info.txt
echo Calling Presentation Address %DCF_CALLING_PRESENTATION_ADDRESS% >>
C:\tmp\images\info.txt
echo Called Presentation Address %DCF_CALLED_PRESENTATION_ADDRESS% >>
C:\tmp\images\info.txt
echo Calling AE Title %DCF_CALLING_AE_TITLE% >> C:\tmp\images\info.txt
echo Called AE Title %DCF_CALLED_AE_TITLE% >> C:\tmp\images\info.txt
```

1.3. Viewing Print SCP output via a web browser

The following feature is only currently available in the DCF Java Toolkit. It is a customization of the C++ *dcf_print_scp* and the Java *PrinterServer* code and provides the ability to view the output from Print SCP via a web browser. It is primarily useful as a debugging tool when developing print related SCU code.

To activate this feature, do the following steps:

1. Edit cfg file: `%DCF_CFG%\apps\defaults\PrinterServer`
2. set attr: `java_lib/OEMPrinter/device_behavior = create_web_files`
3. Run: *run_apache.pl*
4. Select-a-configuration to start: `print_server_win32.cfg`
5. Run your SCU code. To run the DCF cmd line scu you can do:
6. *cd %DCF_ROOT%\test\print*
7. *runscu_win.pl scu001*
8. Point your favorite web browser to `http://localhost/print_jobs` or whatever URL is appropriate for your system.
9. (You may need to modify your listening port as necessary to match your own configuration.)

The code that produces the html output is:

```
   jsrc/com/lbs/OEMPrinter/OEMPrinterDevice.java
```

Note: that this code also invokes *dcm2jpeg.exe* to render the image boxes.

Important: Print-SCP support is not available in all languages - e.g., to access print-scp in C# today, one would need both the C++ *dcf_print_scp* and the Java *PrinterServer* code. These are currently not included with the DCF C# toolkit.

Appendix D: Using DCF Dicom Filters

1. Fixing or working-around protocol problems

`DicomElementFilter` is a subclass of `DicomInputFilter` that is used to modify header fields, instances of `DicomElement`, or attributes that are not large binary arrays.

(Note – in a future DCF release, elements with multi-valued binary types may also be able to be changed with `DicomElementFilter`*.)*

1.1. An application is sending an incorrect field in a DICOM print request

As an example, suppose some external application is sending an incorrect field in a DICOM print request, and you can't or don't want to change that application's source code.

And suppose your application is the standard *`dcf_print_scu`*, which reads a print job description from a configuration file, and submits it to the `LBS::DSS:PrintClient` class for processing.

Furthermore, suppose that the orientation in the N-CREATE-RQ Film Box message is "PORTRAIT" and you want it to be "LANDSCAPE" (for this example, ignore the fact that this field is normally read from a configuration file, and you would ordinarily simply change that file).

Start the server using the web interface, or the command:

```
perl -S dcfstart.pl -cfg %DCF_CFG%\systems\print_server_win32.cfg
```

(Obviously, if you are connecting to an actual printer device, you don't need to run a DCF server to simulate a printer.)

Create a print job configuration file by saving the following text to the file `print_job.cfg`. Adjust directories, hostnames, port numbers, etc., as needed. (See the attributes: client_address, server_address, and persistent_id.)

```
[ print_job_description ]
# calling AE
client_address = PrintSCU
# called host:port:AE
server_address = localhost:2000:PrintSCP1
request_color = NO
request_print_job_sopclass = 0
poll_print_job = 0
print_job_poll_rate_seconds = 0
print_by_session = 0
response_timeout_seconds = 60
job_timeout_seconds = 180
# implicit little endian
association_ts_uid = 1.2.840.10008.1.2
film_session_count = 1

[ print_job_description/film_session ]
number_of_copies = 1
print_priority = HIGH
medium_type = BLUE FILM
film_destination = MAGAZINE
film_session_label = test1
memory_allocation = 0
owner_id = dcftest
film_box_count = 1

[ print_job_description/film_session/film_box_1 ]
image_display_format = STANDARD\1,1
annotation_display_fmt_id =
film_orientation = PORTRAIT
film_size_id = 14INX17IN
magnification_type = NONE
smoothing_type = NONE
border_density = 0
empty_image_density = 0
min_density = 0
max_density = 400
trim = YES
configuration_information = NONE
illumination = 0
reflected_ambient_light = 0
requested_resolution_id = HIGH
image_box_count = 1
annotation_box_count = 0
presentation_lut_count = 0

[ print_job_description/film_session/film_box_1/image_box_1 ]
image_position = 1
polarity = NORMAL
magnification_type = NONE
smoothing_type = NONE
configuration_information = NONE
requested_image_size = 0
reqd_decimatecrop_behavior = DECIMATE
overlay_box_count= 0
presentation_lut_count = 0
# name of image file
persistent_id = /DCF-3.1.10c/test/images/mr-knee.dcm
# transfer syntax of image file - leave blank for auto-detect
persistent_info = 1.2.840.10008.1.2
```

Enable the DIMSE Read/Write debug flags for the DCS library in the `/apps/defaults/dcf_print_scu` application configuration using the web interface, or the command:

```
cds_client saveattr /apps/defaults/dcf_print_scu/DCS/debug_flags
0x300000
```

Run the client with the new job configuration file

```
dcf_print_scu -f  file:/print_job.cfg
```

Now, let's create a filter set configuration to modify some field in the job. Save the following to a file, `filter1.cfg`, for example.

```
[ filter_1 ]
filter_type = DICOM_ELEMENT_FILTER

[ filter_1/elements_to_match ]
# affected SOP class uid = Film Box
0000,0002 = 1.2.840.10008.5.1.1.2
# command field = N-Create-Rq
0000,0100 = 0x140

[ filter_1/elements_to_replace ]
2010,0040 = LANDSCAPE
```

That configuration specifies one filter of type `DICOM_ELEMENT_FILTER`, and that only messages that contain the fields defined in the `elements_to_match` group will be filtered, which should be the N-CREATE-RQ for `FilmBox`. Before that message is sent (or received, depending on how we install the filters), the `elements_to_replace` group will indicate that the field 2010,0040 (orientation) should be added or replaced with a new value.

Modify the configuration for the *`dcf_print_scu`* application to use this filter set for processing out-bound messages – either edit the configuration file *`%DCF_CFG%\apps\defaults\dcf_print_scu`*, and restart the system, or run the command:

```
cds_client saveattr
/apps/defaults/dcf_print_scu/DCS/association/output_filters/filter_set
_name  file:/filter1.cfg
```

(If you want to affect incoming messages, add the filter set to the “input_filters” group.)

Run the application again, and note the DIMSE Message debug output:

```
dcf_print_scu -f  file:/C:/temp/print_job.cfg
```

If you want to see the effect at the printer server back end, set the debug flags for the `OEMPrinter` library component in the *`PrinterServer`* process using the web interface, or use the command:

```
cds_client saveattr /procs/PrinterServer.001/OEMPrinter/debug_flags
0x40000
```

Run the command again, then examine the *`printer_server`* log file.

Stop the print server system using the web interface or the command:

```
dcfstop.pl
```

Note: see the online documentation for C++ class `LBS::DCS::DicomElementFilter` for more information about this object (to access the on-line docs: from the DCF Remote Service Interface, click on “docs”, then “C++ docs”, “namespace list”, “DCS”, and finally “DicomElementFilter”).

1.2. Modifying the DIMSE messages sent by a Java application

This section discusses using `DicomElementFilter` with a Java application. This example can be easily applied to any Java application that is performing DIMSE Message network I/O with another AE.

To modify the DIMSE messages sent by a Java application, such as *ex_jecho_scu*, which is the example Verification Client, perform the following steps:

Start a DCF system that includes a Verification SCP:

```
perl –S dcfstart.pl -cfg %DCF_CFG%/systems/jstore_server_win32.cfg
```

Change to the Java echo scu example directory:

```
cd %DCF_ROOT%\devel\jsrc\com\lbs\examples\ex_jecho_scu
```

Create a filter configuration in the file *sample_filter.cfg* with the following text:

```
 [ filter_1 ]
filter_type = DICOM_ELEMENT_FILTER

[ filter_1/elements_to_replace ]
0010,0010 = This^Is^Bogus
```

This will perform the rather ridiculous action of adding a patient name DICOM attribute to any DIMSE message that is sent.

Set this filter as the output filter set for the Java I/O library by running the following command (type as one line!):

```
cds_client saveattr
/apps/defaults/ex_jecho_scu/java_lib/DCS/association/output_filters/fi
lter_set_name  file:/sample_filter.cfg
```

Set the DIMSE Message debug flags for the Java I/O library using the web interface, or by running the command:

```
cds_client  saveattr
/apps/defaults/ex_jecho_scu/java_lib/DCS/debug_flags  0x300000
```

Run the Java SCU application:

```
jrun_example.pl com.lbs.examples.ex_jecho_scu.ex_jecho_scu StoreSCP1
localhost 2000
```

2. Using DicomTestFilter to support automated testing

`DicomTestFilter` is configured like other `DicomInputFilter`s, but rather than modifying DIMSE Messages or DICOM Data Sets, its purpose is to inject behavior into the system.

It can be used to force delays between messages, force exceptions to be thrown, or to abort the process. See the generated documentation for the C++ class `LBS::DCS::DicomTestFilter` for additional information.

Configuring an SCP to abort after receiving a particular message

As another example, configure a *dcf_echo_scp* to abort after receiving a C-Echo-Request message. Follow these steps:

1. Create a filter set configuration by saving the following text to the file *%DCF_CFG%\dicom\filter_sets\abort_sample*

```
#
# Demo for use with dcf_echo_scp or other.
# Server will abort when it receives C-Echo-Request
#
[filter_1]
filter_type = DICOM_TEST_FILTER

[filter_1/elements_to_match ]
0000,0100 = 0x30

[ filter_1/abort ]
```

   Note: If you want this configuration to be available permanently, create the file in the directory *%DCF_ROOT%\devel\cfgsrc\dicom\filter_configs* and then run *update_cds.pl* to install it to *%DCF_CFG%.*

2. Start the configuration database server and supporting tasks using the web interface, or by running the command:

```
perl –S dcf_start.pl –cfg %DCF_CFG%\systems\dcds_server_win32.cfg
```

3. Modify the Echo SCP application's configuration so that it applies the defined filter set to received messages.

```
cds_client saveattr
/apps/defaults/dcf_echo_scp/DCS/association/input_filters/filter_set_n
ame  /dicom/filter_sets/abort_sample
```

   Note: You could have accomplished the same thing prior to starting *dcds_server* by editing the file *%DCF_CFG%/apps/defaults/dcf_echo_scp* and then adding the `filter_set_name` attribute to the `DCS/association/input_filters group`:

```
[ DCS/association/input_filters ]
filter_set_name = /dicom/filter_sets/abort_sample
```

4. Start the Verification or Echo SCP from the command line. (Note that all other pre-built *dcf_*_scp* applications support Verification, but we're not running them now.)

```
dcf_echo_scp
```

5. In another command window, run the Echo SCU

```
dcf_echo_scu localhost 2000
```

You should see the echo client application fail with an error similar to the following:

```
[ ERROR(-1) 2004/08/05 17:14:28.721 dcf_echo_scu.25929/DCS thrd=2051
DicomSocket.cpp:464 ]
Error reading PDU:
DCSException:
```

```
IOException:
IOReadException:
DicomSocket: OS socket read or recv failed: end of file
file descriptor = 12
remote address  = localhost:3004
local address   = 127.0.0.1:51417
```

This error is returned because the application *`dcf_echo_scp`* intentionally called `abort()` when it got the C-ECHO-RQ message and *`dcf_echo_scu`* logged an error like the one above when it failed to successfully read (i.e., receive) a C-Echo-Response message.

In your application, you might use such a configuration to test that the appropriate user notification is generated on such a failure.

Simulating other types of error conditions by this approach is especially useful in DICOM testing activities.

3. Logging/Debugging DICOM Filter Effects

Debug flags are a special type of configuration data that are defined for each application; they are typically used to control logging verbosity. Most DCF applications "listen" to their own `debug_flags` attribute in their process configuration; this allows them to change dynamically the amount of debugging information that is output when a user (or another application) changes the value of the debug_flags attribute.

The logging code that normally prints received DIMSE messages runs before the filtering code has been executed. This is not very useful for debugging filtering problems. However, if you enable `df_VERBOSE` debug flag in `cpp_lib/DCS`, you will get log output from the filters showing both before and after contents of DIMSE messages. This logging should be much more helpful.

4. Using DCF DICOM Filters Overview

There are various ways filters can be configured.

Filters may be chained together. The input to the first filter is usually either a subclass of `DicomDeviceInput` (e.g., `DicomFileInput` or `DicomNetInput`, which read from files or association-sockets, respectively) or `DicomDataInput`, which is a pseudo-device that returns a particular data set from memory when asked (similar to a `MemoryInputStream` in certain I/O frameworks).

`DCS.DimseMessageUser` (the base class of `DCS.AssociationAcceptor` and `DCS.AssociationRequester`, and thus all SCU's and SCP's) allows a set of input filters and/or output filters to be defined in the session settings (`DCS.DicomSessionSettings`). These filters will be applied to all DIMSE messages that are processed. Note that these filters are applied to messages after all association negotiation has been completed.

Job descriptions for `DSS.StoreClient` or `DPS.PrintClient` can be defined such that each image or instance is processed with a different set of filters. In this case, the image filtering is performed prior to association negotiation, and prior to the application of any output_filters defined by the `DicomSessionSettings`.

Filters can be created by user code to filter a data set as it is read from a file, and then the filtered data set can be given to some DCF SCU to send as is.

Note that filtering operations can be defined that can easily make an image invalid for a particular use. For example, changing the SOP-CLASS-UID element in a C-Store-Request DIMSE message could result in an image being received by an archive for what appears to be a SOP class that was not originally negotiated.

5. DCF DICOM Filter Configuration Overview

One of the DCF's most powerful features is the ability to process DICOM data sets or DIMSE messages with collections of programmable filters. Developers can also implement their own custom filters and have the DCF use them.

Every filter derives from DicomInputFilter. (You can use DicomInputFilter to filter "output" data as well. We'll discuss that later.)

Every `DicomInputFilter` is constructed with a `CFGGroup` that contains the `filter_class_name` and/or the `filter_type` attributes. If the filter_type is known by the `DicomFilterFactory`, then the correct type can be created. If a custom filter has been created, the `filter_class_name` allows the factory to use reflection to create the filter. (Note that for C#, custom filters should also specify the name of the assembly with the filter class, as "`filter_assembly_name`". This is not necessary in Java, which will search the CLASSPATH for the desired class.)

Additionally the sub-group `[ elements_to_match ]` may be present. This data defines DICOM elements that must be present in the input data for the filter to be applied.

For example:

```
      [ filter_1/elements_to_match ]
      0010,0010 = Public^Jane^Q
```

instructs the filter to change data only if the input contains a DICOM patient name element (tag `0010,0010`) with the value "Public^Jane^Q".

Each `DicomInputFilter` sub class may add additional information to the filter configuration `CFGGroup`. This additional information is typically in sub-groups beneath the top level.

`DicomElementFilter` is used to modify DICOM elements in a DIMSE message in various ways. It is constructed with a configuration containing the following additional sub groups:

1. **[elements_to_copy]**
2. **[elements_to_remove]**
3. **[elements_to_remove_if_null]**
4. **[elements_to_replace]**
5. **[elements_to_modify]**

Each sub group defines one filtering operation. The filtering operations are performed in the following order:

1. **[elements_to_copy]**
   If this group is present, then *only* elements with tags contained in the `cfggroup` are "copied" from input to output. All other elements in the input data are ignored and will not be copied to the output, i.e., this filter effectively removes all but the selected elements. If a tag is in the list of tags to copy, but is not present in a particular data set, then nothing is done and this is not considered an error.

   If this group is not present, the output data set starts as a full copy of the input data set.

2. **[elements_to_remove]**

If this group is present, then any elements with tags contained in the `cfggroup` are removed from the output data set.

3. **[elements_to_remove_if_null]**
   If this group is present, then any elements with tags contained in the `cfggroup` that have a zero length value are removed from the output data set.
4. **[elements_to_replace]**
   If this group is present, then each attribute defines a DICOM Element that will be created and added to the output data set. If the element previously existed in the data set, it is overwritten with the new element. (Note: this could be called **elements_to_add_or_replace**.)
5. **[elements_to_modify]**
   If this group is present, it is searched for sub groups. Each sub group defines modification rules for a single element in the output data set.

   For example, consider the following CFGGroup:

```
[ elements_to_modify/1 ]
tag = 0010,0010
old_value = Smith^Joseph
new_value = Doe^John
new_case = U
move_to = 1111,2222
copy_to = 3333,4444
```

   Each of these config group entries is described below:

   The “tag” attribute indicates that only the element with tag `0010,0010` (Patients Name) will be affected.

   The “old_value/new_value” pair defines a Perl-style regular-expression type text substitution. In this example any occurrence of “Smith^Joseph” in the patient name will be changed to “Doe^John”.

   The “new_case” attribute is optional and can contain the values “U” or “L”. If the value is “U” the output value for the element is converted to all upper case; use “L” for lower case. Any other values will result in the case being unchanged.

   The “move_to” attribute is optional and can contain a DICOM tag. The element being modified is removed from the output data set, and a new element with this tag and the modified value is added.

   The “copy_to” attribute is optional and can contain a DICOM tag. A new element with this tag and a copy of the modified value is added.

   More complex regular expressions may be provided – for example, the following combination will swap the first and second name components, that is, “John^Doe” will become “Doe^John”.

```
old_value = ([^\^]+)\^([^\^]+)
new_value = $2^$1
```

The filters that you use have the capability to record the changes made in the Original Attributes Sequence (OAS). This can be used to create a “history” of the changes to the data resulting from the filtering of the data – the sequence will have the original values of the elements that were changed. How to enable this feature will be described below.

5.1. Sample configuration for the DicomElementFilter class.

```
#
[ filter_1 ]
filter_class_name = LaurelBridge.DCS.DicomElementFilter
filter_type = DICOM_ELEMENT_FILTER
```

```
[ filter_1/elements_to_match ]
0010,0010 = Public^Jane^Q

[ filter_1/elements_to_remove ]
tag = 0028,0010

[ filter_1/elements_to_replace ]
0010,0010 = Doe^John
```

5.2. Other Filter Types

The DCF includes additional built-in filters besides the DICOM Element Filter. The **Pixel Value Shift Filter** can be used to shift the bits in pixel data values left or right, for image data manipulation. (This filter is primarily used in unit/integration tests for data-set and DIMSE-message filtering, but it is also available for real world image data manipulation.) The **Planar Configuration Convert Filter** is used to convert color pixel data from interleaved to planar or vice versa. The **Mapping List Filter** is a specialized and more advanced way to add or replace elements in a DICOM message. It provides an efficient way to match a large number of possible values for a particular attribute (key tag), and then add/replace one or more elements, depending on the key tag values that are matched. The configuration attributes for these will be described below. The **Pad Value Filter** is used to pad a string value with a character until the string is a given length. The **Element Composer Filter** is used to create output elements from fixed text and text that is parsed out of one or more input elements.

5.3. Filter Configuration Files

The DCF stores configuration data, including filter specifications, in text "config files" (see Section 9.1 Configuration Files and the CDS interface above for more detail). It will be easiest to understand how to define a filter in a config file by showing a fairly complete example of such a file.

```
[ more_examples 1 ]
filter_type = DICOM_ELEMENT_FILTER
filter_sub_type = unknown
create_original_attributes_seq = TRUE
source_of_previous_values = ....
source_of_previous_values_tag_name = ....
reason_for_modification = COERCE
modifying_system = MyDCFBasedSystem
save_pixels_in_oas = FALSE

[ more_examples 1/elements_to_match ]
0010,0010 = Doe^John
[ more_examples 1/elements_to_copy ]
tag = 1111,1111
[ more_examples 1/elements_to_remove ]
tag = 1234,4321
[ more_examples 1/elements_to_remove_if_null ]
tag = 5678,9123
[ more_examples 1/elements_to_replace ]
0010,0010 = Bob
[ more_examples 1/elements_to_modify ]


[ more_examples 2 ]
filter_type = DICOM_ELEMENT_FILTER
filter_sub_type = copy filter
create_original_attributes_seq = TRUE
source_of_previous_values = ....
source_of_previous_values_tag_name = ....
reason_for_modification = COERCE
```

```
modifying_system = MyDCFBasedSystem
save_pixels_in_oas = FALSE

[ more_examples 2/elements_to_match ]
[ more_examples 2/elements_to_copy ]
tag = 0051,0051
tag = 0051,0052


[ more_examples 3 ]
filter_type = DICOM_ELEMENT_FILTER
filter_sub_type = remove filter
create_original_attributes_seq = TRUE
source_of_previous_values = ....
source_of_previous_values_tag_name = ....
reason_for_modification = COERCE
modifying_system = MyDCFBasedSystem
save_pixels_in_oas = FALSE

[ more_examples 3/elements_to_match ]
[ more_examples 3/elements_to_remove ]
tag = 1111,2222
[ more_examples 3/elements_to_remove_if_null ]
tag = 1111,3333


[ more_examples 4 ]
filter_type = DICOM_ELEMENT_FILTER
filter_sub_type = add/replace filter
create_original_attributes_seq = TRUE
source_of_previous_values = ....
source_of_previous_values_tag_name = ....
reason_for_modification = COERCE
modifying_system = MyDCFBasedSystem
save_pixels_in_oas = FALSE

[ more_examples 4/elements_to_match ]
0010,0010 = Doe^Jane
[ more_examples 4/elements_to_replace ]
0010,0010 = Public^Jane^Q


[ modify_filter_1 ]
filter_type = DICOM_ELEMENT_FILTER
filter_sub_type = modify filter
create_original_attributes_seq = TRUE
source_of_previous_values = ....
source_of_previous_values_tag_name = ....
reason_for_modification = COERCE
modifying_system = MyDCFBasedSystem
save_pixels_in_oas = FALSE

[ modify_filter_1/elements_to_match ]
[ modify_filter_1/elements_to_modify ]
[ modify_filter_1/elements_to_modify/0 ]
tag = 0010,0010
old_value = (.*)
new_value = $1
new_case = N/C
move_to = 1234,5678

[ modify_filter_1/elements_to_modify/1 ]
tag = 0010,0011
old_value = (.*)
new_value = $1
new_case = N/C
```

```
copy_to = 8765,4321
```

The file shows a set of filters; each of the top-level groups (indicated by square brackets) in the file is an individual filter – "more_examples 1", "more_examples 2", "modify_filter_1", etc. Filters are often grouped into sets for the convenience of chaining related operations together into one file, although you could put each filter into its own file and process the data through each one individually. The filters in a set are processed in order, with the results of each filter being used by the succeeding filters. (Note: the names of the filters are not important; the filters are processed in the order that they are specified in the file.)

At the top of each filter definition you can see the attribute-value pairs specifying that the original attribute sequence should be created and populated with the changes made by each filter.

- `create_original_attributes_seq` – TRUE (or 1) to enable the OAS; FALSE or 0 disable
- `save_pixels_in_oas` – TRUE (or 1) if changes to the pixel data should be recorded in the OAS
- `source_of_previous_values` – you can specify the source of previous values
- `source_of_previous_values_tag_name` – the name of the DICOM tag to read for the source of the previous values. For example, setting this to "1234,5678" indicates that tag 1234,5678 will be read to get the value of the source.
- `reason_for_modification` – defaults to COERCE, but you can set it to the other valid values
- `modifying_system` – the system that is changing the data

If you don't want the changes made by some filters to be recorded, set "create_original_attributes_seq" to FALSE for those filters. This allows you to record the changes for some filters but not for others. (These attributes can be set programmatically via the CFGGroup method `setAttributeValue`.)

> *Note on the Original Attributes Sequence:* You can specify directly the source of the previous values by setting the attribute "`source_of_previous_values`". If you specify "`source_of_previous_values_tag_name`", then its value will be read to determine the source. If "…`tag_name`" is not specified, then the value of the "…`values`" attribute is used. If the "…`tag_name`" attribute **is** specified but that tag does not exist in the original dataset, a blank value will be used to indicate that the appropriate value is unknown.

For each filter, the `filter_type` attribute is required; this tells the DCF what filter code to use. The `filter_sub_type` is not necessary – this is used by DCF GUIs to simplify the displays for users.

Each filter must then specify the necessary sub-groups (e.g., `[more_examples 1/elements_to_match]`) that tell the filter what to do: elements that must be matched in order for the filter to work, which elements should be copied, which should be removed, which ones modified and how they are modified, etc. If the sub-group is empty, it can optionally be omitted without affecting the filter's operation. (For example, note that "`more_examples 2`" only has "elements to match" and "elements to copy"; it doesn't need "elements to remove" or the other possible operations.)

For elements to match, you must specify the tags and their values that must be matched for the filter to be applied. If this is empty, the filter is always applied.

> **Note**: Elements to Match is usually used to make sure that a filter is applied if a certain *value* is *present* in a certain tag in the DICOM dataset to be filtered. But sometimes you may want to filter the data if a tag simply *exists* in the dataset, regardless of the value it has.

Currently, for string VRs (such as PN, AE, CS, LO, SH, ST, LT, and UT) you can specify the value in Elements to Match as an asterisk ("*") to check if the tag exists. Note that this only works for string VRs. To check if a tag exists for date/time VRs (DA, DT, TM), you would need to specify a range of dates or times that would match anything, e.g.,

```
DA = 19000101-20200101.
```

(Future releases of the DCF will include a more generic way to check if a tag exists, including a way that applies to binary and UI VRs.)

For elements to copy, remove, and remove if null, specify the tags that the operation should apply to; note that the format is "tag = value", for each tag that is required.

For elements to add/replace (the filter operation name is "`elements_to_replace`"), you specify the tag and the tag's new value; if the tag does not exist, it will be created with the new value.

For elements to modify, each element to modify is specified in another subgroup ("`modify_filter_1/elements_to_modify/0`", "`modify_filter_1/elements_to_modify/1`", etc). Each subgroup (0, 1, 2,…) indicates a tag and how the data in that tag should be modified.

- `old_value` – a regular expression to parse the data
- `new_value` – how the data should be rearranged or modified from the regex
- `new_case` – U for uppercase, L for lower-case, N/C to leave the case as-is
- `move_to` – move the data from this tag to the new tag
- `copy_to` – copy the data from this tag to the new tag

The Modify filter uses Perl-style regular expressions for its regex syntax.

To create a **Pixel Value Shift Bits Filter**, the filter definition should look like this:

```
[ pixel filter ]
filter_type = DICOM_PIXEL_VALUE_SHIFT_FILTER
[ pixel filter/elements_to_match ]
[ pixel filter/pixel_shift ]
shift_bits = 2
```

The shift bits determines how many bits to left-shift each byte or word (16-bit) sample in OB or OW pixel data values; negative values indicate a right shift.

Note that "elements to match" can be specified to determine if the filter should be applied. Also note that the `shift_bits` attribute is in the `pixel_shift` subgroup. (The OAS attributes have been omitted here for clarity; if needed, they would be inserted directly beneath the `filter_type` attribute.)

To create a **Mapping List Filter**, specify a filter definition like this:

```
[ mapping list ]
filter_type = DICOM_MAPPING_LIST_FILTER
match_tag = 0008,0050
replace_tag = 0020,000D
replace_tag = 0010,0010
mapping_cfg_name = /tmp/bob
no_match_option = 0
mapping_cfg_format = CSV
mapping_cfg_delimiter_char = ","
create_original_attributes_seq = FALSE
[ mapping list/elements_to_match ]
```

The `match_tags` act like a key to a hash of changes that should be made, based on those values; the `replace_tag`s indicate what tags should have their values replaced. `mapping_cfg_name` specifies the full path of the file of mappings to search; `mapping_cfg_format` indicates the format of that file (which **must** be `CSV`), and `mapping_cfg_delimiter_char` specifies the data delimiter character in the file. `no_match_option` tells the filter what to do if the match tag's value cannot be found in the mapping file. (Note that in this example filter, the OAS is turned off.)

The Mapping List Filter is a complicated filter, so it deserves a little more explanation here. The mapping list file is normally a plain text file with comma-separated values corresponding to the "match tags" and the "replace tag" elements. The values are usually separated by commas, but you can also use tabs, semi-colons, or other characters. Note that the delimiter character **must** be enclosed in quotes.

When this filter is applied, the match tags are checked against the "mapping key" values in the mapping list file. If a match is found, the corresponding replace tag values are used to replace those tag values in the data set being processed.

For example, let's say that your "match tag" is 0008,0050 (Accession Number), your "replace tags" are 0020,000D [Study Instance UID] and 0010,0010 [Patient Name] (recall that you can have more than one), and the data in your mapping list file is:

```
12345, 1.2.3.4.5, Doe^John
45678, 4.5.6.7.8, Public^Jane^Q
...
```

The first column in your mapping list file corresponds to the match tag (0008,0050), and the following ones to the replace tags (0020,000D and 0010,0010).

If the match tag in a dataset passing through the filter has the value "12345", then the Study Instance UID would be replaced with "1.2.3.4.5" and the Patient Name with "Doe^John"; if the match tag has the value "45678", then the Study Instance UID would be replaced with "4.5.6.7.8" and the Patient Name with "Public^Jane^Q", and so on.

When specifying this filter you must also set the behavior when no matching "match tag" is found in the mapping list file. There are three choices when no match is found:

- **0** – Reject the data set by aborting the association; the data set is not forwarded to the destination.
- **1** – Log a warning message and forward the filtered data set to the destination.
- **2** – Ignore the error and forward the filtered data set to the destination

It is possible to have multiple match tags – in this case, the replacement occurs only if the values of each of the specified match tags match values in the mapping configuration file. If all of them match, then the replacement values are used to modify the dataset. If *any* of them do *not* match, then the replacement does not occur; in such a case, the resulting behavior is determined by the `no_match_option`.

For example, consider this filter configuration:

```
 [ mapping list ]
filter_type = DICOM_MAPPING_LIST_FILTER
match_tag = 0008,0050
match_tag = 0010,0010
replace_tag = 0020,000D
replace_tag = 0010,0010
mapping_cfg_name = /tmp/bob
no_match_option = 0
mapping_cfg_format = CSV
mapping_cfg_delimiter_char = ","
create_original_attributes_seq = FALSE
```

and its corresponding mapping list file

```
MY_ACCESSION_NUMBER_1, JOHN DOE, 1.2.3.4.5, John^Q^Public
MY_ACCESSION_NUMBER_2, JANE WOE, 1.2.3.4.6, Jane^Citizen
```

- Dataset #1 is to be filtered; it has 0008,0050 (accession number) with a value of "MY_ACCESSION_NUMBER_1", and its patient name (0010,0010) is "JOHN DOE". In this case, the Study Instance UID (0020,000D) will be set to "1.2.3.4.5" and the patient name will become "John^Q^Public".
- Dataset #2 is now to be filtered; its accession number is "MY_ACCESSION_NUMBER_1" and its patient name is "GARY DOE". Since one (or more) of its values do not match, the association will be rejected.
- But suppose dataset #3 has an accession number of "MY_ACCESSION_NUMBER_2" and a patient name of "JANE WOE". All values will find matches in the mapping configuration, and so the Study Instance UID will become "1.2.3.4.6" and the patient name will be changed to "Jane^Citizen".

Note that you can have as many match tags and as many replace tags as you require. In the mapping configuration file, the first **n** values on each line will be used for the mapping values, while the remaining **m** values will be the replace values. The order of the values on each line in the mapping file is important – the first value will be matched against the first specified match tag, the second value will be matched against the second match tag, and so on. Similarly, the values after the match tags will be the replacement tags – the first value will be used for the first replacement tag, the second value for the second replacement tag, and so on. Consider the above example, where the first two values are to be matched, and the remaining two are the replacement values.

To create a **Pad Value Filter**, the filter definition should look like this:

```
[ pad value filter ]
filter_type = DICOM_PAD_VALUE_FILTER
tag = 0010,0010
pad_left = true
length = 20
pad_character = "0"
[ pad value filter/elements_to_match ]
```

The "`tag`" specifies the DICOM tag in the dataset that should have its value padded. "`pad_left`" is *true* if the value should be padded to its left (with leading characters); set it to *false* if the value should be padded to the right (with trailing characters); the default is to pad to the left. "`length`" is how long the final string should be – it is *not* how many pad characters to add. "`pad_character`" is the character to pad the value with; it is enclosed in quotation marks to allow for spaces or other whitespace characters to be used. You may also specify NULL (e.g., "`pad_character = NULL`") to indicate that the value should be padded with the null character. You should be careful when padding a value with nulls, spaces, or other whitespace characters, since such characters can be stripped off when the value is sent or written.

Note that "elements to match" can be specified to determine if the filter should be applied. (The OAS attributes have been omitted here for clarity; if needed, they would be inserted directly beneath the `filter_type` attribute.)

To create an Element Composer Filter, the filter definition should look like this:

```
[ composer_filter ]
filter_type = DICOM_ELEMENT_COMPOSER_FILTER
```

```
create_original_attributes_seq = false
save_pixels_in_oas = FALSE
[composer_filter/elements_to_match ]
0000,0100 = 1

[composer_filter inputs ]
[composer_filter inputs/1 ]
# Accession Number
tag = 0008,0050
regex = (.*)
[composer_filter inputs/2 ]
# Patient ID
tag = 0010,0020
regex = (.*)
[composer_filter outputs ]
new_case = N/C
[composer_filter outputs/1 ]
# Accession Number
tag = 0008,0050
value = ${2.1}
[composer_filter outputs/2 ]
# Patient ID
tag = 0010,0020
value = ${1.1}
```

Here, the Composer is used to swap the Accession Number and the Patient ID for a C-Store-Request. Regular expressions are used to parse the elements in the **inputs** group, and the captured values are recombined in the **outputs** group to create or replace element values. The case of the output values can be changed by specifying the **new_case** attribute – allowed values are "N/C" (no change), "U" (upper), or "L" (lower).

To create a Planar Configuration Convert Filter, the filter definition should look like this:

```
[ cfg ]
filter_type = DICOM_PLANAR_CONFIG_CONVERT_FILTER
[ cfg/planar_config ]
output_planar_config = 1
```

The **output_planar_config** value should be 0 (zero) for interleaved, or set it to 1 (one) for planar.

5.3.1. Specifying a Sequence in a Configuration File

A sequence may be entered in a config group file as a tag by appending it to a numeric tag (the traditional group-element pair) with a period ("`.`"). You may also indicate an item in the sequence with "`#`" and the sequence item ID, followed by the tag indicating the sequence. There may be multiple sequences and sequence IDs as part of one "tag". Examples are shown below:

- Simple tag – `0010,0010`
- Tag with sequence – `0080,0100.0008,0060`
- Tag with sequence ID and sequence – `0080,0100.#0.0008,0060`
- Tag with multiple sequences and IDs – `0080,0100.#1.0080,0100.#0.0008,0060`

If no item number is specified, the first item (#0) is assumed. You can also specify the last element in a sequence by "#L" (upper-case is important!) if you don't know how many items are in a sequence. If you are creating new elements, you can specify the next item in the sequence via "#N" (again, case is important) to append to the sequence. For example: `0080,0100.#L.0010,0010.#N.0008,0060`

Please notice that:

- The sequence IDs (e.g., `#1`) and the tag-value pairs for the sequences are all separated by periods (“`.`”).
- The tags for the sequences are simple group-element pairs themselves.

5.3.2. Using Macros to Specify Data

As you use the DCF to filter data, you may encounter situations where the new values that you want are dynamically changing. For example, you may wish to specify that a tag has the current date and time. Obviously, you can’t specify the current date and time exactly as a text string, since the date and time keep changing. Instead, the DCF provides *macros* to fill in values that are changing.

The DCF provides the following macros for your use:

- ${DATE} – the current date in the format YYYYMMDD, e.g., “20071108” (November 8, 2007)
- ${TIME} – the current time in the format HHmmSS, e.g., “142035” (2:20:35 PM)
- ${DATETIME} – the current date and time in the format YYYYMMDDHHmmSS, e.g., “20071108142035”
- ${GMT_TIME} – the current time for Greenwich Mean Time in the format HHmmSS, e.g., “182035” (6:20:35 PM GMT/UTC)
- ${GMT_DATETIME} – the current date and time for Greenwich Mean Time in the format YYYYMMDDHHmmSS, e.g., “20071108182035”
- ${TZ} – the current time zone, e.g., “Eastern Daylight Time”. Note that the full name, not an abbreviation is returned. (For Linux platforms, “EDT” is returned, not the full name.)
- ${TZOFFSET} – the offset of the current time zone from GMT, e.g., “-0400” for EDT
- ${UID} – generates a new Unique IDentifier

Note that the times specified are in local time unless you use the GMT macros.

You can use these macros to specify the new values for tags just the same way as if you were specifying the exact text. For example, to change the E_INSTANCE_CREATION_DATE (0008,0012), you would specify the new value as “`${DATE}`”. You can also have a value that mixes text and macros. Let us suppose you wanted to change the username to be “Bob <current time>”; you would set the new value to be “`Bob ${TIME}`”, which would give you a result something like “Bob 150721”.

If you wish to specify a value that has a dollar sign (“$”) in it and not have it interpreted as a macro, you should escape it with a backslash, e.g., “`\${UID}`”; this would insert the string ${UID} literally in the value.

These macros greatly increase the flexibility and power of the DCF’s filters and allow you much more capabilities in how your data is filtered. For example, if you wanted to specify a sequence that has known information but also dynamic information like the date, you could create a filter to insert the literal values but also uses the macros to set the date and time.

5.4. Example Filters

- Example 1: Replacing a Value
- Example 2: Removing an Element
- Example 3: Modifying an Element's Value with Regular Expressions
- Example 4: Padding an Element's Value

5.4.1. Example 1: Replacing a Value

Suppose you have a DICOM data set that has an incorrect value, and you want to use a filter to correct that value. For example, if the patient name was "Public^Jane^Q" but was supposed to "Doe^John". Then you could set up a filter to replace the incorrect value every time it was processed by your application.

You would create a new Element filter and specify the elements to replace. If you only want to correct the element when it has the value "Public^Jane^Q", you would first specify as "elements to match" the following values:

Tag	Value
0010,0010	**Public^Jane^Q**

If you want to correct the element every time, even when the value is not "Public^Jane^Q" or the element is not present at all, you would leave this table empty.

Secondly, you would specify the "elements to add/replace":

Tag	Value
0010,0010	**Doe^John**

5.4.2. Example 2: Removing an Element

If you want to remove a DICOM element from a data set, you would create a new element filter and specify the "elements to remove". For example, let us suppose that you always want to remove element 0028,0010 (rows). You would define as "elements to remove" the following values:

Tag	**0028,0010**

You could configure the filter to be applied under certain conditions – i.e., when an element has a certain value – by specifying the "elements to match" (as in the previous example); or you would leave "elements to match" empty to apply the filter all the time.

5.4.3. Example 3: Modifying an Element's Value with Regular Expressions

The "elements to modify" portion of the Element Filter is designed to allow you to modify the value of an element by using a regular expression (or a "regex", as it is commonly called); the so-called "Modify Filter" also allows you the option to move or copy one element's data into other elements.

For example, suppose that your receiving software expects the patient's name to be in your proprietary DICOM tag "`abcd,abcd`", but you are not getting that tag sent from the modalities. In addition, suppose that the data has the first name first, instead of the DICOM default of last name first. You could use the Modify Filter and regular expressions to switch the elements around and move it to the desired tag.

First, you would specify for "elements to modify" the tag value for the standard DICOM Patient Name tag:

Tag	**0010,0010**

Recall that this is part of a sub-group under "`elements_to_modify`", in which you would specify the tag and how the tag's data should be modified.

Use `old_value` and `new_value` to define how the regular expression should modify the data. (An explanation of regular expression syntax is beyond the scope of this document, but many fine examples can be easily found on the Internet.) In this case, you would enter the following:

Old value	**([^]*)\^([^]*)**
New value	**$2\^$1**

Second, to copy the modified result to your private tag as proposed in this example, you would enter your proprietary tag – "`abcd,abcd`" – as the value for `copy_to`. The end result is that the order of the first name and last name will be swapped, and the data will be copied to your tag.

Your text configuration file would look like this:

```
[ my_filter ]
[ my_filter/elements_to_modify ]
[ my_filter/elements_to_modify/0 ]
tag = 0010,0010
old_value = ([^ ]*)\^([^ ]*)
new_value = $2\^$1
new_case = N/C
copy_to = abcd,abcd
```

5.4.4. Example 4: Padding an Element's Value

Suppose you want the patient name (tag `0010,0010`) to always be at least 20 characters long and have leading zeroes as part of it. Your filter configuration would look like this:

```
[ my_filter ]
filter_type = DICOM_PAD_VALUE_FILTER
tag = 0010,0010
pad_left = true
length = 20
pad_character = "0"
```

This would result in the value "`John^Doe`" becoming "`000000000000John^Doe`".

Or suppose you want trailing characters instead of leading. Then set `pad_left` to false, as shown below:

```
[ my_filter ]
filter_type = DICOM_PAD_VALUE_FILTER
tag = 0010,0010
pad_left = false
length = 20
pad_character = "0"
```

Then "`John^Doe`" would become "`John^Doe000000000000`", while "`John^Philip^Sousa`" would become "`John^Philip^Sousa000`", and "`John^Jacob^Jingleheimer^Schmidt`" would be unchanged, since its length is more than 20 characters.

6. Developing Custom Filters

The DCF allows you to create your own custom filters in Java and C#, and also to create your own custom filter editors in Java – these custom filter editors can be used by the Filter Set Editor application/applet to make it easier for users to create and edit filters. These custom filters can be used by the DCF by dynamically loading the classes using reflection – no modifications to the DCF or its interfaces are required. For example, you could create a custom filter that could be used directly by the Java filter example, `ex_jdcf_filter`; the filter application does not need any modification – it can simply load your filter class and use it.

For C++ implementations, dynamic loading of OEM-provided filter implementations (in a user's named DLL, for example) is not currently supported. Instead, the OEM can create a custom `DicomFilterFactory` subclass to their DCF based C++ application. The comment in `DCS/DicomFilterFactory` explains this (excerpted from `DicomFilterFactory.h`):

DicomInputFilter objects are created by the DicomFilterFactory by the createFilter() method. The DicomFilterFactory is a singleton that can be extended by an OEM to allow custom filters to be added to the system.

To create a custom factory, do the following:

```
class CustomFilterFactory : public DicomFilterFactory
{
   virtual DicomInputFilter* createFilter( DicomInput* p_source, const
LBS::CDS::CFGGroup& cfg )
   {
      if ( cfg indicates that this is a custom filter )
         return new MyCustomFilter( p_source, cfg); // inherits from DicomInputFilter
      else
         return DicomFilterFactory::createFilter( p_source, cfg );
   }
}
```

Note for Java, C# DicomConditionalFilter:

Prior to DCF3.3.50, classes implementing matching logic for DicomConditionalFilters would only match the on first value of multi-valued string attributes. As of DCF3.3.50, conditional filters now are able to match multi-valued string attributes.

See the class documentation in Java and C# for more information on the DicomConditionalFilter class.

Additional documentation is available by sending an email to support@laurelbridge.com and requesting the "DicomConditionalFilter Changes for DCF3.3.50" technote.

6.1. Custom Filters in Java

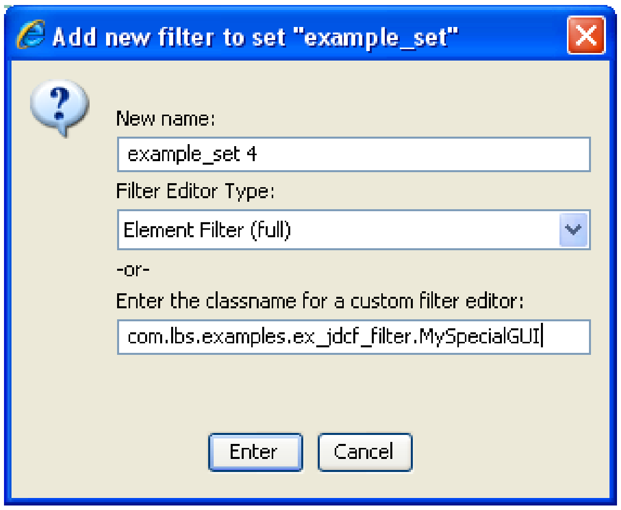
The files `MySpecialFilter.java` and `MySpecialGUI.java`, in the `ex_jdcf_filter` example directory, are simple examples of a custom filter and its corresponding editor GUI.

MySpecialFilter is a very simple filter that can be loaded dynamically by DCF Java implementations; the associated class MySpecialGUI would provide a graphical user interface for configuring MySpecialFilter. The Filter Set Editor allows you to specify the class name of a custom filter editor GUI, which is dynamically loaded by the Filter Set Editor – this allows you to edit your custom filters within the DCF framework and to expand the filtering capabilities beyond what is built into the DCF.

The custom filter to load can be specified with a configuration like this:

```
[ example_filter ]
filter_class_name = com.lbs.examples.ex_jdcf_filter.MySpecialFilter
filter_editor_class = com.lbs.examples.ex_jdcf_filter.MySpecialGUI
…
```

The `filter_class_name` attribute specifies the Java class to be loaded by the DCF, while the `filter_editor_class` attribute specifies the Java class that the Filter Set Editor should load to permit a user to configure the filter. The filter configuration would also include whatever data is necessary to define the filter's operation.


Add new filter to set "example_set"

New name: example_set 4

Filter Editor Type: Element Filter (full)

-or-

Enter the classname for a custom filter editor: com.lbs.examples.ex_jdcf_filter.MySpecialGUI

Figure 16: Using a custom filter editor class

To use your custom filter editor GUI in the DCF Filter Set Editor, you would enter the name of the GUI class when you are adding a new filter to a filter set (see above). The class you specify must be in the CLASSPATH; for applets (including the Filter Set Editor applet), this means that the class must be loaded in the JAR file along with the Filter Set Editor's classes.

For more information, see the documentation for the class `com.lbs.DCS.FilterEditorGUI`.

6.2. Custom Filters in C#

The file `MySpecialFilter.cs`, in the `ex_ndcf_filter` example directory, is a simple example of a custom filter. (Currently, the C# DCF does not provide any graphical user interfaces for editing filters.) It can be loaded dynamically by DCF C# implementations, allowing you to extend the DCF's filtering capabilities as you desire.

The custom filter to load can be specified with a configuration like this:

```
[ example_filter ]
filter_class_name = LaurelBridge.ex_ndcf_filter.MySpecialFilter
filter_assembly_name = ex_ndcf_filter.exe
…
```

The `filter_class_name` attribute specifies the C# class to be loaded by the DCF; the `filter_assembly_name` specifies the assembly that has the class in it – the assembly is loaded first, and the filter class is loaded from it. The filter configuration would also include whatever data is necessary to define the filter's operation.

Appendix E: DCF MakeUID Function

The class DicomDataDictionary in DCF handles generating DICOM UID's. The method makeUID() will return a unique identifier string.

Note: This function is in DicomDataDictionary, since customizing the UID generation algorithm is in some ways similar to customizing the data-dictionary to add new tags, etc. This means UID generation can be done using the DCF code, with custom OEM configuration data, or the OEM can install a new implementation of the DicomDataDictionary class.

The DCF DicomDataDictionary searches the file $DCF_CFG/dicom/uid for the attributes **oem_info/uid_prefix** and **oem_info/uid_system_prefix.**

1. Description of the DCF makeUID function

Implementation: makeUID returns the concatenation of the strings returned by getUIDPrefix() and getUIDSuffix().

i.e., `String uid = getUIDPrefix() + "." + getUIDSuffix();`

The UID is checked for validity (illegal characters, length overflow, etc) prior to returning it. The OEM can provide a custom implementation of getUIDPrefix, getUIDSuffix, initUIDPrefix, or the DCF default implementation can be used: The DCF implementation is described below.

In summary, if you use the default DCF algorithm, you get:

<1.2.840.114089.1.0.1 or cfg-setting>. <ip-addr or cfg-setting>.<time-stamp>.<pid>.<incrementing_sequence_number>

Note: Calls to makeUID are thread safe, i.e. two threads in the same process calling at nearly the same time will not get duplicate UIDs.

2. The function getUIDPrefix() returns the uid_prefix.

The uid_prefix is initialized once for a process by initUIDPrefix, which does the following:

```
uid_prefix =  uid-base + system_prefix + time_stamp
```

- uid_base is either read from the attribute **"oem_info/uid_prefix"** in the configuration file **$DCF_CFG/dicom/uid**, or it is initialized to the DCF default value: `"1.2.840.114089.1.0.1"`

  Note: the uid_base can be configured, this allows your OEM organization code to appear in UID prefixes.

- system_prefix is either read from attribute **"oem_info/uid_system_prefix"** in the configuration file **$DCF_CFG/dicom/uid**, or it is initialized to the DCF default: `<base-10-ip-address-of-localhost>`

  Note: system_prefix can be configured, which is particularly useful if the localhost does not have a unique IP address - e.g., your network is using NAT or "Masquerading", or there is no network.

- time_stamp is the result of calling the `time()` in C++ (or `getCurrentTimeMillis()/1000` in Java), that is, it equals the: <time in seconds that the UID generator was initialized in the current process>

3. The function getUIDSuffix() returns a new UID suffix each time it is called.

The default implementation does: `uid_suffix = pid + sequence_number`

- pid is the process id of the calling process - for C++ this is retrieved for each UID, in the event that fork() was called, and the UID prefix (like all other static/global data) is shared by multiple processes.
- sequence_number is an unsigned 32 bit integer that starts at 1, and is incremented after each call to getUIDSuffix().

  *Note: If you generate 2**32 UID's, getUIDSuffix() will detect the sequence_number wrap, and will throw an exception. In this case, makeUID() will re-initialize the uid-prefix (with a new time-stamp), and will re-call getUIDSuffix() which will re-start the sequence-numbers.*

Appendix F: Using Nunit tests with DCF .NET Applications

What Is NUnit?: NUnit is a unit-testing framework for all Microsoft .Net languages, written entirely in C#; NUnit brings xUnit capabilities to all .NET languages. *(See http://www.nunit.org and the next page for additional information.)*

1. Example NUnit test class

Listed below is a sample test class that illustrates the use of NUnit with a DCF related test.

```
namespace OEM_name
{
   namespace SomeTest
   {
      [TestFixture]
      public class SomeTestClass
      {
         //Use these for shutdown listener test.
         private int phase1 = 0;
         private int phase2 = 0;

         #region Setup and Teardown Functions
         [TestFixtureSetUp]
         public void runBeforeTests()
         {
            InitTest.setup();
         }
…
```

Here is the `InitTest` class. This will keep `AppControl` and other common services from being setup multiple times.

```
using System;
using LaurelBridge.DCF;

namespace OEM_name
{
   namespace SomeTest
   {
      public class InitTest
      {
         private static bool isInitialized_ = false;
         public static AppControl apc_;
         public static CFGDB cfgdb_;
         private static string[] args_ = {"-appcfg",
"/apps/defaults/NAppControl_atest"};
         public static void setup()
         {
            if( ! isInitialized_ )
            {
               try
               {
                  LaurelBridge.CDS_a.CFGDB_a.setup(args_);
                  cfgdb_ = CFGDB.Instance;
                  LaurelBridge.APC_a.AppControl_a.setup(args_, CINFO.Instance);
                  apc_ = AppControl.Instance;
               }
               catch ( SystemException e )
               {
                  System.Console.WriteLine( e );
               }
               isInitialized_ = true;
```

```
                    }
                }
            }
        }
    }
```

2. Some Background on NUnit:

The following notes are excerpted from NUnit's web site; links to additional information are included in the text. Please see the NUnit web site (http://www.nunit.org) for the most current information.

What Is NUnit? "NUnit is a unit-testing framework for all .Net languages. The current version, 2.2 is the fourth major release of this xUnit based unit testing tool for Microsoft .NET. It is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, for example custom attributes and other reflection related capabilities. NUnit brings xUnit to all .NET languages." See: http://www.nunit.org/index.html .

Permission is granted to anyone to use the NUnit software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to certain restrictions. See: http://www.nunit.org/license.html . *Portions Copyright © 2002 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov or Copyright © 2000-2002 Philip A. Craig.*

Getting Started with NUnit. Go to the Download page (http://www.nunit.org/download.html), select a version of NUnit and download it. The Installation page (http://www.nunit.org/installation.html) contains instructions for installing on your system.

To get started using NUnit, read the Getting Started document (http://www.nunit.org/files/QuickStart.doc). This article demonstrates the development process with NUnit in the context of a C# banking application. Check the Samples page (http://www.nunit.org/samples.html) for additional examples, including some in VB.Net, J# and managed C++.

Which Test Runner to use? NUnit has two different ways to run your tests. The console runner, nunit-console.exe, (http://www.nunit.org/getStarted.html#console) is the fastest to launch, but is not interactive. The gui runner, nunit-gui.exe, (http://www.nunit.org/getStarted.html#gui) is a Windows Forms application that allows you to work selectively with your tests and provides graphical feedback.

…

Please see the NUnit web site (http://www.nunit.org) for additional and the most current information on this tool.

Appendix G: Using Perl with the DCF

The DCF uses many Perl scripts to simplify and to automate tasks that need to be done in the development, testing, and deployment of an application. To use these scripts, Perl must be installed on your system, and the Perl interpreter must be in your PATH. The Windows DCF includes a version of Perl that can be used if you do not already have Perl installed; most versions of Linux come with Perl already installed.

On Linux, the “shebang” line – “#!” – at the top of the script is sufficient to run the Perl interpreter without explicitly invoking it on the command line. For example, you can type “*`dcfmake.pl`*” instead of “*`perl dcfmake.pl`*”.

On Windows, you need to call the Perl interpreter explicitly – e.g., *`perl -S dcfstop.pl`* – unless there is a Windows file association between the **.pl** file extension and the Perl interpreter. If the file association exists, you may call *dcfstop.pl* and other Perl scripts just by using their names, without invoking the Perl interpreter, e.g., *`dcfstop.pl`*.

If you do not have a Perl file association, the Perl interpreter looks for the script to run in the current directory. You may also specify the script’s path explicitly – e.g., *`perl /home/mydir/myscript.pl`* – or specify the “-S” flag to find the script in the PATH, e.g., *`perl -S dcfstop.pl`*

To create a Perl file association on Windows, enter the following commands in a DOS prompt, such as the DCF Command Prompt:

- *assoc .pl=Perl*
- *ftype Perl=<path to perl interpreter> “%1” %**
    - Example: `ftype Perl=”C:\Program Files\DCF-3.1.1a\perl\bin\perl.exe” “%1” %*`

Note that most of the example invocations of Perl scripts throughout this guide assume that such a file association exists, or the explicit invocation of Perl is omitted for the sake of simplicity and brevity.

Note that on Windows platforms that in order to do I/O redirection the Perl interpreter must be explicitly invoked, e.g. “perl –S dcfmake.pl > make.txt 2>&1”

Appendix H: Customizing the DCF Remote Service Interface

The DCF's web-based remote service interface is fully configurable and customizable by the OEM developer.

Note that you can use different web servers than the DCF's default choice of using Apache 2.2.16. To substitute an alternate web server, you must configure your web server to both support the DCF's CGI scripts and to serve up its static web pages; see Section 2.4.1.1 for more information.

1. Shortcuts for Setting Debug Flags

Within the DCF there are debugging flags that can be enabled or disabled as desired to alter the verbosity of the debugging output to the log files. These flags can be accessed via the "Set Debug Flags" link on the "DCF Home: Operations" page and then selecting the desired application and then the correct component in the application. This is most useful for a developer, who knows details of the DCF system, but it can be confusing for an end user, who would not know about all the components and which ones he needed to choose to get more information.

To meet this need, a utility has been provided that can display shortcuts to the debug flag attributes of a component and can modify those flags. It is called "**sdfgroup_cgi**", so-named because one of its functions it to set debug flags in groups.

This CGI parses the data in a CFGGroup to determine what debugging shortcuts exist. Each shortcut is a CFGGroup with attributes giving information about some debugging flags to modify. Each shortcut group can have any name, but the set of shortcut groups must be in a CFGGroup named **debug_shortcuts**. (The names of the shortcut groups must be unique within the **debug_shortcuts** group – no duplicate names are allowed.) The **debug_shortcuts** group can be in a CFGGroup with any name; usually this is a file with a collection of related debug flag shortcuts. The name of this file (or CFGGroup) is passed as an argument to the CGI, allowing for different debug shortcut configuration files for different circumstances. The data is parsed to produce an HTML form of checkboxes, one for each shortcut group; they are displayed in the order that they are present in the **debug_shortcuts** group. At the bottom of the form that is produced is a checkbox for saving the updated debugs flags in the application's configuration data, for the next time the application is started.

The CGI takes two arguments on its command line: an action to perform, and the name of the debug shortcut configuration file to use. Valid actions are "**select_debug_flags**" and "**set_debug_flags**".

Example:

```
sdfgroup_cgi select_debug_flags  /dpa_dbg_shortcut.cfg
sdfgroup_cgi  set_debug_flags  /dpa_dbg_shortcut.cfg
```

In a web page, the first one would be written as follows:

```
http://host/cgi-bin/sdfgroup_cgi.pl?select_debug_flags+/dpa_dbg_shortcut.cfg
```

If the CGI is being used to set the values of the debug flags, the data is passed to the CGI through its STDIN. (For HTML viewed in a web browser, this means that the data is passed via the POST method.)

1.1. Debug Shortcuts

The debugging shortcut groups are organized into a set in a **debug_shortcuts** CFGGroup. This containing CFGGroup may have an attribute **heading** that will be displayed before all the debug values;

this attribute is optional. The **heading** attribute is useful for describing the overall application/process that will be affected by these debug flags.

Each debug shortcut group has six attributes that must be set.

- **app_name** – This is the name of the CFGGroup for the application whose debug flags should be modified.
- **proc_name** – This is the prefix of the process CFGGroup that is created for an instance of the application, in the /procs directory. (It is a prefix since the process IDs are appended at runtime to the actual groups.)
- **name** – This text is displayed to describe the debug flag shortcut.
- **component_name** – This is the name of the DCF component of the debug flag.
- **component_type** – This is the type of the DCF component for the debug flag.
- **df_name** – This is the actual name of the debug flag that is to be set/unset.

Each shortcut group can also have the optional attribute **df_linked_to**. This attribute can have multiple values. If this attribute is set with the name(s) of another shortcut group, when the first group is selected on the generated web page, the other group(s) will also be selected. This provides a way of suggesting that certain debug flags should be set when other flags are set, while allowing the user to disable those flags if they are not desired.

An example shortcut group is shown below:

```
app_name = /apps/defaults/dcm_switch
proc_name = /procs/dcm_switch
name = Show DICOM Association information
component_name = DCS
component_type = cpp_lib
df_name = df_SHOW_GENERAL_FLOW
df_linked_to = 5
```

These attributes are put together to determine the complete name of the debug flag to set/unset. Please note that when the flag is modified, it can be changed in both the application's configuration file and in *every* process instance of the application that is running – you cannot change the flag in one instance of an application but not another. Checking the **"Save these settings for next time**" box will change the debug flags in the application's configuration data as well as in any matching processes. (The processes' debug flags are *always* changed.) When this group is selected, the shortcut group named "5" will also be selected.

The above example would load the attribute **/components/cpp_lib/DCS/debug_controls/debug_flag**, find the value for the debug flag **df_SHOW_GENERAL_FLOW**, and modify it in the attribute **/apps/defaults/dcf_switch/cpp_lib/DCS/debug_flags** and in **/procs/defaults/dcf_switch.*/cpp_lib/DCS/debug_flags**.

Note that it is possible for the **df_name** value to have multiple debug flags, separated by colons ('**:**'). This allows for multiple flags to be enabled/disabled in one shortcut. One caveat of this is that if a group sets a flag that was unset by a previous group, the flag will be set.

1.2. Viewing the current settings

Running the CGI with the command "**select_debug_flags**" will display a page of checkboxes, one checkbox for each shortcut group. If the flag is set, the box will be checked; it will be empty if the flag is unset. If multiple flags are in a shortcut group, the box will be checked if *any* of them are set. If the flags are set in the application's default configuration data but not in any of the process instances, the flags will be marked set. Similarly, if they are set in a process instance but not in the application's default configuration data, the flags will be marked set. Basically, if they are set anywhere in a matching process or application, the flag is marked set – this is to prevent the illusion that they are *not* set anywhere.

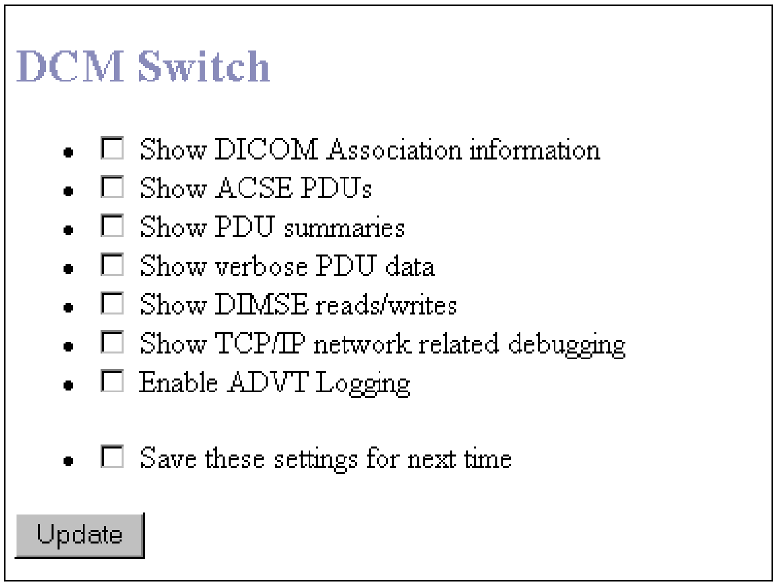
Clicking a set checkbox will unset the shortcut group and its corresponding flags. Clicking an empty checkbox will set the shortcut group and its flags. Pressing the “Update” button at the bottom of the form will submit the new values to be processed. After the data is processed and the new flags are updated, the results page will show which shortcut groups have been enabled and which were disabled.

2. Example Debug Shortcut File

Below is an example debug shortcut file and its various groups and their flags. Underneath are images showing the form that a user sees for setting the flags, and then the results. Since none of the boxes were checked, all the corresponding flags were disabled. (The beginning of each shortcut group has been highlighted to make it easier to find.) Note that group 5 has multiple flags being affected by one shortcut, while selecting group 4 will also select groups 1 and 3.

```
[ debug_shortcuts ]
heading = DCM Switch

[ debug_shortcuts/7 ]
app_name = /apps/defaults/dcf_switch
proc_name = /procs/dcf_switch
name = Show DICOM Association information
component_name = DCS
component_type = cpp_lib
df_name = df_SHOW_GENERAL_FLOW

[ debug_shortcuts/1 ]
app_name = /apps/defaults/dcf_switch
proc_name = /procs/dcf_switch
name = Show ACSE PDUs
component_name = DCS
component_type = cpp_lib
df_name = df_DUMP_ACSE

[ debug_shortcuts/3 ]
app_name = /apps/defaults/dcf_switch
proc_name = /procs/dcf_switch
name = Show PDU summaries
component_name = DCS
component_type = cpp_lib
df_name = df_DUMP_PDATA

[ debug_shortcuts/4 ]
app_name = /apps/defaults/dcf_switch
proc_name = /procs/dcf_switch
name = Show verbose PDU data
component_name = DCS
component_type = cpp_lib
df_name = df_DUMP_PDATA_VERBOSE
df_linked_to = 1
df_linked_to = 3

[ debug_shortcuts/5 ]
app_name = /apps/defaults/dcf_switch
proc_name = /procs/dcf_switch
name = Show DIMSE reads/writes
component_name = DCS
component_type = cpp_lib
df_name = df_SHOW_DIMSE_READ:df_SHOW_DIMSE_WRITE
```

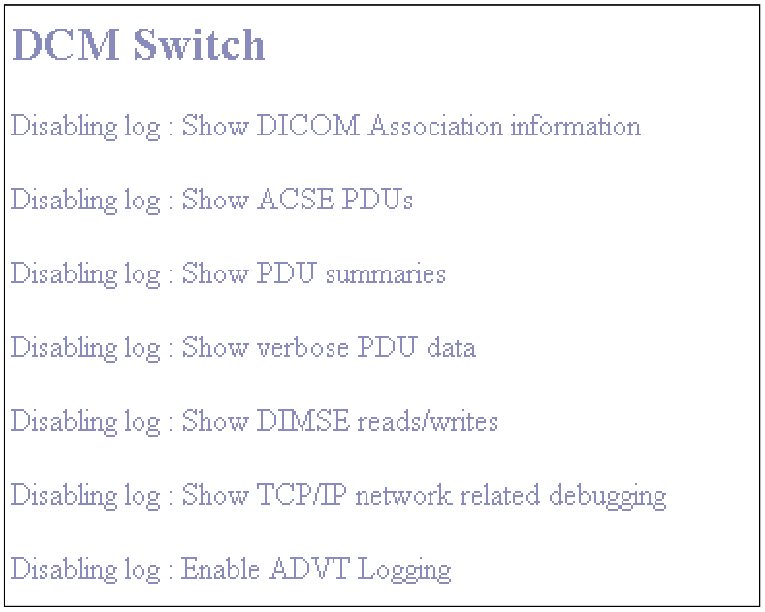
```
df_linked_to = 1

[ debug_shortcuts/6 ]
app_name = /apps/defaults/dcf_switch
proc_name = /procs/dcf_switch
name = Show TCP/IP network related debugging
component_name = DCS
component_type = cpp_lib
df_name = df_TCP_NETWORK
df_linked_to = 5

[ debug_shortcuts/2 ]
app_name = /apps/defaults/dcf_switch
proc_name = /procs/dcf_switch
name = Enable ADVT Logging
component_name = DCS
component_type = cpp_lib
df_name = df_LOG_ADVT_FORMAT
```


DCM Switch

- ☐ Show DICOM Association information
- ☐ Show ACSE PDUs
- ☐ Show PDU summaries
- ☐ Show verbose PDU data
- ☐ Show DIMSE reads/writes
- ☐ Show TCP/IP network related debugging
- ☐ Enable ADVT Logging

- ☐ Save these settings for next time

Figure 17: Displayed Debug shortcuts.


DCM Switch

Disabling log : Show DICOM Association information

Disabling log : Show ACSE PDUs

Disabling log : Show PDU summaries

Disabling log : Show verbose PDU data

Disabling log : Show DIMSE reads/writes

Disabling log : Show TCP/IP network related debugging

Disabling log : Enable ADVT Logging

Figure 18: After Clicking the Update button.

3. CDS Configuration Shortcuts

Configuring the DCF, with its many configuration groups and attributes, is not always the simplest of tasks. There are many attributes that may need to be modified, and often it would be useful to set many attributes on one page rather than navigating through the CDS tree to each attribute and modifying them individually. To this end the CGI script **cds_cgi** has been provided. It parses a configuration group of CDS shortcuts – these are shortcuts to the attributes to be modified – and displays the attributes specified in an HTML form for modification. The CGI also parses the results of the form to update the attributes in the CDS database.

The CGI parses the data in a CFGGroup to determine what the shortcuts are and how to display them. The shortcut groups must be in a CFGGroup with the name **cds_shortcuts**. The **cds_shortcuts** group can be in a group with any name; usually this is a file with a collection of related attributes that should be configured at the same time. The name of the file (or containing CFGGroup) can be passed as an argument to the CGI, allowing for different shortcut configuration files to produce different HTML pages.

The CGI takes two arguments on its command line: an action to perform, and the name of the shortcut configuration file to use. Valid actions are "**view_attributes**" and "**set_attributes**".

Example:

```
cds_cgi  view_attributes  /dpa_config.cfg
cds_cgi  set_attributes  /dpa_config.cfg
```

If the CGI is being used to set the values of attributes, the data is passed to the CGI through its STDIN. (For HTML viewed in a web browser, this means that the data is passed via the POST method.)

3.1. Shortcut Attributes

CDS shortcut groups can have many attributes, with varying effects.

There are two attributes that are mandatory – they are required by all CDS shortcut groups.

1. **type** – This is the type of the shortcut, and determines how the group will be processed and displayed.
2. **desc** – This is the text that will be displayed for the shortcut; it describes the shortcut.

Most groups will also require the attribute **attribute_name** – this is the full path of the CDS attribute to be retrieved/modified.

As each group is parsed, it will be put into a bulleted list of shortcuts on the generated page. Each shortcut will be put on the page in the order that it is in the **cds_shortcuts** group. Each group can have any name, but the name must be unique in the **cds_shortcuts** group.

3.1.1. Types of shortcuts

3.1.1.1. title

This type of group is used to display a heading on the page. The description ("**desc**") attribute of the group will be displayed in an <H2> tag. This item will not be bulleted, but it will be indented since it is contained in the enclosing <UL> tag for the list.

Required attributes: type, desc

Example:

```
type = title
desc = DPA Configuration Data
```

When the data is updated, this will be displayed on the page, but it cannot be modified.

3.1.1.2. html

This type of group is used to display HTML on the page, allowing the user to add some customization to the display of the page. The text of the HTML to be inserted into the page is contained in the **desc** attribute – note that this means that the lines of HTML text must be continued with a backslash ("\") if there is more than one line to display. This item will not be bulleted, but it will be indented since it is contained in the enclosing <UL> tag for the list.

Required attributes: type, desc

Example:

```
type = html
desc =    <hr>\
   <center><img src="/world1.gif"></center> \
   <hr>
```

This example will display the image file **world1.gif**, centered between two lines.

When the data is updated, this will be displayed on the page, but it cannot be modified.

3.1.1.3. display_only

This type of group retrieves the value of a specified attribute from the CDS database and displays it in <PRE> tags on the page.

Required attributes: type, desc, attribute_name

Example:

```
type = display_only
desc = DPA TCP port
```

```
attribute_name = /apps/defaults/dcf_switch/cpp_lib/DCS/AssociationManager/tcp_port
```

When the data is updated, this will be displayed on the page, but it cannot be modified.

3.1.1.4. integer or string

The integer and string types are currently handled identically, although it may later be possible to handle them differently. These groups must specify the **attribute_name** to retrieve/modify. These also have the additional attribute of **multiplicity**, indicating if the attribute can have multiple values or not – the possible multiplicity values are “single” and “multiple”. The data from these types will be displayed in a single-line text field or in a multiple-line text area, depending on the **multiplicity** value. If the multiplicity is single, the attribute **width** must also be specified – this will be the width of the text field displaying the value. If the multiplicity is multiple, the attributes **width** and **height** must be specified – these are the columns and rows of the text area displaying the value.

Required attributes:

type, desc, attribute_name, multiplicity, width

– or –

type, desc, attribute_name, multiplicity, width, height

Example:

```
type = integer
desc = DPA TCP port
attribute_name = /apps/defaults/dcf_switch/cpp_lib/DCS/AssociationManager/tcp_port
multiplicity = single
width = 5
```

3.1.1.5. boolean

This type will display a checkbox to simplify enabling/disabling the attribute. This type also requires the attributes **on_value** and **off_value** – these are used to determine if the checkbox should be checked when the value is displayed, and what the value should be set to when the box is changed. (A Boolean value cannot have multiple values, so multiplicity is ignored.)

Required attributes: type, desc, attribute_name, on_value, off_value

Example:

```
attribute_name = /apps/defaults/dcf_switch/cpp_app/dcf_switch/enable_statistics
type = boolean
desc = Enable statistics
on_value = yes
off_value = no
```

3.1.2. Optional attributes:

Each group may also have the optional attribute **read_only**. If this attribute is present and has the value “true”, then the shortcut will be displayed normally (i.e., with a checkbox or text field), but the input for the attribute will be disabled in the browser window and the data cannot be modified. (If the browser does not recognize the disabling of the input [as older browsers may do], the data will be ignored and not updated when the form is processed.)

3.1.3. Synchronizing attributes:

There may be times when you want several CFGDB attributes to have the same value but you don't want to have to set each one manually – you want to set one attribute and have the rest get the same value. This can be done by adding the optional attribute **sync_attributes** to the group. This can be a multi-valued attribute, whose values are the complete names of the attributes to keep in sync with the primary attribute (**attribute_name** above).

Consider this example:

```
type = integer
desc = DPA TCP port
attribute_name = /apps/defaults/dcf_switch/cpp_lib/DCS/AssociationManager/tcp_port
multiplicity = single
width = 5
sync_attributes = /apps/defaults/dcf_echo_scp/cpp_lib/DCS/AssociationManager/tcp_port
sync_attributes = /components/cpp_lib/DCS/DCS/AssociationManager/tcp_port
```

This will cause both CFGDB attributes specified by `sync_attributes` to get the same value as the primary attribute – if the main one is set to 20000, those two will also be set to 20000 at the same time.

If you are setting a Boolean attribute, you have additional options that you can specify: **sync_on_value** and **sync_off_value**. If the Boolean attribute is set, then the `sync_on_value` is used for any `sync_attributes` specified; if the Boolean attribute is unset, then the `sync_off_value` will be used. If either of `sync_on_value` or `sync_off_value` is unspecified, then the `sync_attributes` will get the same value as the primary attribute.

Consider this example:

```
attribute name = /apps/defaults/dcf_switch/cpp_app/dcf_switch/enable_statistics
type = boolean
desc = Enable statistics
on_value = yes
off_value = no
sync_attributes = /apps/dcf_switch/dcf_switch/cpp_app/dcf_switch/enable_statistics
```

Since `sync_on_value` and `sync_off_value` are not specified, the `sync_attributes` will be turned on and off exactly the same as the primary attribute.

Consider another example:

```
attribute_name = /apps/defaults/dcf_switch/cpp_app/dcf_switch/enable_statistics
type = boolean
desc = Enable statistics
on_value = yes
off_value = no
sync_attributes = /apps/dcf_switch/dcf_switch/cpp_app/dcf_switch/enable_statistics
sync_on_value = false
sync_off_value = true
```

In this case, the `sync_attributes` will be set to "false" when the primary attribute is set to "yes", or to "true" when the primary is set to "no". Any values can be used – the values for the `sync_attributes` could be "1" and "5", "Peter" and "Mary", or whatever you desire, set in sync with the primary attribute. This capability also allows you to toggle settings together – as you can see in this example, one attribute is set to a positive value when the other is negative, and vice versa.

3.2. Automatic Shortcut Generation

While there are situations in which a custom-generated shortcut configuration file is good and necessary, there are many situations in which the attributes to be configured are very similar from one startup instance to another. Such is the case with the set of servers that are started up by the DCF's startup scripts. Depending on what configuration is started, certain processes are started and should be configured. Rather than require a custom shortcut file be written for each startup configuration, a Perl module (**DCFGenStartCfg.pm**) is available to write the shortcut file as the system is being started. The module parses the startup configuration and determines what servers are being started and automatically generates a shortcut configuration file for that startup configuration.

There are nine attributes that are commonly set in each server that is started, and these are put in the shortcut configuration file.

1. *TCP port* – cpp_lib/DCS/AssociationManager/tcp_port
2. *Server host address* – cpp_lib/DCS/AssociationManager/server_host_address
3. *Maximum number of associations* – cpp_lib/DCS/AssociationManager/max_concurrent_associations
4. *Association idle timeout* – cpp_lib/DCS/association/association_idle_timeout_seconds
5. *First PDU read timeout* – cpp_lib/DCS/AssociationManager/first_pdu_read_timeout
6. *PDU read timeout* – cpp_lib/DCS/association/pdu_read_timeout
7. *Maximum PDU receive length* – cpp_lib/DCS/association/max_pdu_receive_length
8. *Max num of log files* – java-or-cpp_lib/LOG_a/outputs/file_output_1/max_files
9. *Max size per log file* – java-or-cpp_lib/LOG_a/outputs/file_output_1/max_size

This list is easily viewed at the top of DCFGenStartCfg.pm should the list need to be changed. Each server is also given a title attribute within the shortcut file, to clearly indicate what server is being configured by each set of attributes.

As the startup configuration is parsed, the application configuration group for each server is determined, and shortcuts are set up for the common attributes in that appconfig. If an attribute does not exist, the shortcut is omitted.

Some servers may have attributes that need to be set but that are not part of the common nine listed above. In this case, special parameters may be set up in the server's per-instance application configuration group describing these attributes; these are listed as multiple values of the "*special_params*" attribute. For example, special parameters for the DLOG_Server, using its default appconfig, would be in the configuration attribute

/apps/defaults/DLOG_Server/java_app/DLOG_Server/special_params.

Each value should be a semi-colon separated list of items describing the desired shortcut, in this order: `type, description, subgroup, attribute_name, multiplicity, width, height, on_value, off_value, read_only`. Subgroup and attribute_name are combined by the module to create the complete name of the attribute to be configured; the values of the other fields are as described earlier in this document. If a field is not applicable to a certain shortcut type, it may be omitted and will be ignored, but its place should be indicated by the semi-colon separator.

Example:

```
special_params = integer;TCP Port;
java_app/DLOG_Server/output_configuration_info;
server_port_number;single;6;;;;
```

(Note: All of this text would be on one line.)

In addition, each server group in the startup configuration file may have special parameters that should be configured for that specific startup configuration. This is indicated via the *special_params* attribute in the server's startup group. The format is the same as for the other *special_params* attribute, except that this one has the name of the appconfig to use at the beginning.

Example:

```
special_params = /apps/defaults/DCDS_Server;boolean;
Enable upward notifications;java_app/DCDS_Server;
enable_upward_notifications;single;;;YES;NO;true
```

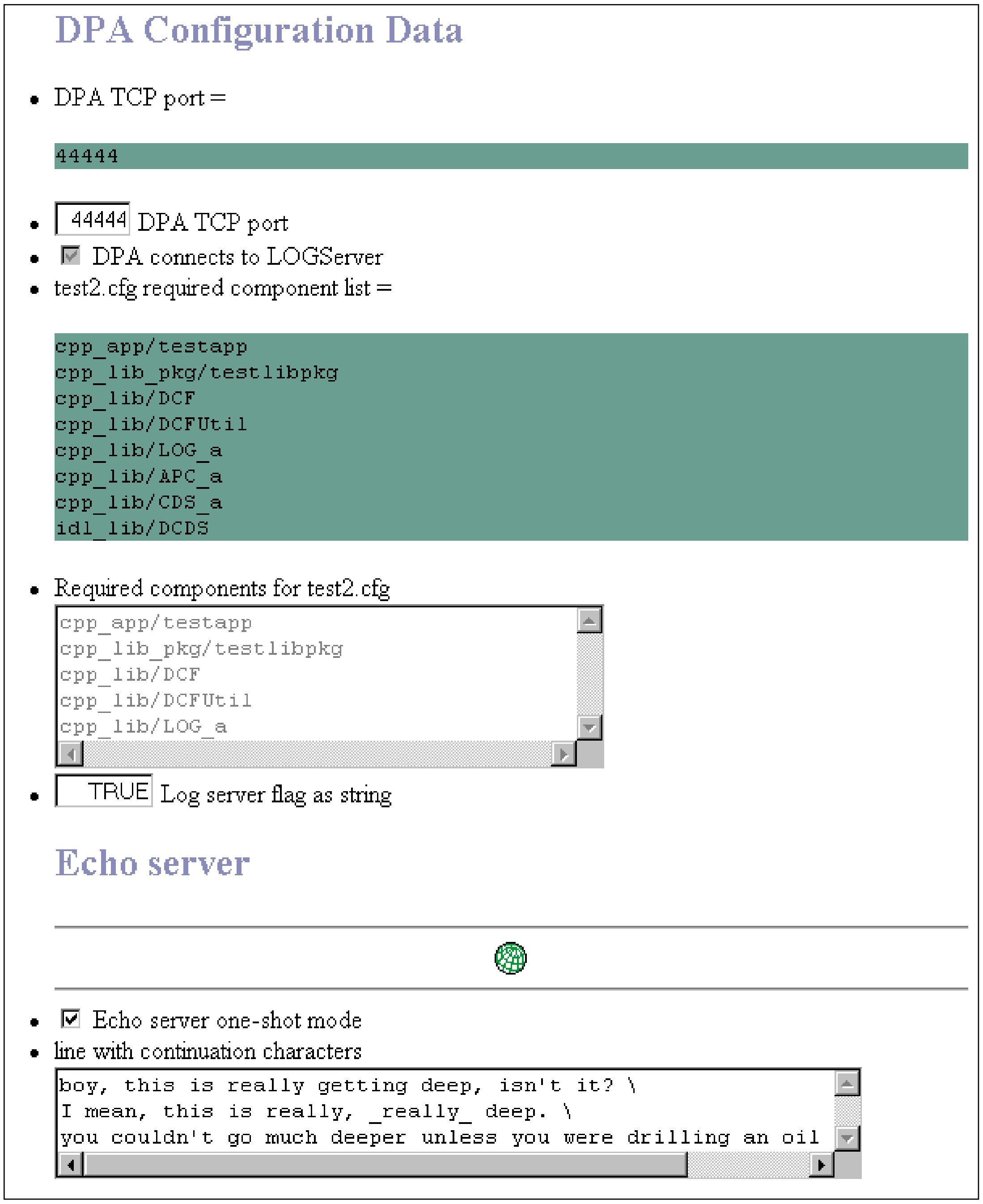
The shortcut configuration file is generated upon startup of the system, and a link is provided on the DCF Operations page to configure the servers via this shortcut file.

3.3. Example shortcut configuration file:

The results of using this shortcut configuration file are shown below.

```
[ cds_shortcuts ]
[ cds_shortcuts/title_1 ]
type = title
desc = DPA Configuration Data
[ cds_shortcuts/0 ]
attribute_name = /apps/defaults/dcf_switch/cpp_lib/DCS/AssociationManager/tcp_port
type = display_only
desc = DPA TCP port

[ cds_shortcuts/1 ]
attribute_name = /apps/defaults/dcf_switch/cpp_lib/DCS/AssociationManager/tcp_port
multiplicity = single
type = integer
desc = DPA TCP port
width = 5
[ cds_shortcuts/2 ]
attribute_name = /apps/defaults/dcf_switch/cpp_lib/LOG_a/use_log_server
type = boolean
read_only = true
desc = DPA connects to DLOG_Server
on_value = true
off_value = false
[ cds_shortcuts/3a]
attribute_name = /test2.cfg/required_components/component
type = display_only
desc = test2.cfg required component list
[ cds_shortcuts/3 ]
attribute_name = /test2.cfg/required_components/component
multiplicity = multiple
type = string
read_only = true
desc = Required components for test2.cfg
width = 40
height = 5
[ cds_shortcuts/log_server_flag ]
attribute_name = /apps/defaults/dcf_switch/cpp_lib/LOG_a/use_log_server
multiplicity = single
type = string
desc = Log server flag as string
width = 7
[ cds_shortcuts/title_2 ]
```

```
type = title
desc = Echo server
[ cds_shortcuts/html_1 ]
type = html
desc = <hr> \
<center><img src="/world1.gif"></center> \
<hr>
[ cds_shortcuts/echo_server_mode ]
attribute_name = /apps/dcf_switch/dcf_echo_scp/cpp_app/dcf_echo_scp/one_shot_mode
type = boolean
desc = Echo server one-shot mode
on_value = yes
off_value = no
[ cds_shortcuts/continued_line_test ]
desc = line with continuation characters
attribute_name =
/test3.cfg/testapp/another_group/level3/level4/level5/level6/level7/level8/level9/attr1
type = string
multiplicity = single
width = 60
```



DPA Configuration Data

- DPA TCP port =

```
44444
```

- 44444 DPA TCP port
- ✓ DPA connects to LOGServer
- test2.cfg required component list =

```
cpp_app/testapp
cpp_lib_pkg/testlibpkg
cpp_lib/DCF
cpp_lib/DCFUtil
cpp_lib/LOG_a
cpp_lib/APC_a
cpp_lib/CDS_a
idl_lib/DCDS
```

- Required components for test2.cfg

```
cpp_app/testapp
cpp_lib_pkg/testlibpkg
cpp_lib/DCF
cpp_lib/DCFUtil
cpp_lib/LOG_a
```

- TRUE Log server flag as string

Echo server

- ✓ Echo server one-shot mode
- line with continuation characters

```
boy, this is really getting deep, isn't it? \
I mean, this is really, _really_ deep. \
you couldn't go much deeper unless you were drilling an oil
```

Figure 19: Display results of applying a shortcut configuration file.

4. Authenticating Access to DCF Web Pages

As you develop and use the DCF and its web pages, you will realize how powerful the web interface is and how easy it could be for an untrained user to change its settings and disrupt the operation of the DCF and your application. During development of your application, this may be less of a concern, but it could be an issue in the field once your application is deployed, if it is deployed with a web interface. In such situations, you might want to configure the DCF's web server to require users to "log in" before being able to access the web pages. This would restrict access to only those users that have the appropriate permissions; it is also possible to configure the access restrictions so that some users could access certain sections but not other sections.

The steps described below will allow you to require authentication to access the DCF's web pages.

(Please note that the steps described have been tested for Apache **1.3.33 and 2.2.16**; if you are using a different web server, the steps will be similar but not identical.)

Steps to require authentication to access DCF web pages:

1. Create the password file.

   This is done via the **htpasswd** utility, provided by Apache when you installed it.

   To see the usage, at a command prompt type:

   ```
   htpasswd
   ```

   You must create the password file to use if it does not already exist. From **outside** the *httpd* docs tree (for example, at DCF_ROOT), run

   ```
   htpasswd -c %DCF_ROOT%\dcf_passwords <username>
   ```

   This will create the password file "**dcf_passwords**". (The file should be created outside the web server root so that it is not accidentally served up by the web server.)

   You will be prompted for a password for the user, and required to type it twice. Once the password file exists, you should omit the "-c" flag. (The "-c" flag is only for creating the file the first time; if you use it and the file exists, then you will create a new file, overwriting the existing one, and you will lose any user info in the original file.)

   For subsequent users that need to be added or to change the passwords of existing users, run the command

   ```
   htpasswd %DCF_ROOT%\dcf_passwords <username>
   ```

   (Note that the "-c" option is not used.)

2. Configure Apache to use the file for authenticating users.

   Create an "**`.htaccess`**" file in the directory you wish to protect. If you wish to require authentication for all DCF web pages, you might put an `.htaccess` file in both the `%DCF_ROOT%/httpd/html` and `%DCF_ROOT%/httpd/cgi-bin` directories.

   The contents of the file would look something like this:

   ```
   AuthType Basic
   AuthName "Access restricted to authorized DCF users"
   AuthUserFile %DCF_ROOT%/dcf_passwords
   Require valid-user
   ```

   This specifies that authentication is required and that the `dcf_passwords` file should be checked for the authorized usernames and passwords. Note that you can change the **AuthName** value to whatever text you want to be displayed in the password entry dialog box that will pop up when a user is required to log in. You should specify the complete value for "%DCF_ROOT%", not just assuming that Apache will recognize the environment variable, and you may need to enclose the value in quotes if it has spaces.

3. Modify the httpd.conf file (in %DCF_ROOT%/httpd/conf) to allow the .htaccess file to override some options.

You should do this for each "<Directory>" section in the httpd.conf file that has a directory you wish to protect. Thus, to require authentication for all DCF web pages, you would modify the "<Directory>" section in the file that corresponds to *%DCF_ROOT%/httpd/html*, and change the value **AllowOverride** in those sections from "none" to "AuthConfig".

If you wish to restrict access to the CGI scripts, find the "<Directory>" section for the **cgi-bin** directory (%DCF_ROOT%/httpd/cgi-bin) and change **AllowOverride** from **None** to **AuthConfig**.

4. Stop, then restart the Apache server.
   From a DCF Command Prompt, run the command
   ```
   perl kill_apache.pl
   ```
   This will (obviously) stop the DCF's Apache web server. Restart the server by running the command
   ```
   perl run_apache.pl
   ```

For more information about authenticating users and additional measures you can use, see

http://httpd.apache.org/docs/2.2/howto/auth.html

For example, this page can show you how to configure the Apache web server so that it can be accessed only from your internal network.

5. Java Applet Issues

Java applets are used to provide part of the functionality of the DCF. They are used for various tasks, including editing the filter sets and viewing the log output in real-time. These applets provide a simple interface that can be accessed from anywhere that has web access to the system where the DCF is installed and running. While you will typically configure the DCF while sitting at the computer where it is running, you don't *have* to: you can configure and control the DCF from any computer with a web browser that has network access to the DCF's host.

In order to use the applets, you ***must*** have the Java plug-in installed on your system, as well as in the Web browser you are using. The plug-in is needed because some advanced Java components are used that may not be part of the default Java VM in some browsers, especially older ones. The Java plug-in is designed to provide consistency across web browsers. It can be downloaded from Sun's Java site, at http://www.java.com as part of the Java Runtime Environment (JRE). Since it is part of the JRE, it may already be installed if you have that on your system. If you do not have the plug-in installed, when you try to start an applet all you will probably see is an unchanging gray box with the text "`Loading Java applet`". In this case, please download and install the plug-in; you may need to restart your browser for this change to take effect.

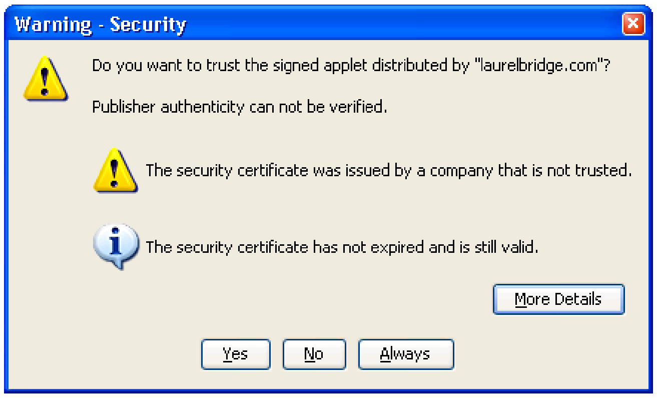
Browser Note: If you are using a Firefox web browser, as of Firefox 3.6, you must use the Next Generation Java Plug-in. This comes with Java 6 Update 10 (1.6.0_10) and higher and should be automatically installed into your browser when the JDK or JRE is installed, although you may need to enable it in your browser.

Some firewalls (such as the default firewall included with Windows) may warn when a web browser tries to run one of the DCF's applets; they may even block the applet from running. If this happens, you should register your browser as an exception with the firewall to unblock access to the browser/applet and to allow it to function normally.

The Java applets used by the DCF use a signed JAR file. This is due to a known bug in Java's CORBA implementation for applets, which causes a security exception to be thrown. (See the Bug IDs in Sun's bugs database: 6203567, 5031209) At this time, when a user downloads or runs a DCF applet in their web browser, the user will be prompted to accept the signed applet from "laurelbridge.com" (see the example figure below).

Some applets in the DCF may cause you to be prompted to accept the signed applet. Most browsers will let users view the certificate for the JAR file before accepting it. If the users select "always", they won't be prompted again about accepting applets in that JAR file. Some users may be concerned that accepting the applets will open up security holes on their system – this is not the case, as the applets make no changes to the local box.

Note: you may have to add a security exception for "`http://<hostname>:8080`" via the Java console Security tab to allow an applet to run on the system.


Warning - Security

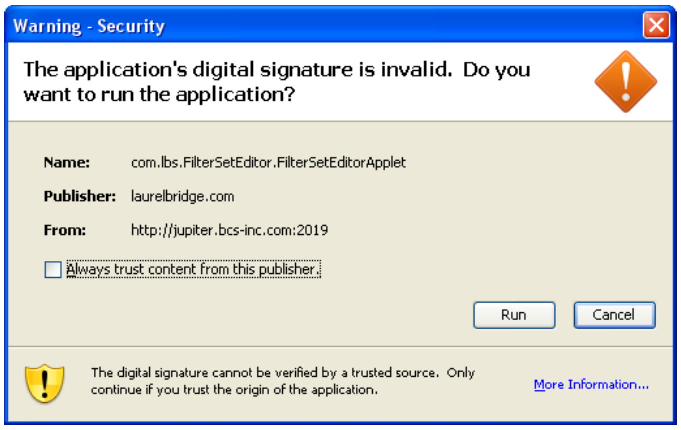
Do you want to trust the signed applet distributed by "laurelbridge.com"?

Publisher authenticity can not be verified.

The security certificate was issued by a company that is not trusted.

The security certificate has not expired and is still valid.

Figure 20: Example security warning from Firefox for a signed JAR file


Warning - Security

The application's digital signature is invalid. Do you want to run the application?

Name: com.lbs.FilterSetEditor.FilterSetEditorApplet
Publisher: laurelbridge.com
From: http://jupiter.bcs-inc.com:2019

☐ Always trust content from this publisher.

The digital signature cannot be verified by a trusted source. Only continue if you trust the origin of the application.

Figure 21: Example security warning from Internet Explorer for a signed JAR file

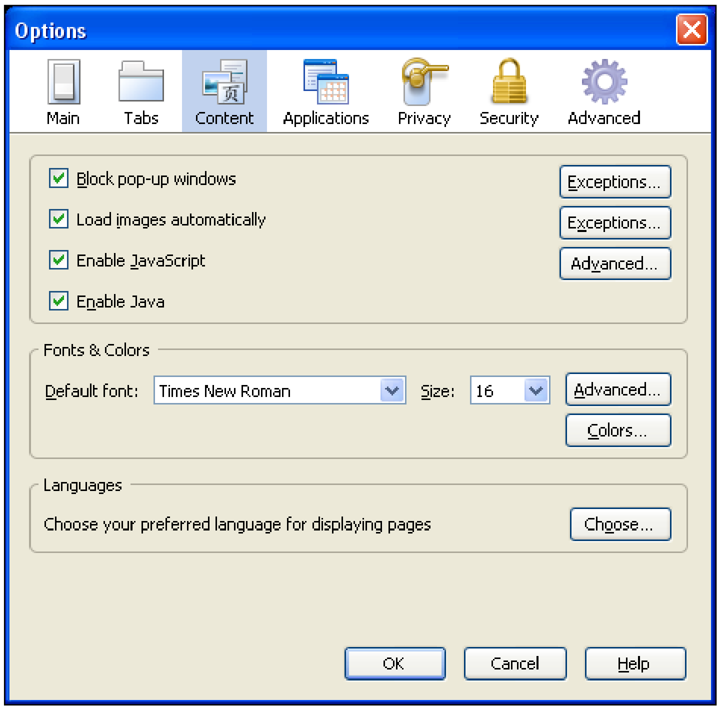
6. Other Applet Issues

If the applets are not working for you, or maybe they work for you when connecting on one computer but not from another, there are many possible problems – Java, firewall settings, plug-in issues, etc.

6.1. Enabling Java

If the applets aren't working for you, one possibility is that Java is disabled in your web browser. You should re-enable it (you may have to restart your browser) and try them again.

On Firefox, the panel for controlling Java may look like this:



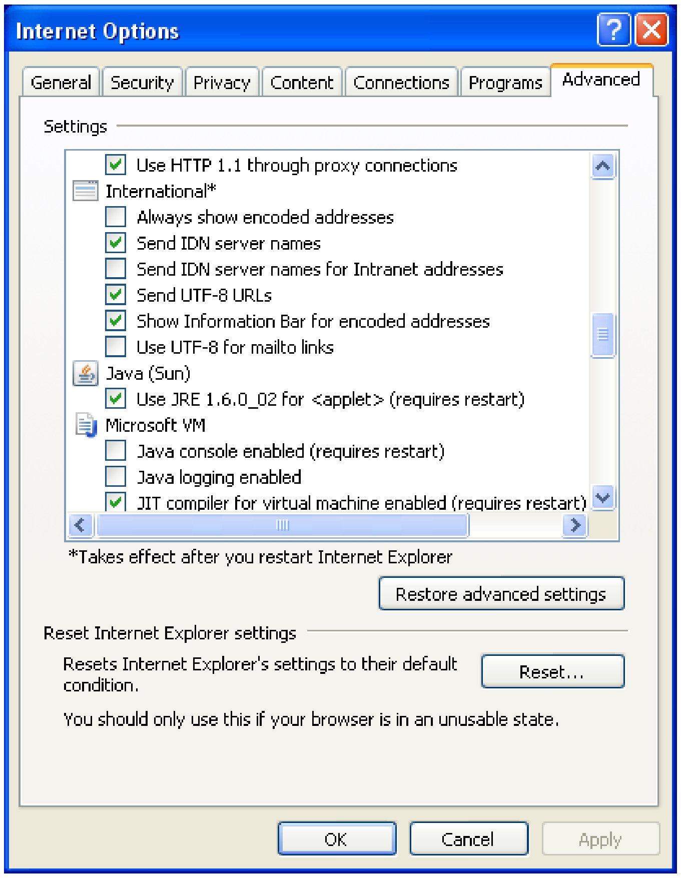
☑ Block pop-up windows
☑ Load images automatically
☑ Enable JavaScript
☑ Enable Java

Default font: Times New Roman	Size: 16

Choose your preferred language for displaying pages

Figure 22: Enabling Java in Firefox

On Internet Explorer, the panel to enable Java may look like this:



- [✓] Use HTTP 1.1 through proxy connections
- International*
  - [ ] Always show encoded addresses
  - [✓] Send IDN server names
  - [ ] Send IDN server names for Intranet addresses
  - [✓] Send UTF-8 URLs
  - [✓] Show Information Bar for encoded addresses
  - [ ] Use UTF-8 for mailto links
- Java (Sun)
  - [✓] Use JRE 1.6.0_02 for <applet> (requires restart)
- Microsoft VM
  - [ ] Java console enabled (requires restart)
  - [ ] Java logging enabled
  - [✓] JIT compiler for virtual machine enabled (requires restart)

*Takes effect after you restart Internet Explorer

Resets Internet Explorer's settings to their default condition.

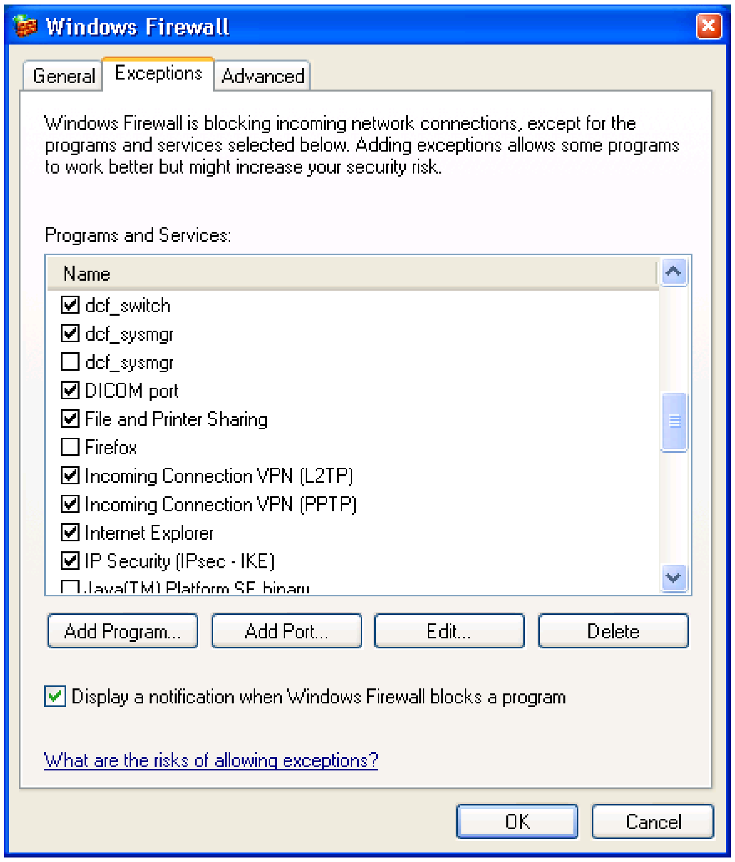
You should only use this if your browser is in an unusable state.

Figure 23: Enabling Java in Internet Explorer

A similar issue could be that you have an older version of the Java plug-in, one that does not have the correct components for running the DCF's applets. The DCF is designed to run with Java 6 Update 2 (formerly called "1.6.0_02") or newer (as seen in the above image). If you have an older version, you should download a newer version from http://www.java.com/ and install it.

6.2. Firewall Issues

Another possibility if the applets are not working for you – especially if you are trying to connect from a remote box – is that your firewall is configured in such a way that the applets cannot run correctly. (It is even possible that the firewall may block all communication between the remote applets and the DCF.) This could include that the firewall is set up to block Java from downloading, or that it needs to be configured to allow the DCF's applications (including Java) to run and to communicate, or they could be blocking communications between the applets and the DCF's services. While you should not disable your firewall, you should configure it to allow the DCF and its applications to communicate by setting them as exceptions. For example, below is a screenshot of what this may look like for the default Windows Firewall. (Note that the example below shows that the `dcf_switch` application has been enabled as an exception but that Java has not *yet* been enabled.)



Windows Firewall is blocking incoming network connections, except for the programs and services selected below. Adding exceptions allows some programs to work better but might increase your security risk.

Programs and Services:

Name
☑ dcf_switch
☑ dcf_sysmgr
☐ dcf_sysmgr
☑ DICOM port
☑ File and Printer Sharing
☐ Firefox
☑ Incoming Connection VPN (L2TP)
☑ Incoming Connection VPN (PPTP)
☑ Internet Explorer
☑ IP Security (IPsec - IKE)
☐ Java(TM) Platform SE binary

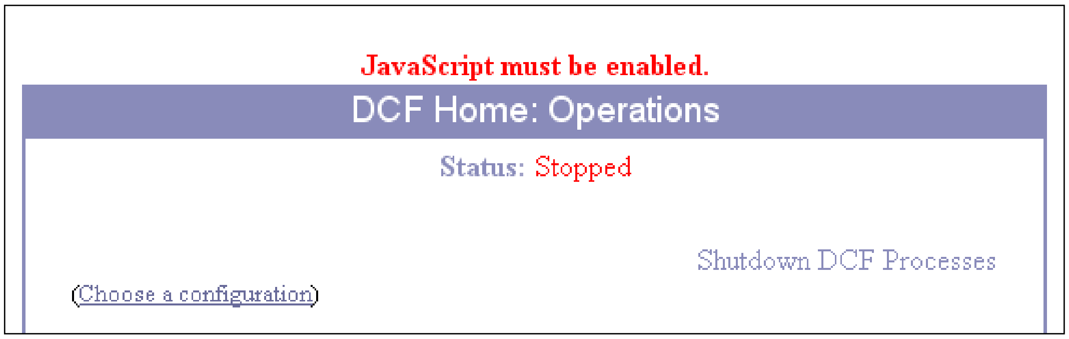
☑ Display a notification when Windows Firewall blocks a program

What are the risks of allowing exceptions?

Figure 24: Setting firewall exceptions

6.3. JavaScript Issues

JavaScript must be enabled to use the DCF and its web pages properly. If JavaScript is not enabled in your browser, you may see error messages like those shown below:


JavaScript must be enabled.

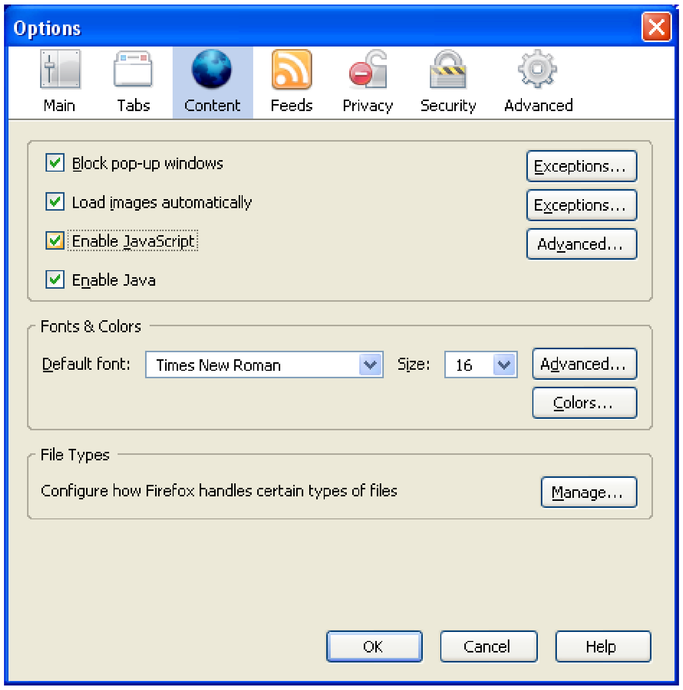
DCF Home: Operations

Status: Stopped

Figure 25: Example JavaScript warning

Each web browser has its own way of enabling and disabling JavaScript, so consult the documentation and their user manuals for specific information on how to enable JavaScript.

In newer versions Firefox enables JavaScript by default, for older versions of Firefox, select the “Tools” menu, then the “Options” sub-menu. Select the “Content” tab and click “Enable JavaScript”.



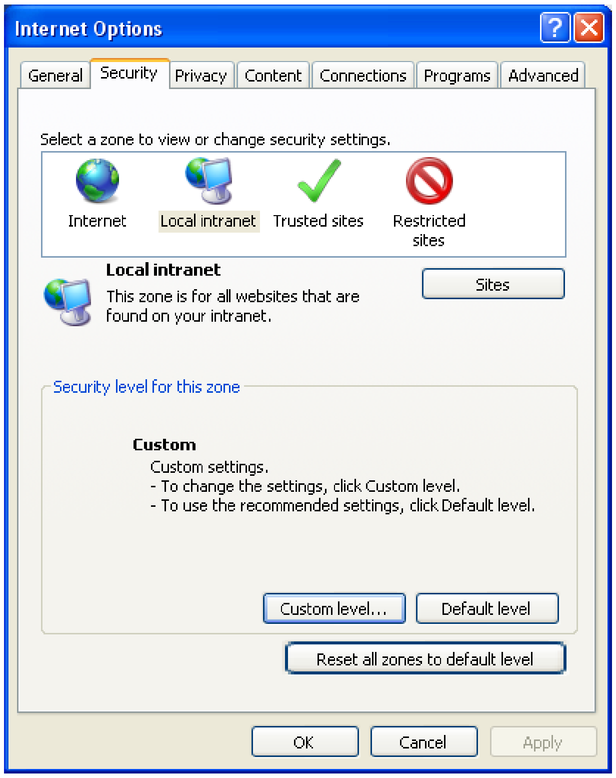
[✓] Block pop-up windows
[✓] Load images automatically
[✓] Enable JavaScript
[✓] Enable Java

Fonts & Colors
Default font: Times New Roman Size: 16

File Types
Configure how Firefox handles certain types of files

Figure 26: Enabling JavaScript in Firefox

For Internet Explorer consulte the documentataion for your current version. For most versions, select the “Tools” menu and then the “Internet Options” sub-menu. Select the “Security” tab, and then select the “Local intranet” zone. Click “Custom level…”; find the “Scripting” section near the bottom, and click “Enable” for “Active scripting”.

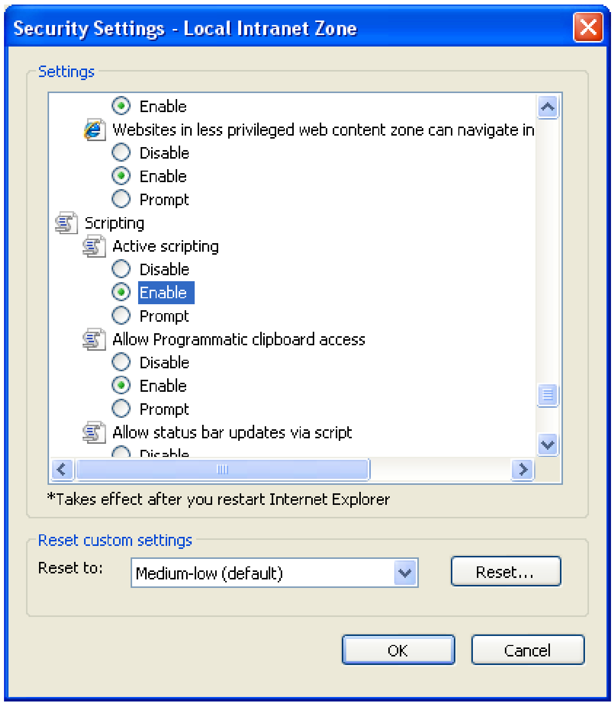


Select a zone to view or change security settings.

Local intranet
This zone is for all websites that are found on your intranet.

Security level for this zone

Custom
Custom settings.
- To change the settings, click Custom level.
- To use the recommended settings, click Default level.



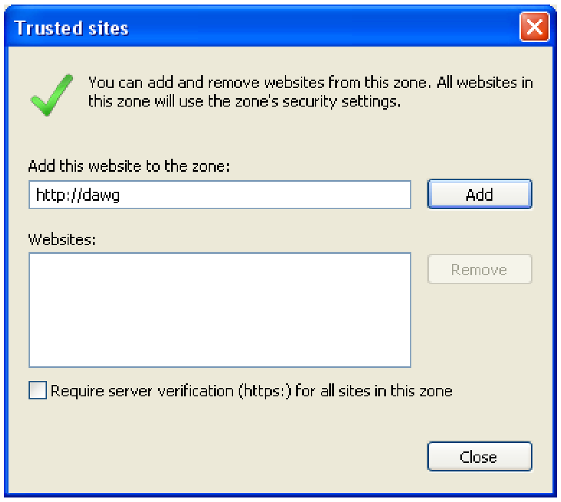
Settings
- [✓ Enable]
- Websites in less privileged web content zone can navigate in
  - Disable
  - [✓ Enable]
  - Prompt
- Scripting
  - Active scripting
    - Disable
    - [✓ Enable]
    - Prompt
  - Allow Programmatic clipboard access
    - Disable
    - [✓ Enable]
    - Prompt
  - Allow status bar updates via script
    - Disable

*Takes effect after you restart Internet Explorer

Reset custom settings
Reset to: Medium-low (default)

Figure 27: Enabling JavaScript in Internet Explorer

Alternatively, you can select the "Trusted sites" zone. Click the "Sites button", and then click "Add" to add http`://<your computer name>` to the list of trusted sites. You may wish to check that "Active scripting" is enabled for the "Trusted sites" zone, following the steps described above for the "Local intranet" zone.



You can add and remove websites from this zone. All websites in this zone will use the zone's security settings.

Add this website to the zone: http://dawg

Websites:

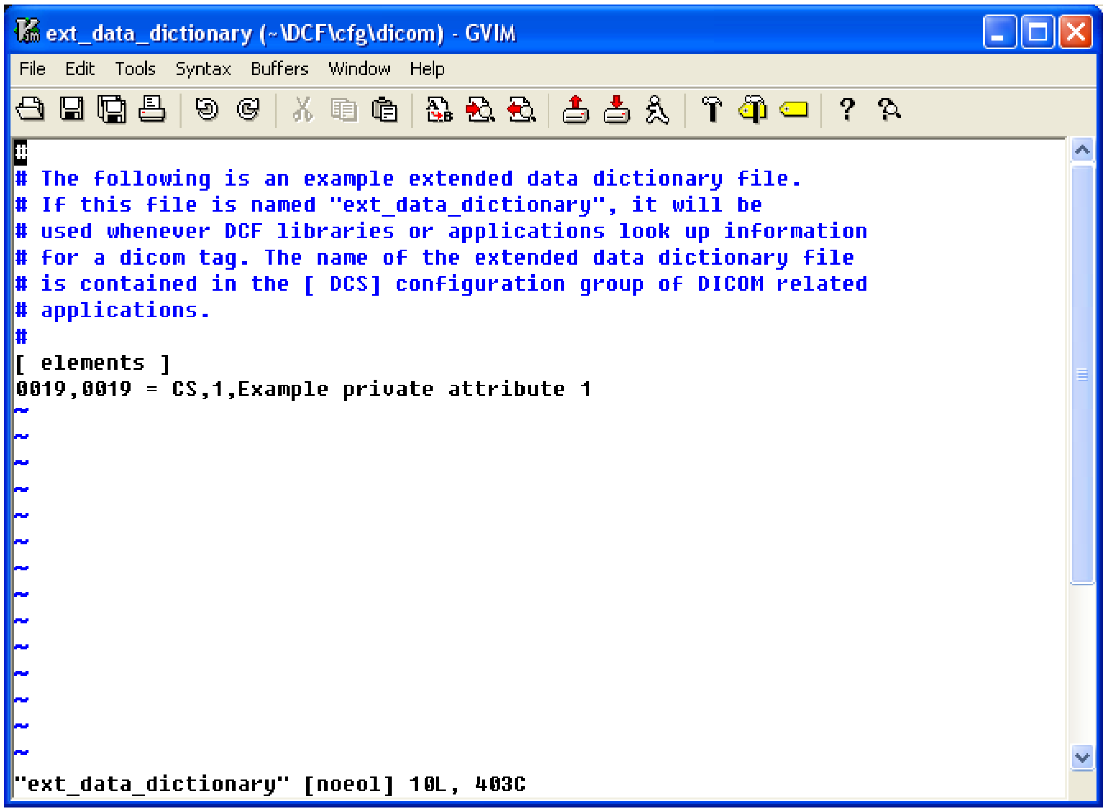
☐ Require server verification (https:) for all sites in this zone

Figure 28: Adding your computer to the list of trusted sites

Be sure to click "OK" as you close each menu to accept the changes in security preferences.

Appendix I: Editing the Extended Data Dictionary

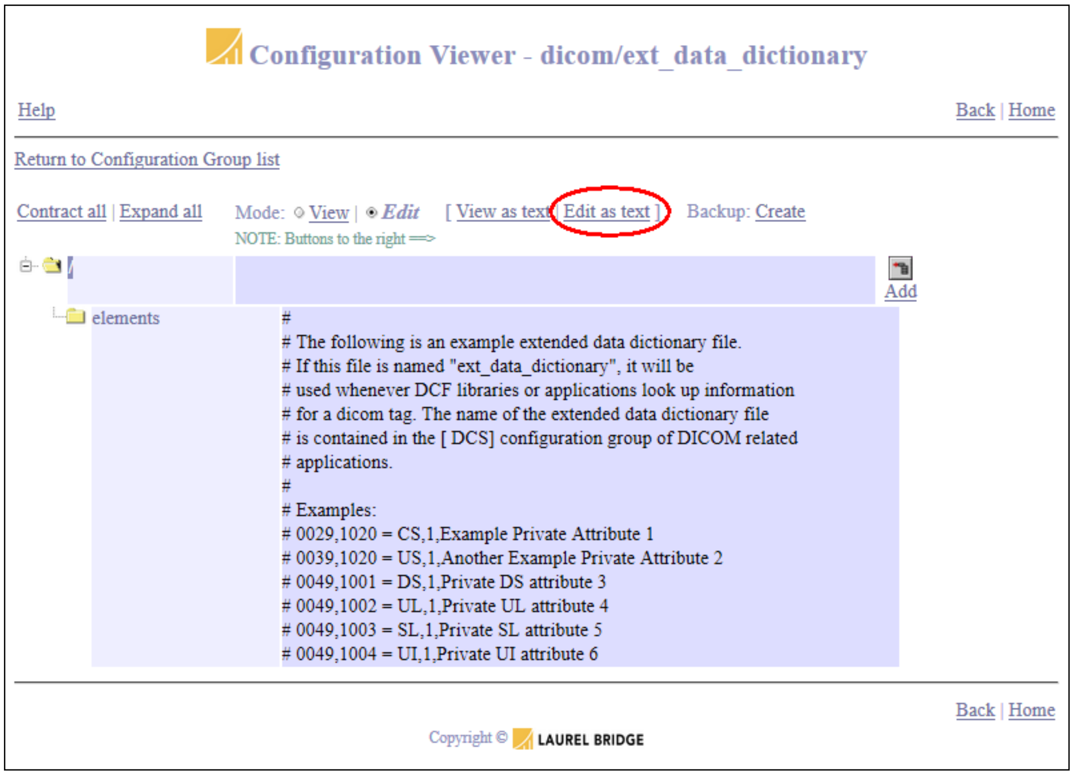
The DCF allows you to add tags to the data dictionary that it uses – this allows you to specify private tags or new tags that older versions of the DCF may not have. The DCF provides a web page to make it easy to add elements to the data dictionary – this usage is described below. You can also edit the data dictionary via a simple text editor, such as VI, Emacs, or Notepad; this is shown in the image below. (The default file for the extended data dictionary is `$DCF_ROOT/cfg/dicom/ext_data_dictionary`.)


ext_data_dictionary (~\DCF\cfg\dicom) - GVIM
File Edit Tools Syntax Buffers Window Help

```
#
# The following is an example extended data dictionary file.
# If this file is named "ext_data_dictionary", it will be
# used whenever DCF libraries or applications look up information
# for a dicom tag. The name of the extended data dictionary file
# is contained in the [ DCS] configuration group of DICOM related
# applications.
#
[ elements ]
0019,0019 = CS,1,Example private attribute 1
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"ext_data_dictionary" [noeol] 10L, 403C
```

Figure 29: Editing the Extended Data Dictionary in GVIM

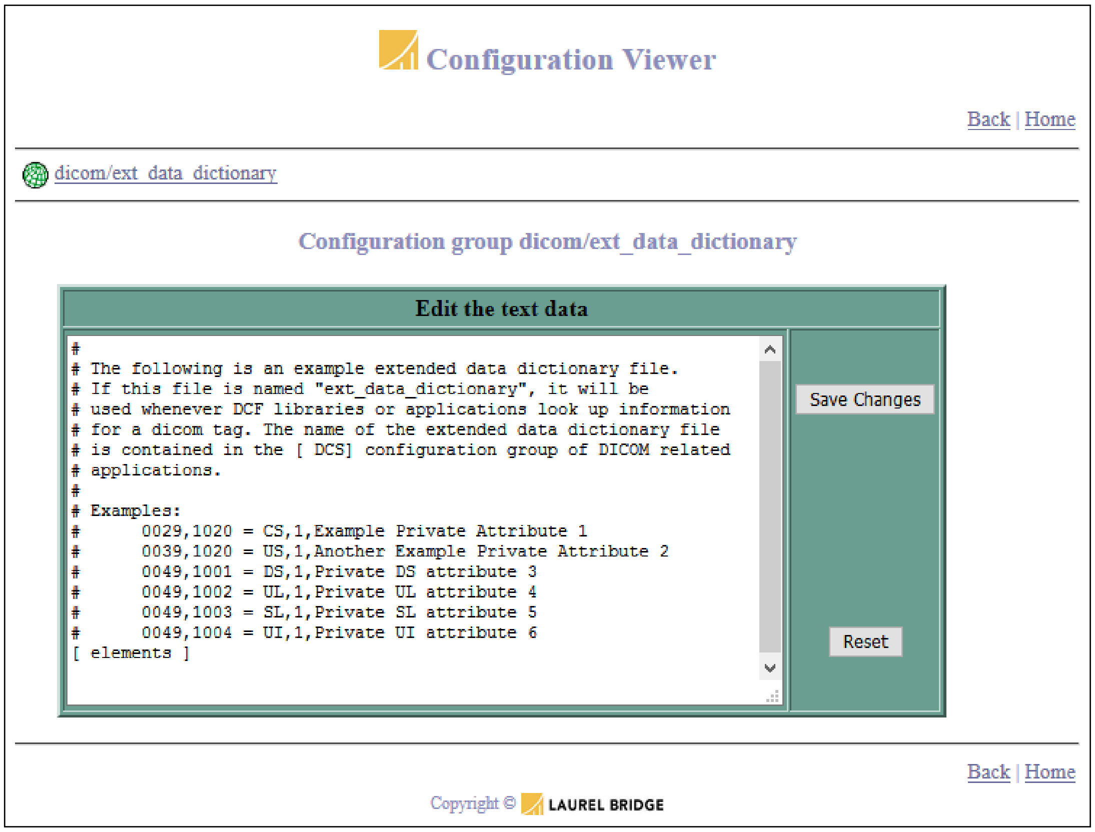
- To edit the extended data dictionary via the web: In the lower left corner of the DCF's web page is "Edit Extended Data Dictionary". Clicking that will bring you to the web pages for editing the extended data dictionary. (Only the default one – `$DCF_ROOT/cfg/dicom/ext_data_dictionary` – can be edited via this page.)


Configuration Viewer - dicom/ext_data_dictionary
Help
Back | Home
Return to Configuration Group list
Contract all | Expand all
Mode: ○ View | ● Edit
[View as text | Edit as text]
Backup: Create
NOTE: Buttons to the right ==>
Add
elements
Back | Home
Copyright © LAUREL BRIDGE

```
#
# The following is an example extended data dictionary file.
# If this file is named "ext_data_dictionary", it will be
# used whenever DCF libraries or applications look up information
# for a dicom tag. The name of the extended data dictionary file
# is contained in the [ DCS] configuration group of DICOM related
# applications.
#
# Examples:
# 0029,1020 = CS,1,Example Private Attribute 1
# 0039,1020 = US,1,Another Example Private Attribute 2
# 0049,1001 = DS,1,Private DS attribute 3
# 0049,1002 = UL,1,Private UL attribute 4
# 0049,1003 = SL,1,Private SL attribute 5
# 0049,1004 = UI,1,Private UI attribute 6
```

Figure 30: Editing the Extended Data Dictionary

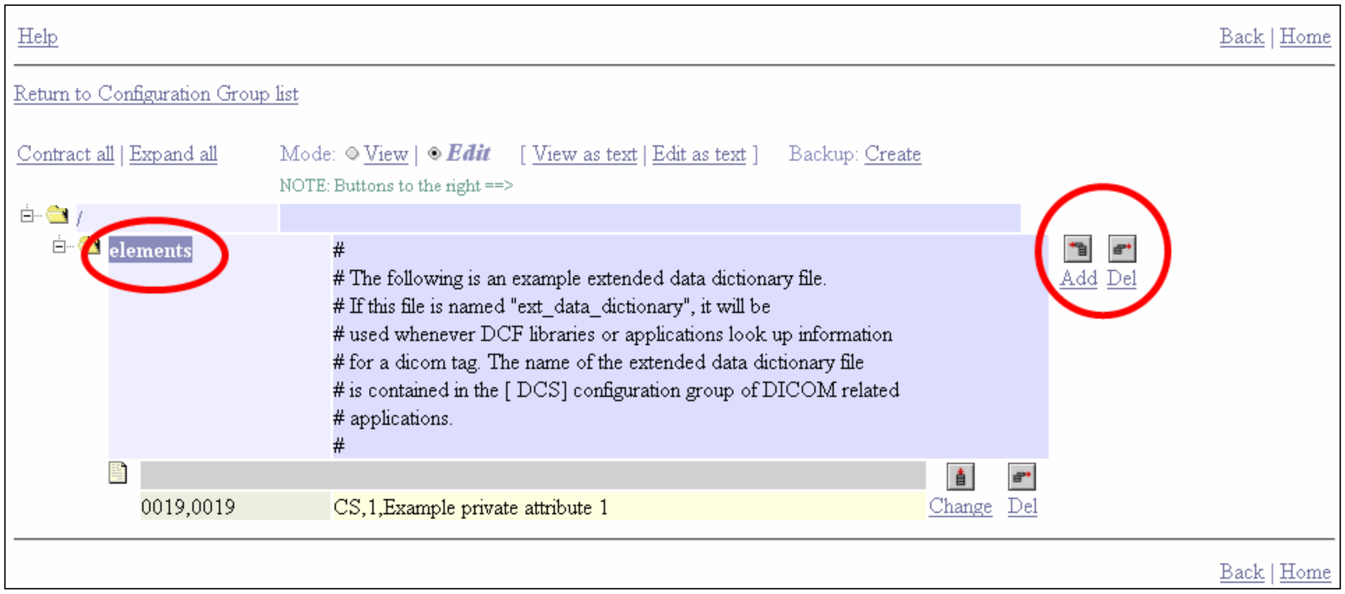
A user can edit it as text by clicking the "Edit as text" link near the top of the page.


Configuration Viewer
Back | Home
dicom/ext_data_dictionary
Configuration group dicom/ext_data_dictionary
Edit the text data
Save Changes
Reset
Back | Home
Copyright © LAUREL BRIDGE

```
#
# The following is an example extended data dictionary file.
# If this file is named "ext_data_dictionary", it will be
# used whenever DCF libraries or applications look up information
# for a dicom tag. The name of the extended data dictionary file
# is contained in the [ DCS] configuration group of DICOM related
# applications.
#
# Examples:
#       0029,1020 = CS,1,Example Private Attribute 1
#       0039,1020 = US,1,Another Example Private Attribute 2
#       0049,1001 = DS,1,Private DS attribute 3
#       0049,1002 = UL,1,Private UL attribute 4
#       0049,1003 = SL,1,Private SL attribute 5
#       0049,1004 = UI,1,Private UI attribute 6
[ elements ]
```

Figure 31: Editing the Extended Data Dictionary as text

Or he can add attributes to it via the GUI – expand the top group by clicking on plus sign next to the slash at the top level, then choose the "elements" group by clicking its name.



Help Back | Home

Return to Configuration Group list

Contract all | Expand all Mode: View | [✓ Edit] [View as text | Edit as text] Backup: Create

NOTE: Buttons to the right ==>

/

elements

```
#
# The following is an example extended data dictionary file.
# If this file is named "ext_data_dictionary", it will be
# used whenever DCF libraries or applications look up information
# for a dicom tag. The name of the extended data dictionary file
# is contained in the [ DCS] configuration group of DICOM related
# applications.
#
```

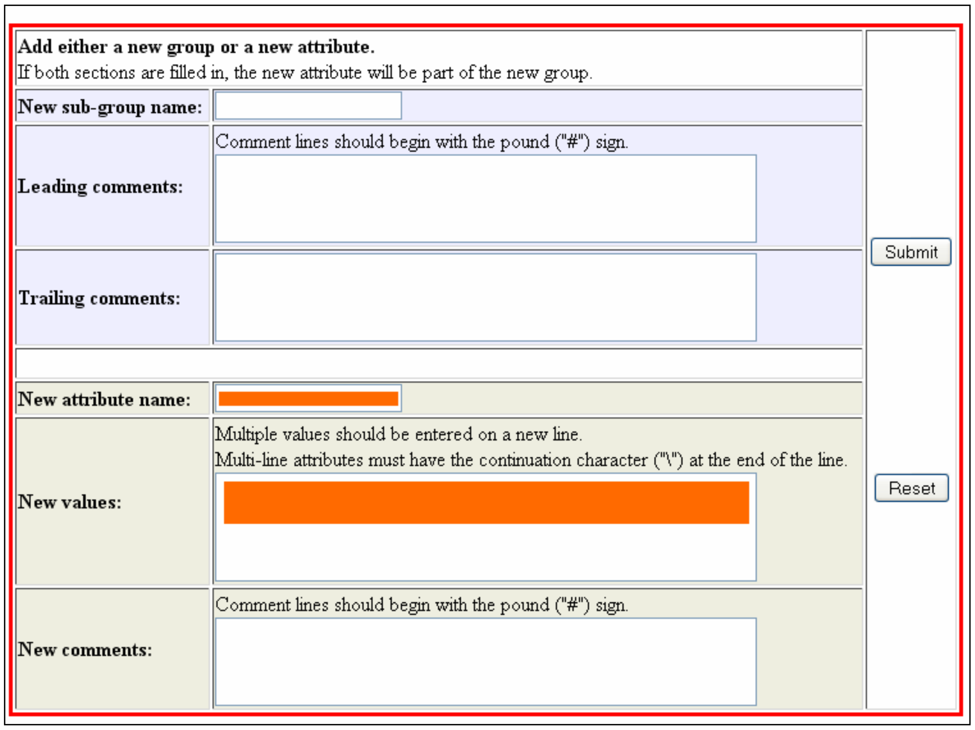
Add Del

0019,0019 CS,1,Example private attribute 1 Change Del

Back | Home

Figure 32: Select the "elements" group to add to the data dictionary, and the click "Add".

Then click the "Add" button (at the far right) that is at the same level as the Elements group. This will open up a new page where you can add new groups or attributes. For the extended data dictionary, you want to enter values in the "attribute" boxes – these are highlighted in orange below.



Add either a new group or a new attribute.
If both sections are filled in, the new attribute will be part of the new group.

New sub-group name:	
Leading comments:	Comment lines should begin with the pound ("#") sign.
Trailing comments:	
New attribute name:	🟧
New values:	Multiple values should be entered on a new line. Multi-line attributes must have the continuation character ("\") at the end of the line. 🟧
New comments:	Comment lines should begin with the pound ("#") sign.

Figure 33: Enter the new values for the data dictionary

Enter the appropriate values in the boxes, and click the “Submit” button.

The values to enter: the tags are entered as new attribute names, and then the values for the attributes are the VR, VM, and a description, separated by commas.

For example: To add tag 0029,1020 to the data dictionary, enter “0029,1020” in the **new attribute name** field. In the **new values** field, enter “CS,1,Example private attribute 1”.

Repeat as necessary for additional tags.

When you are done adding tags to the extended data dictionary, click the “Home” link at the top right corner to return to the DCF’s home page. You will have to restart any applications that will use the new values in the extended data dictionary.

Appendix J: Product Licensing and Activation

This Appendix contains:

Section 1: Installing a new SDK License (for developers)
Tells you how to install a new toolkit license for the DCF toolkit (SDK) that you are using for your software development;

Section 2 Activating an SDK License (for developers)
Explains how to activate a toolkit license for the DCF toolkit (SDK);

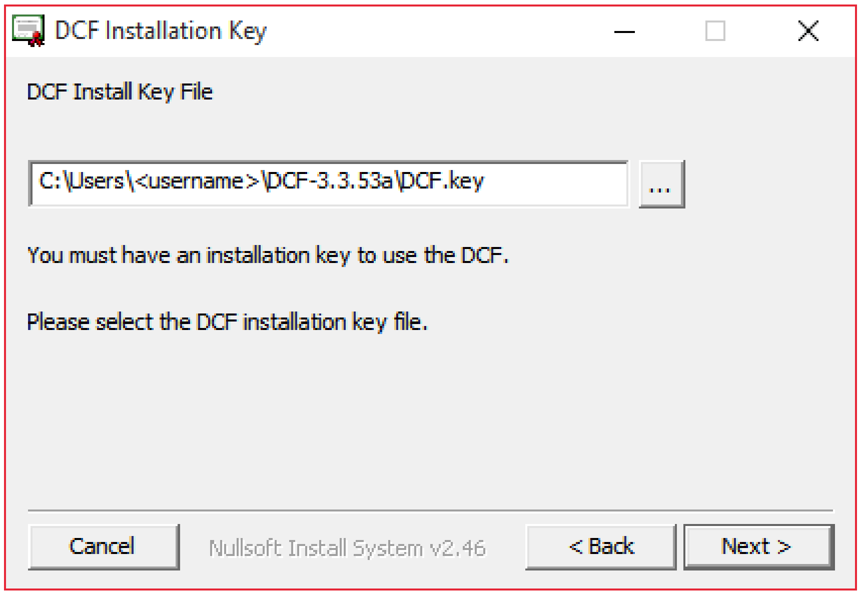
Section 3: Deploying OEM Applications with an Activatable License
Explains how to deploy an activatable license with the application that you have developed.

The licensing topic is covered for MAC-based licenses in section 13.2, License Key for a Deployed Application.

1. Installing a new SDK License

A license is normally installed at the time of initial installation of your DCF Toolkit (SDK).

If your existing DICOM Connectivity Framework SDK developer license has expired, you may request a new one from Laurel Bridge Software. A utility, available from the Windows Start Menu, is used to install your new SDK license on your system.



C:\Users\<username>\DCF-3.3.53a\DCF.key

You must have an installation key to use the DCF.

Please select the DCF installation key file.

Figure 34: Installing a new license

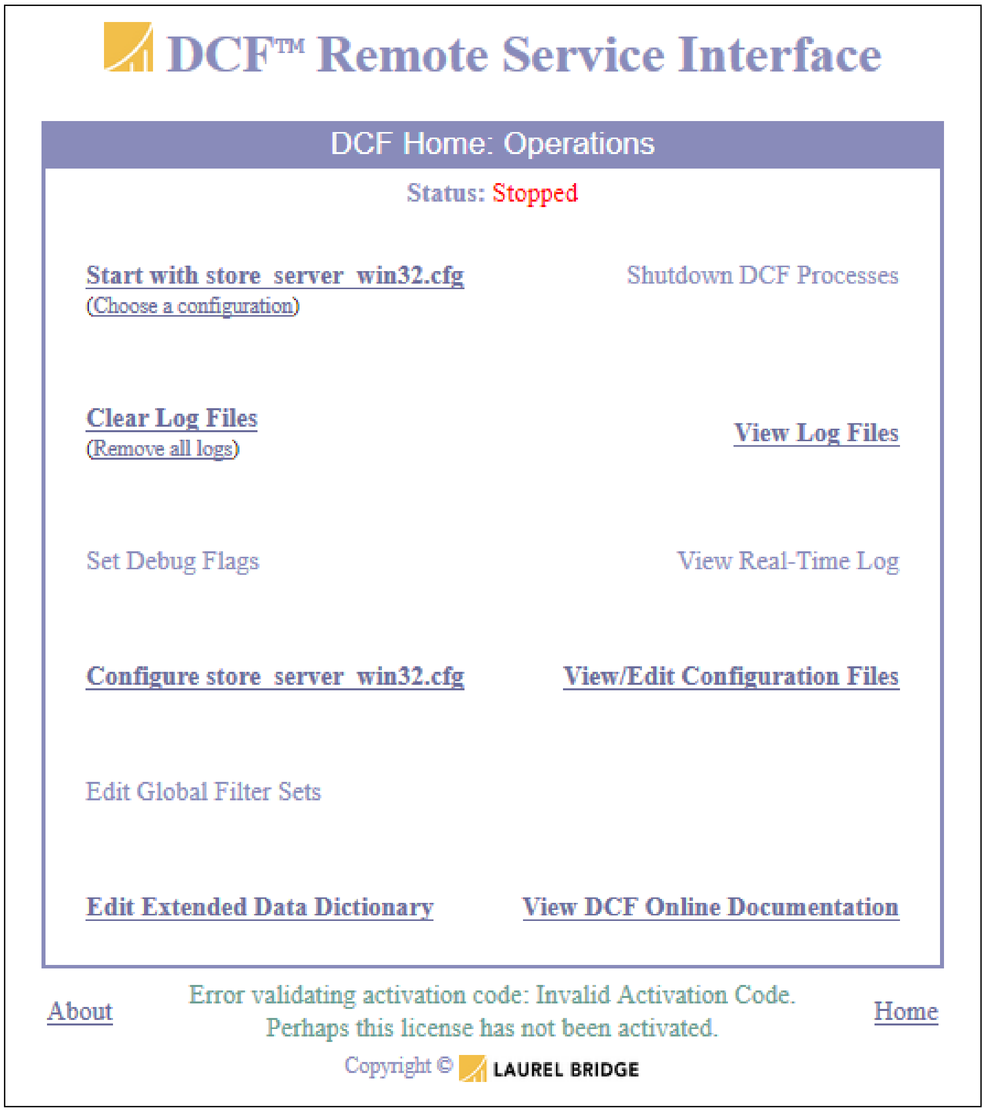
Note, for Linux: Install a new license by copying the key file into the `cfg` directory as follows:

```
cp <keyfile> $DCF_ROOT/cfg/systeminfo
```

2. Activating an SDK License

There are two styles of licenses available for the DCF Toolkit – one type works right away; another type requires activation before the DCF Toolkit applications will work.

If you need to activate your SDK license, you may see a warning message when you first install the DCF, or you may see a message like the one in green at the bottom of the DCF's Operations window.



DCF™ Remote Service Interface

DCF Home: Operations

Status: Stopped

Start with store_server_win32.cfg
(Choose a configuration)

Shutdown DCF Processes

Clear Log Files
(Remove all logs)

View Log Files

Set Debug Flags

View Real-Time Log

Configure store_server_win32.cfg

View/Edit Configuration Files

Edit Global Filter Sets

Edit Extended Data Dictionary

View DCF Online Documentation

About

Error validating activation code: Invalid Activation Code.
Perhaps this license has not been activated.

Home

Copyright © LAUREL BRIDGE

Figure 35: Warning to activate the license

To activate your SDK license, launch the License Activation Utility from the Start menu:

Start → All Programs → DICOM Connectivity Framework → Activate license

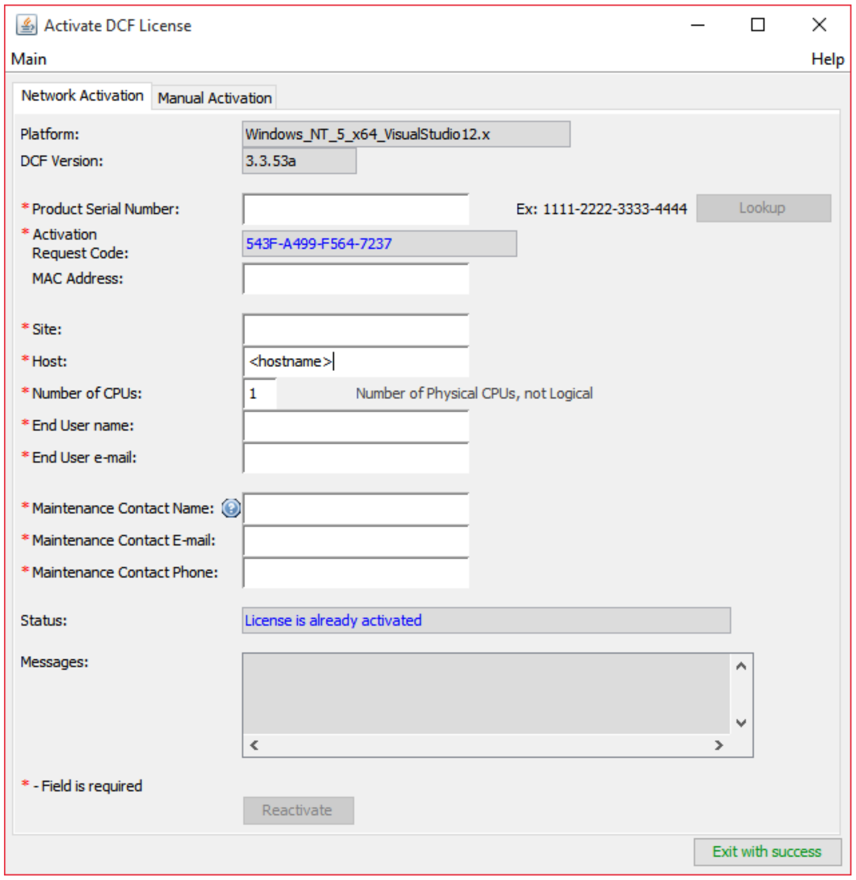
The License Activation Utility will let you activate your license in either Network mode or in Manual mode; each is described below. Note, as mentioned above, some keys do not require activation – in this case, the utility will warn you of this fact and you can only exit the tool.

2.1. SDK Network Activation Mode

If you have Internet connectivity, it is preferred to activate the DCF Toolkit license via the Network – you will see a screen like that shown below.

Launch the License Activation Utility from the Windows Start menu:

Start → All Programs → DICOM Connectivity Framework → Activate license

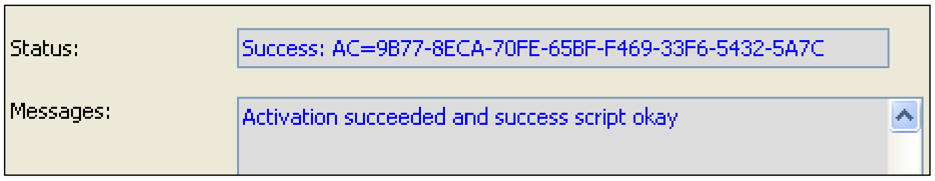


Platform: Windows_NT_5_x64_VisualStudio12.x
DCF Version: 3.3.53a
* Product Serial Number: Ex: 1111-2222-3333-4444
* Activation Request Code: 543F-A499-F564-7237
MAC Address:
* Site:
* Host: <hostname>
* Number of CPUs: 1 Number of Physical CPUs, not Logical
* End User name:
* End User e-mail:
* Maintenance Contact Name:
* Maintenance Contact E-mail:
* Maintenance Contact Phone:
Status: License is already activated
Messages:
* - Field is required

Figure 36: Network mode for activating a license

Fill in each of the fields specified. The Product Serial Number was given to you when you purchased the DCF, or it can be found on the LBS licensing web site as you view your available SDK license keys. Note that the fields in blue do not need to be entered by you – the Activation Request Code (above) is a system identifier that is generated on your computer by the application.

Once all the fields are filled in correctly, press the Activate button. The utility will communicate with the Laurel Bridge licensing web site and receive an Activation Code and other information back from the web site. Upon success, the status fields will look something like this:



Status: Success: AC=9B77-8ECA-70FE-65BF-F469-33F6-5432-5A7C
Messages: Activation succeeded and success script okay

Figure 37: Activation succeeded via Internet

The DCF license should now be activated, allowing the DCF to be fully functional on this system. If activation failed, you will see error messages explaining why. Resolve the errors if possible and try activating again.

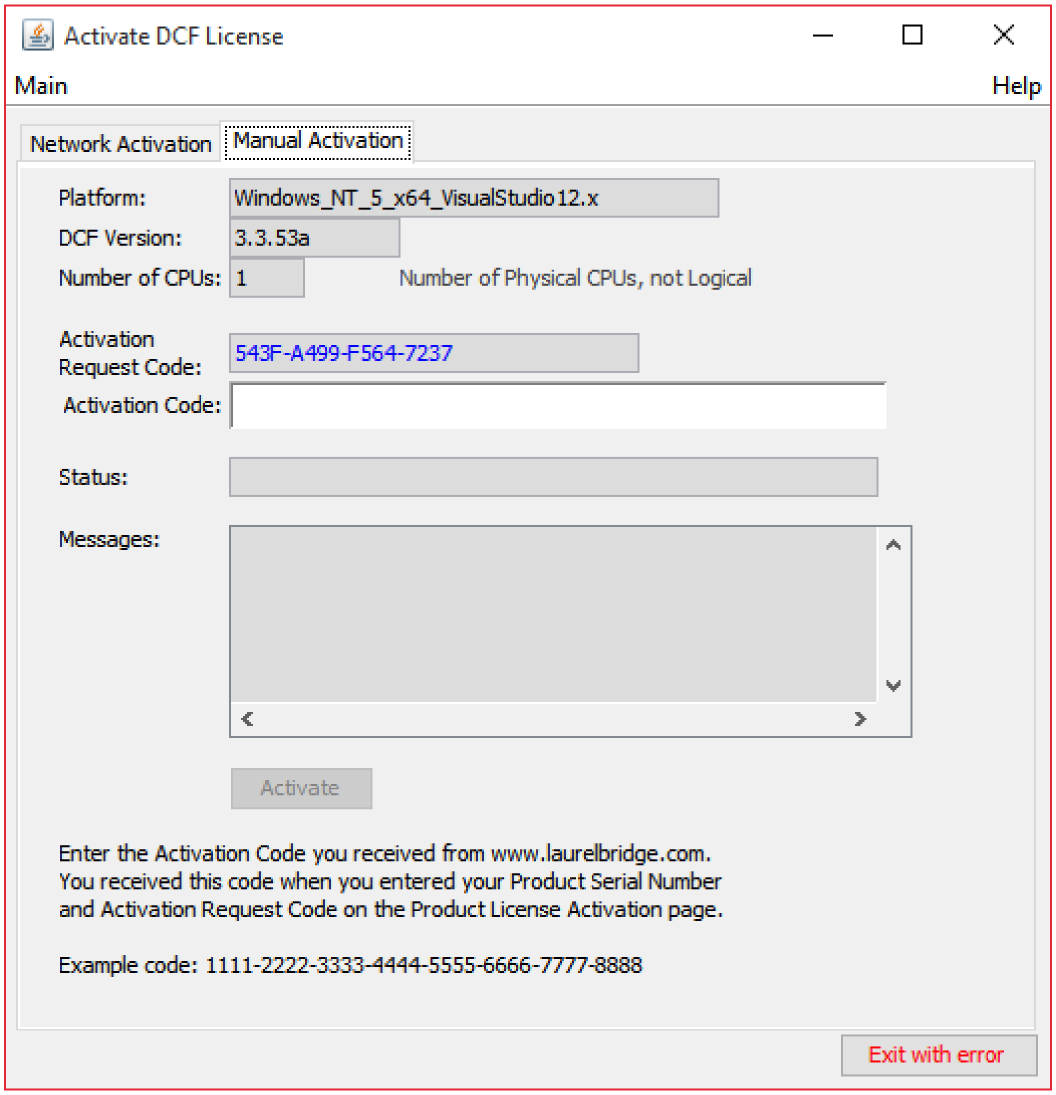
2.2. SDK Manual Activation Mode

Manual activation is used when the target computer with the DCF Toolkit does not have Internet access to the Laurel Bridge licensing web site – note that Network Activation is the *preferred* mode.

Launch the License Activation Utility from the Windows Start menu:

Start → All Programs → DICOM Connectivity Framework → Activate license

After you launch the utility, you should select the Manual tab if it is not already selected.



Platform:	Windows_NT_5_x64_VisualStudio12.x
DCF Version:	3.3.53a
Number of CPUs:	1 Number of Physical CPUs, not Logical
Activation Request Code:	543F-A499-F564-7237
Activation Code:	
Status:	
Messages:	

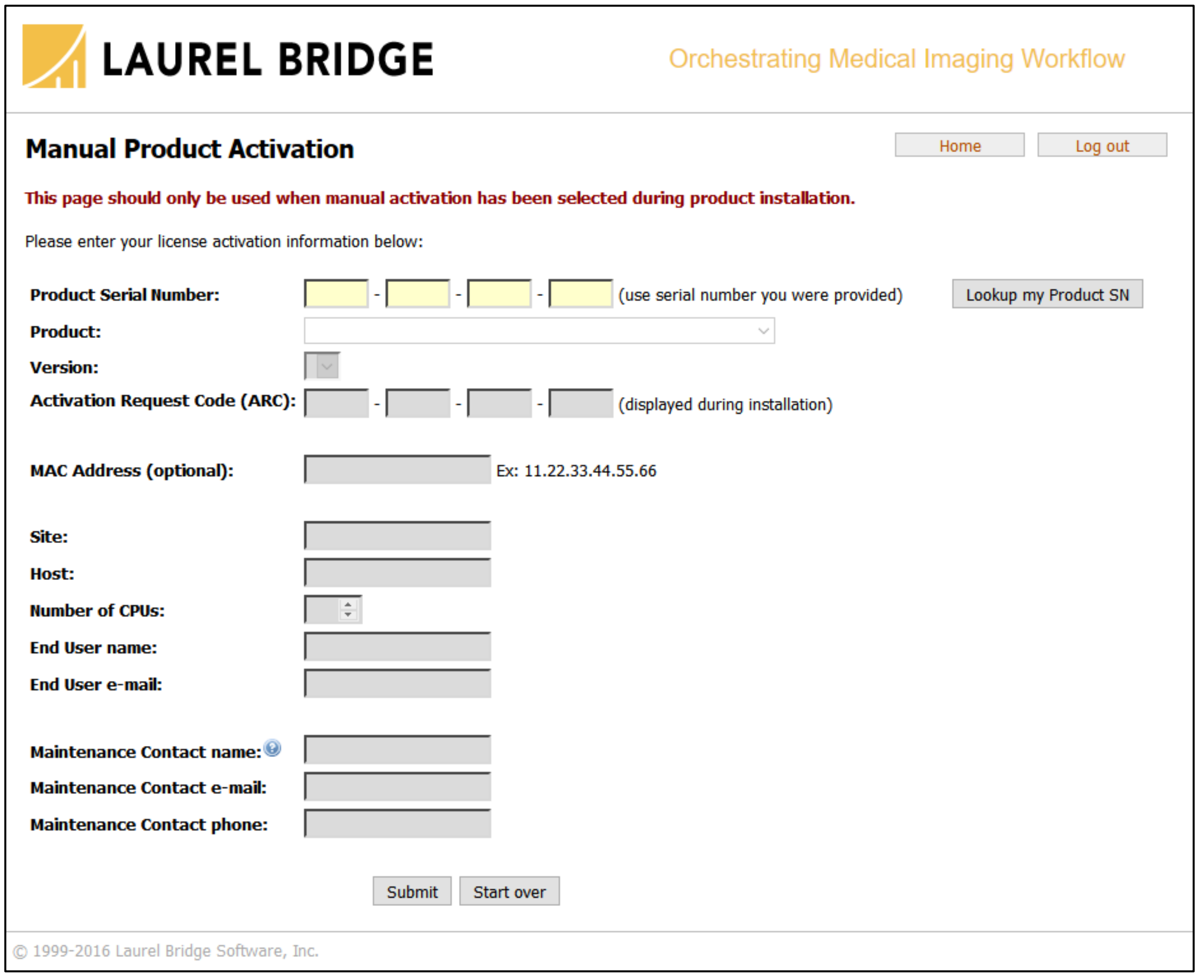
Enter the Activation Code you received from www.laurelbridge.com.
You received this code when you entered your Product Serial Number
and Activation Request Code on the Product License Activation page.

Example code: 1111-2222-3333-4444-5555-6666-7777-8888

Figure 38: Manual mode for activating a license

To get the required information to complete the form, you will nee to use a web browser on a different system that does have Internet access. Do the following:

- Go to https://www.laurelbridge.com/product_activation.php. Alternatively, you can proceed to the Laurel Bridge Software customer web site at www.laurelbridge.com, select "Support", and then choose "Product License Activation".
- See the screenshot below. Enter the Product Serial Number that was obtained and the Activation Request Code displayed on the target system by the utility (in the example shown above, it is "`A638-DBCD-B08C-F237`"). Also enter the site and contact information, and the number of CPUs for the system that is being activated.
- Press the Submit button when complete, see the screenshot below.


LAUREL BRIDGE
Orchestrating Medical Imaging Workflow
Home
Log out
Lookup my Product SN
Submit
Start over
© 1999-2016 Laurel Bridge Software, Inc.

Manual Product Activation

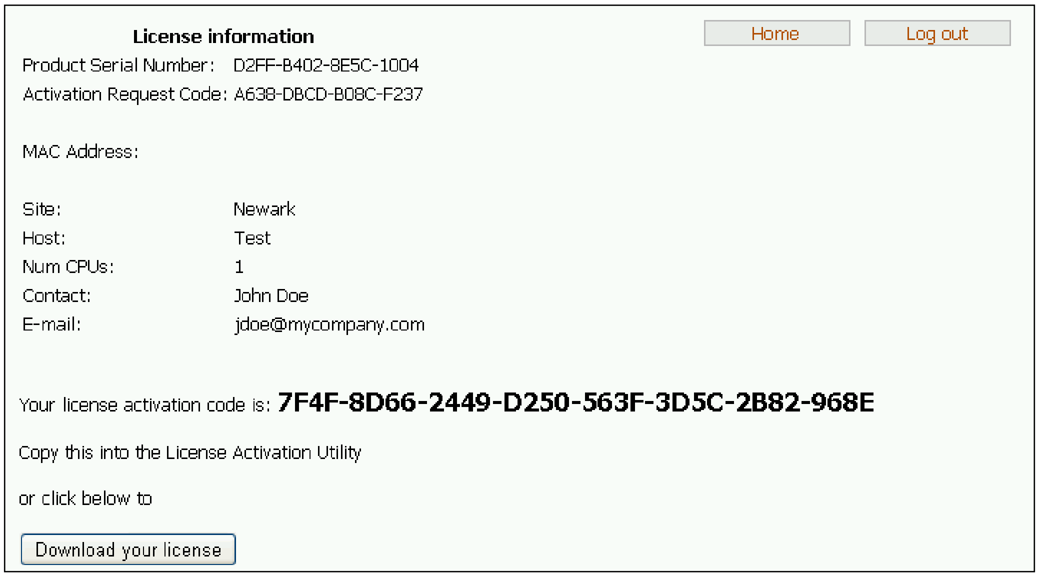
This page should only be used when manual activation has been selected during product installation.

Please enter your license activation information below:

Product Serial Number:	- - - (use serial number you were provided)
Product:	
Version:	
Activation Request Code (ARC):	- - - (displayed during installation)
MAC Address (optional):	Ex: 11.22.33.44.55.66
Site:	
Host:	
Number of CPUs:	
End User name:	
End User e-mail:	
Maintenance Contact name:	
Maintenance Contact e-mail:	
Maintenance Contact phone:	

Figure 39: License activation web page

- After you click Submit, you will see a screen like that below.



License information

Product Serial Number: D2FF-B402-8E5C-1004
Activation Request Code: A638-DBCD-B08C-F237

MAC Address:

Site: Newark
Host: Test
Num CPUs: 1
Contact: John Doe
E-mail: jdoe@mycompany.com

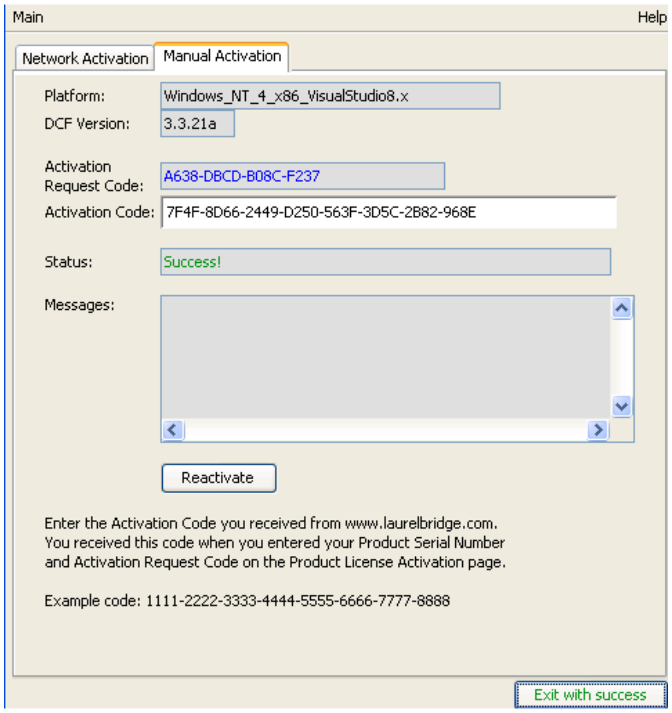
Your license activation code is: **7F4F-8D66-2449-D250-563F-3D5C-2B82-968E**

Copy this into the License Activation Utility

or click below to

Figure 40: Web page showing license activation code

- Note the license activation code and enter it onto the License Activation Utility on your target computer system.
  (Alternately, if you wish, you can also click the "download" button to download the key – if you do this, install the downloaded key using the Install License utility and don't continue with the manual activation process.)
  After clicking Activate, the utility should display something similar to this:



Platform: Windows_NT_4_x86_VisualStudio8.x
DCF Version: 3.3.21a

Activation Request Code: A638-DBCD-B08C-F237
Activation Code: 7F4F-8D66-2449-D250-563F-3D5C-2B82-968E

Status: Success!

Messages:

Enter the Activation Code you received from www.laurelbridge.com.
You received this code when you entered your Product Serial Number
and Activation Request Code on the Product License Activation page.

Example code: 1111-2222-3333-4444-5555-6666-7777-8888

Figure 41: Successfully activated the license manually

The DCF license should now be activated, allowing the DCF libraries to be fully functional.

3. Deploying OEM Applications with an Activatable License

3.1. Developer Perspective

An OEM user developing software applications with the DCF may want to let their clients (customers) use activatable DCF license keys. Each OEM user must contact Laurel Bridge to ask for a template, and Laurel Bridge will supply a template license file for the OEM to distribute with their software. End Users can then use the activation procedures described below in "User Perspective" to fully activate the DCF license for their application.

If a template license file is used, then it must be installed in the `%DCF_ROOT%/cfg` directory and must be named `systeminfo`.

The developer typically will want to have the license activation occur as part of the OEM application installation process. A developer would write an installer that would invoke the License Activation GUI provided with the DCF. For example, after installing his own software and the required DCF libraries and binaries, the OEM installer would then invoke the appropriate Activation GUI tool:

java com.lbs.ActivateDcfLicense.ActivateDcfLicense

This will launch the Java GUI, letting the user input the required information and activate the license either over the Internet ("network" mode) or proceed to activate it manually (as described below under "manual activation"). **Note** that there is also a C# version of the tool – *nActivateDcfLicense* – if you are shipping a C# application; this document will refer to the Java version for consistency, but the same options exist with the C# version.

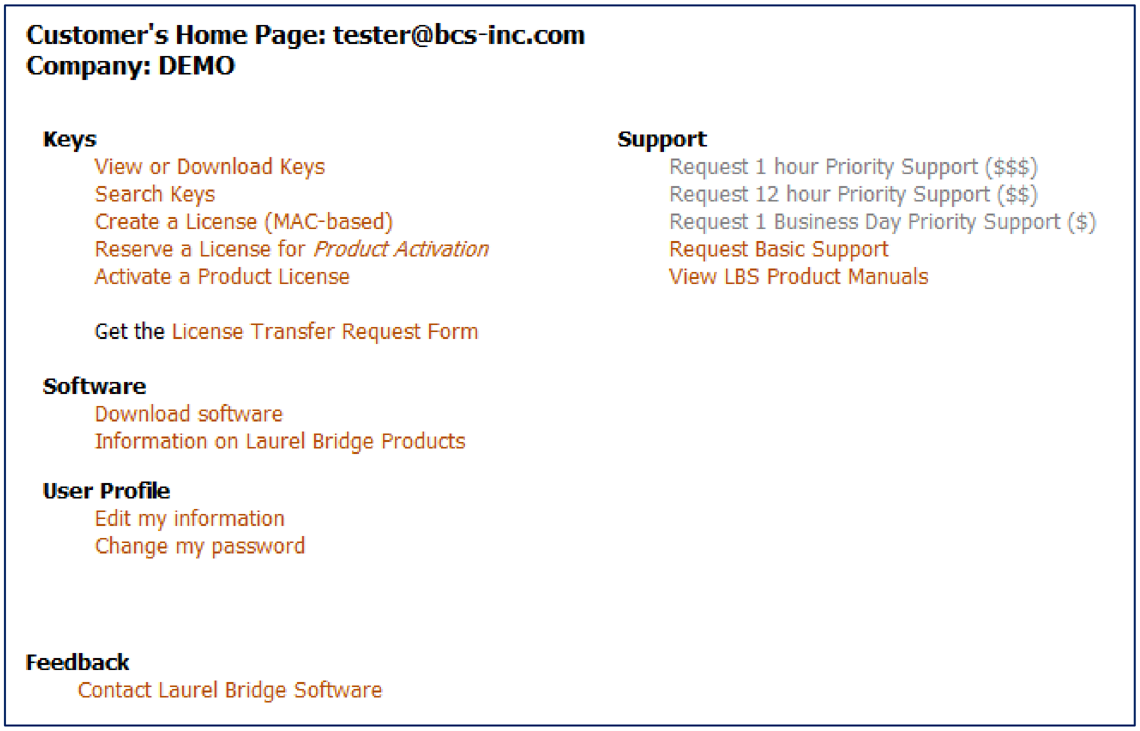
The License Activation GUI will return 0 (zero) if the activation succeeds; any other status means that the license activation failed. This lets the developer configure his installer so that it can check if the license was successfully activated.

The developer can also write their own method for activating the license by looking at the source code for the GUI – the source code can be found in `%DCF_ROOT%/devel/jsrc/com/lbs/ActivateDcfLicense`, specifically in the `ActivateDcfLicense.java` file – and adapting it accordingly.

The License Activation tool can also be invoked in command line mode if the developer doesn't want to write his own activation utility but does have his own GUI look-and-feel. To see the parameters that are required, from a DCF Command Prompt, run the utility with the *–h* switch as follows:

java com.lbs.ActivateDcfLicense.ActivateDcfLicense -h

To distribute a license to a client, an authorized person (OEM or Laurel Bridge) will go onto the Laurel Bridge web site (as shown below) and "reserve" a key (as described in section 3.2.2 below) by filling out the form's fields; doing that will generate a product serial number that the OEM will give to their customer or client. The recipient can then activate the key using the network or manual methods described below.



Customer's Home Page: tester@bcs-inc.com
Company: DEMO

Keys
- View or Download Keys
- Search Keys
- Create a License (MAC-based)
- Reserve a License for *Product Activation*
- Activate a Product License

Get the License Transfer Request Form

Software
- Download software
- Information on Laurel Bridge Products

User Profile
- Edit my information
- Change my password

Feedback
- Contact Laurel Bridge Software

Support
- Request 1 hour Priority Support ($$$)
- Request 12 hour Priority Support ($$)
- Request 1 Business Day Priority Support ($)
- Request Basic Support
- View LBS Product Manuals

Figure 42: Licensing Web Site Options

3.1.1. Files to Distribute

To use the License Activation Utility, a developer will need to distribute both the utility itself and also the helper utility `dcf_cpu_count`.

Note that the utility `dcf_cpu_count` should be available in the environment so that it can be used by the License Activation Utility – for example, if `dcf_cpu_count` is extracted into the `%DCF_ROOT%/bin` directory, then that directory should be put into the PATH environment variable.

The License Activation Utility itself is in the `LaurelBridge.jar` file – this should be included in your distribution, and its location should be put into the `CLASSPATH` environment variable.

If you are using the C# version of the Utility, instead of the Java jar file, then you will need the executable `nActivateDcfLicense`, typically located in the DCF's bin directory. The location of the executable should be put in the PATH environment variable.

To summarize, the files to distribute are:

- `LaurelBridge.jar` (Java) *or* `nActivateDcfLicense` (C#)
- `dcf_cpu_count` executable

3.2. User Perspective

OEM users of the DCF Toolkit may develop an application that uses the activation model for DCF licensing. In this mode, the OEM purchases some number of runtime licenses (typically one per physical CPU that hosts DCF code). Rather than creating licenses for individual machines by providing a network adapter MAC address and other information, the process in the activation model is divided into the following three parts.

- **Install the un-activated template license** (typically distributed along with the application)
- **Create and Reserve a license** (typically done by LBS via the license web site)
- **Activate a license** (typically done by the end user at time of the application installation)

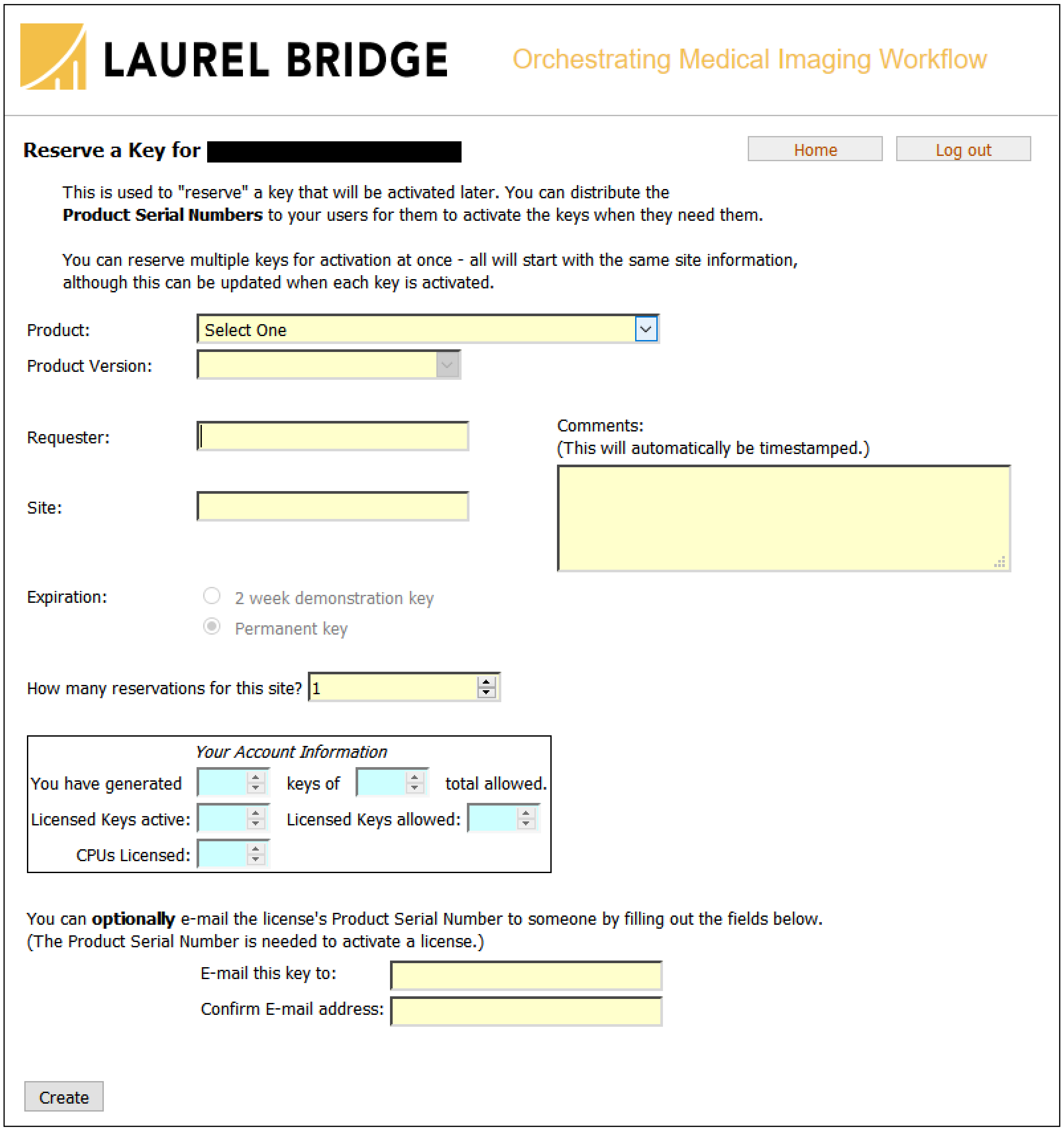
3.2.1. Install the un-activated template license

The template license is a license file that is distributed and installed by the OEM along with DCF library DLLs and other files as part of each customer system or application installation. The DCF software being distributed or installed will not be fully functional until this template license is activated.

Should an OEM choose this option, the template license will be generated by Laurel Bridge and provided to the OEM developer upon request. The OEM can distribute this template license with his software and must provide one or more Product Serial Numbers to his customers/clients so that they can activate the license(s) during product installation. Typically the template is installed by the OEM application installer and is unknown to the end-user; the Product Serial Number is provided to the end-user who enters that number when requested during the application installation process.

3.2.2. Create and Reserve a license

Using the Laurel Bridge Software “Customer Access” web site, an authorized user (typically LBS) provides contact information and information about the deployment location within the end-user customer organization for the license. A Product Serial Number is generated and it is ultimately provided to the end-user customer for each reserved license. The OEM distributes those serial numbers to the installer/administrator/end-user for the system to be activated. After the “Create” button is clicked on the Reserve a Key screen, the Product Serial Number for the key will be displayed at the bottom of the form (below the “Create” button) shown in the following screen shot:



Reserve a Key for [redacted]

This is used to "reserve" a key that will be activated later. You can distribute the **Product Serial Numbers** to your users for them to activate the keys when they need them.

You can reserve multiple keys for activation at once - all will start with the same site information, although this can be updated when each key is activated.

Product:	Select One
Product Version:	
Requester:	
Site:	

Comments:
(This will automatically be timestamped.)

Expiration: ( ) 2 week demonstration key [✓ Permanent key]

How many reservations for this site? 1

Your Account Information

You have generated ___ keys of ___ total allowed.
Licensed Keys active: ___ Licensed Keys allowed: ___
CPUs Licensed: ___

You can **optionally** e-mail the license's Product Serial Number to someone by filling out the fields below.
(The Product Serial Number is needed to activate a license.)

E-mail this key to:
Confirm E-mail address:

Figure 43: Reserving a license key

3.2.3. Activate a license

During or after the process of installing the customer application, the DCF license can be activated, typically this is done by the installer/administrator/end-user for the system to be activated. Two modes for activation are supported. For systems connected to the Internet and with connectivity to the Laurel Bridge web server, “Network Activation” is the preferred approach. For systems without such connectivity, “Manual Activation” can be performed.

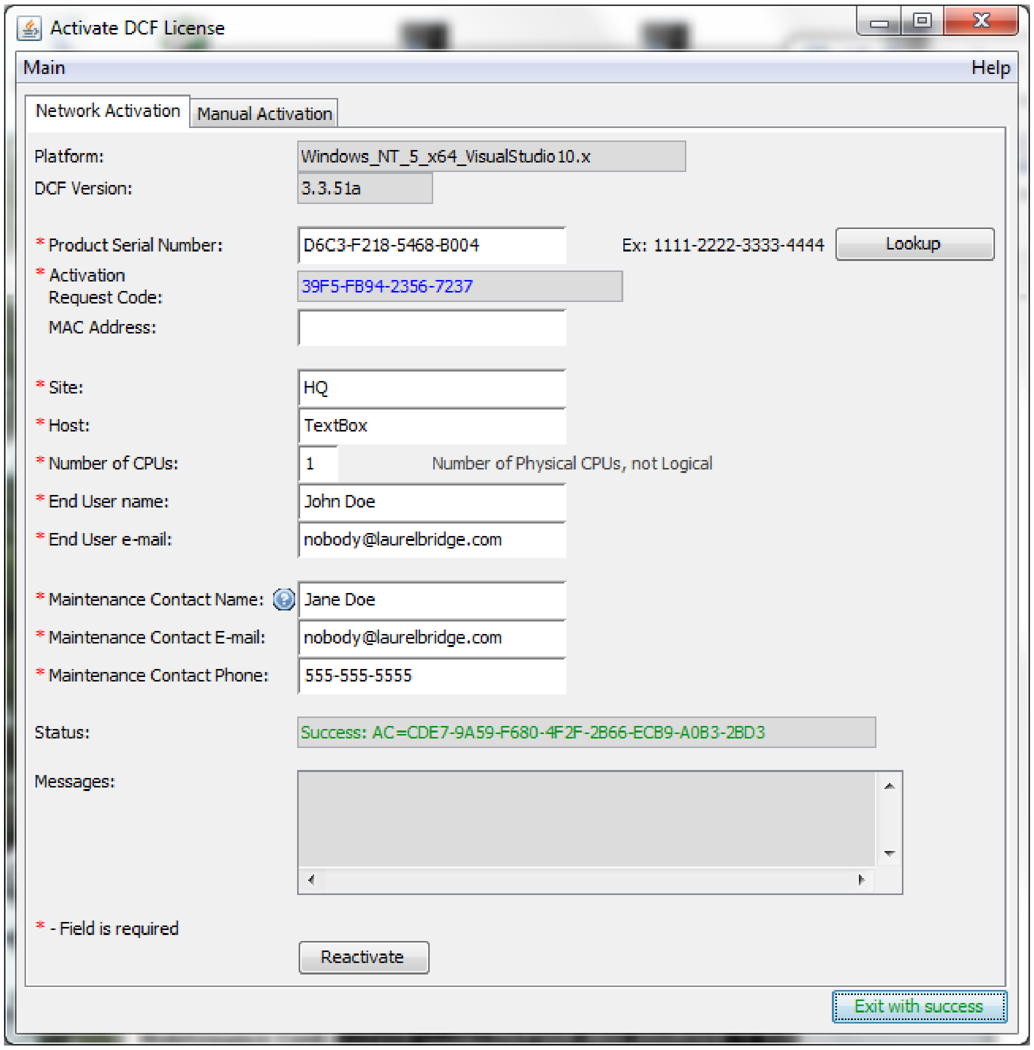
3.2.3.1. Network Activation Mode

For systems connected to the Internet and with connectivity to the Laurel Bridge web server, "Network Activation" is the preferred approach.

Run the license activation tool—the *ActivateDcfLicense* utility; this is typically done by the OEM's installer program. For Java, this tool is a Java application that is invoked with the following command:

java com.lbs.ActivateDcfLicense.ActivateDcfLicense

Once the activation tool graphical user interface comes up, select the "Network Activation" tab, if that is not currently selected. After providing the necessary information, and clicking the "Activate" button, the screen should look similar to the following:



Platform:	Windows_NT_5_x64_VisualStudio10.x
DCF Version:	3.3.51a
* Product Serial Number:	D6C3-F218-5468-B004 Ex: 1111-2222-3333-4444
* Activation Request Code:	39F5-FB94-2356-7237
MAC Address:	
* Site:	HQ
* Host:	TextBox
* Number of CPUs:	1 Number of Physical CPUs, not Logical
* End User name:	John Doe
* End User e-mail:	nobody@laurelbridge.com
* Maintenance Contact Name:	Jane Doe
* Maintenance Contact E-mail:	nobody@laurelbridge.com
* Maintenance Contact Phone:	555-555-5555
Status:	Success: AC=CDE7-9A59-F680-4F2F-2B66-ECB9-A0B3-2BD3
Messages:	

* - Field is required

Figure 44: Network activation of a license

Note that the fields in blue do not need to be entered by the user. The Activation Request Code is a system identifier that is generated on the customer computer by the DCF libraries. This Activation Request Code, as well as the other user supplied fields, is sent to the Laurel Bridge Software license server. Upon success, the server returns the "Activation Code" and other licensing information – in this example the code is the string "`72EB-22AF-0732-9C95-B0A8-E279-4BEB-D9FF`".

After successful activation, the license file (`%DCF_CFG%\systeminfo`) will have been modified to include the activation code. The DCF library software should now be fully functional on the licensed system.

To be able to use the Network Activation model the end user system must have Internet access as described below in Section 3.3.3, Networking Issues Related to Network Activation.

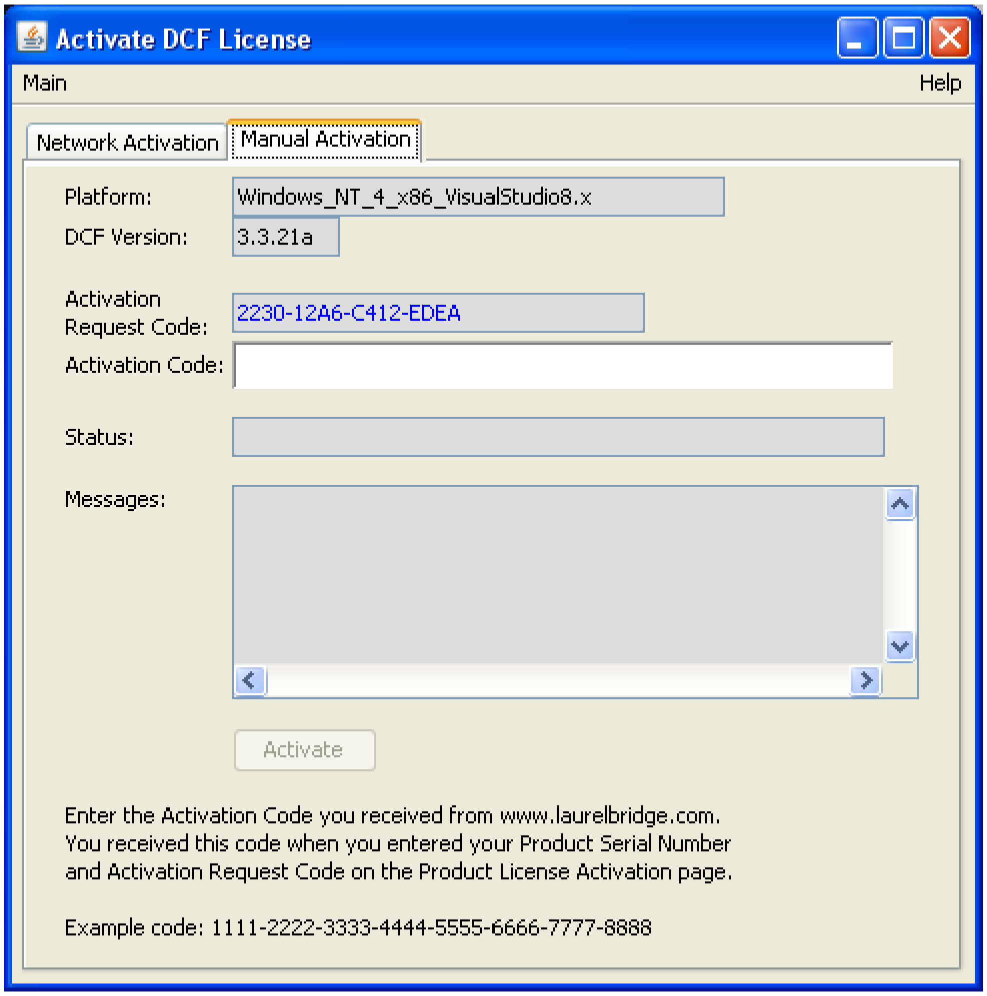
3.2.3.2. Manual Activation Mode

For systems without Internet connectivity to the Laurel Bridge web server, “Manual Activation” must be performed.

Run the license activation tool on the target system—the `ActivateDcfLicense` utility; this is typically done by the OEM’s installer program. This tool is a Java application that is invoked with the following command:

java com.lbs.ActivateDcfLicense.ActivateDcfLicense

Once the GUI comes up, select the “Manual Activation” tab; the screen should look similar to the following:



Platform: Windows_NT_4_x86_VisualStudio8.x
DCF Version: 3.3.21a
Activation Request Code: 2230-12A6-C412-EDEA
Activation Code:
Status:
Messages:

Enter the Activation Code you received from www.laurelbridge.com.
You received this code when you entered your Product Serial Number
and Activation Request Code on the Product License Activation page.

Example code: 1111-2222-3333-4444-5555-6666-7777-8888

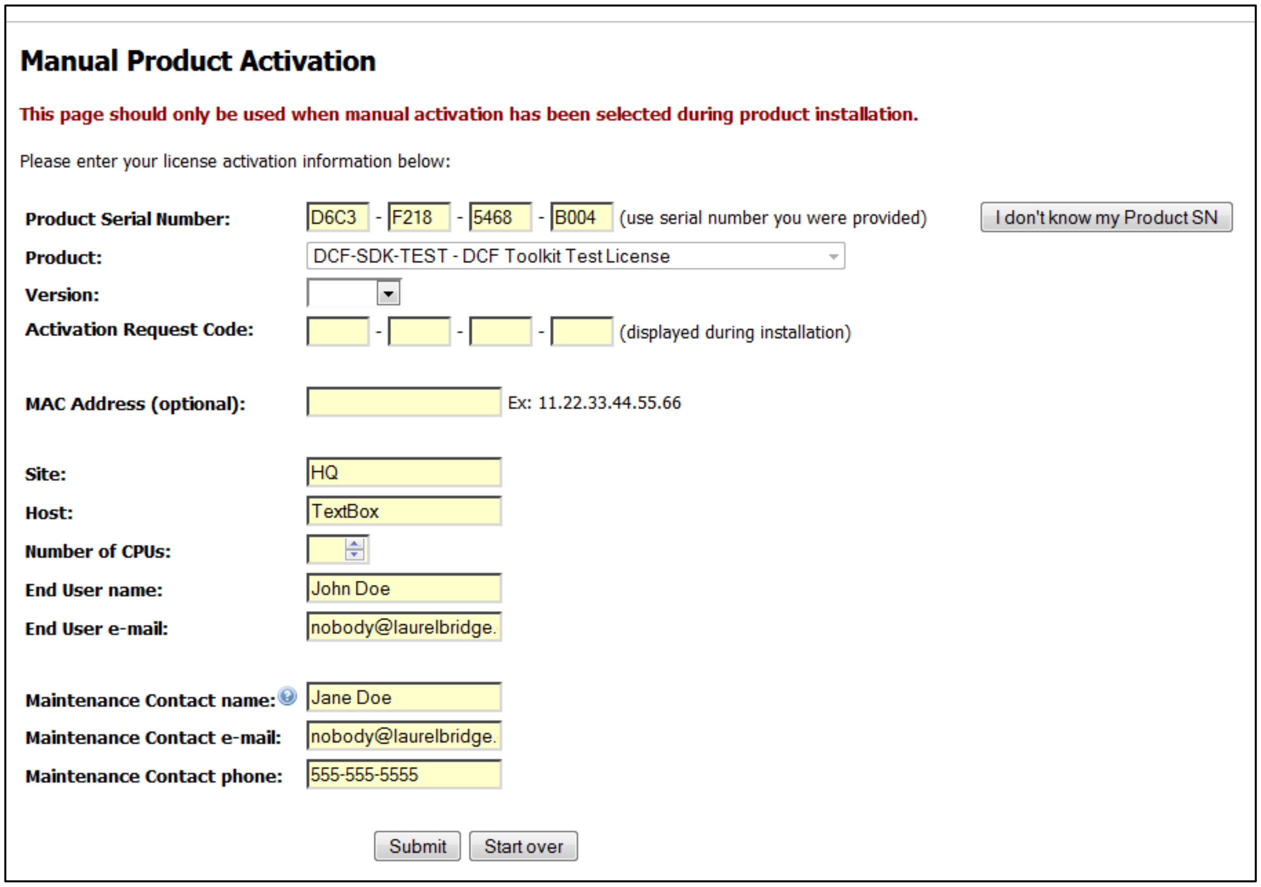
Figure 45: Manual activation of a license

Note the Activation Request Code shown in the form above (in this example, the string “`2230-12A6-C412-EDEA`”). The Activation Request Code is a system identifier code that is generated on the target computer by the DCF libraries. You will need to record this Code so you can complete the next steps.

Using a web browser on a different system, proceed to the Laurel Bridge Software (LBS) product license activation page. There are several ways to access this page.

- Open http://www.laurelbridge.com, click "Customer Access" at the upper right. From the initial screen, select "Activate a Product License". This link is also available on the LBS Support page.
- The license activation page is available directly at: https://www.laurelbridge.com/product_activation.php
- For registered users, if you are logged into the license site, there is also a link: "Activate a Product License" – for an illustration, see Figure 42: Licensing Web Site Options above.

Once at the product license activation page, complete the form: enter the Product Serial Number that was provided to the customer (this was obtained by the OEM when the license was reserved) and the Activation Request Code displayed by the `ActivateDcfLicense` utility during the application installation. Also enter the site and contact information, and the number of physical CPUs for the system that is being activated. See the following screenshot example:



Manual Product Activation

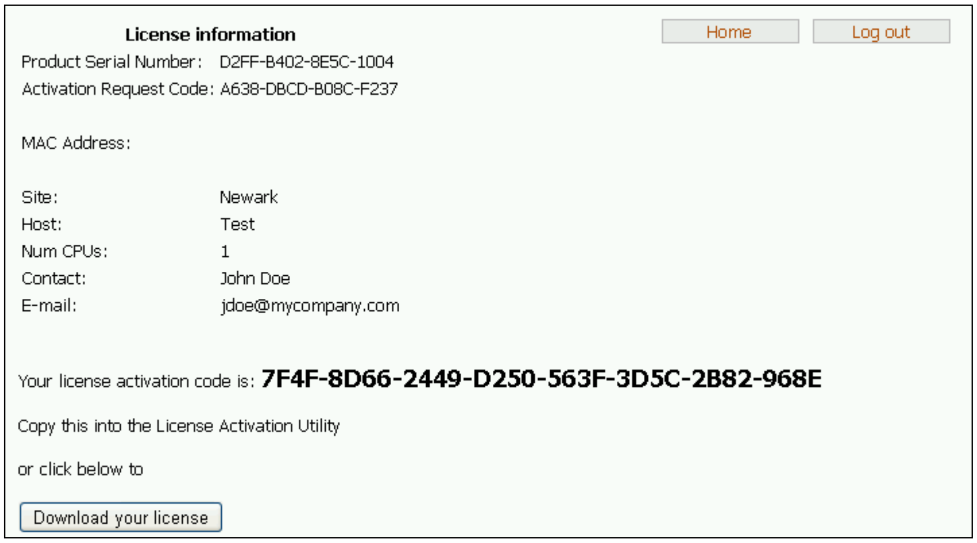
This page should only be used when manual activation has been selected during product installation.

Please enter your license activation information below:

Product Serial Number:	D6C3 - F218 - 5468 - B004 (use serial number you were provided)
Product:	DCF-SDK-TEST - DCF Toolkit Test License
Version:	
Activation Request Code:	- - - (displayed during installation)
MAC Address (optional):	Ex: 11.22.33.44.55.66
Site:	HQ
Host:	TextBox
Number of CPUs:	
End User name:	John Doe
End User e-mail:	nobody@laurelbridge.
Maintenance Contact name:	Jane Doe
Maintenance Contact e-mail:	nobody@laurelbridge.
Maintenance Contact phone:	555-555-5555

Figure 46: License activation web page

After clicking the "Submit" button, a screen similar to the following should be displayed:



License information

Product Serial Number:	D2FF-B402-8E5C-1004
Activation Request Code:	A638-DBCD-B08C-F237
MAC Address:	
Site:	Newark
Host:	Test
Num CPUs:	1
Contact:	John Doe
E-mail:	jdoe@mycompany.com

Your license activation code is: **7F4F-8D66-2449-D250-563F-3D5C-2B82-968E**

Copy this into the License Activation Utility

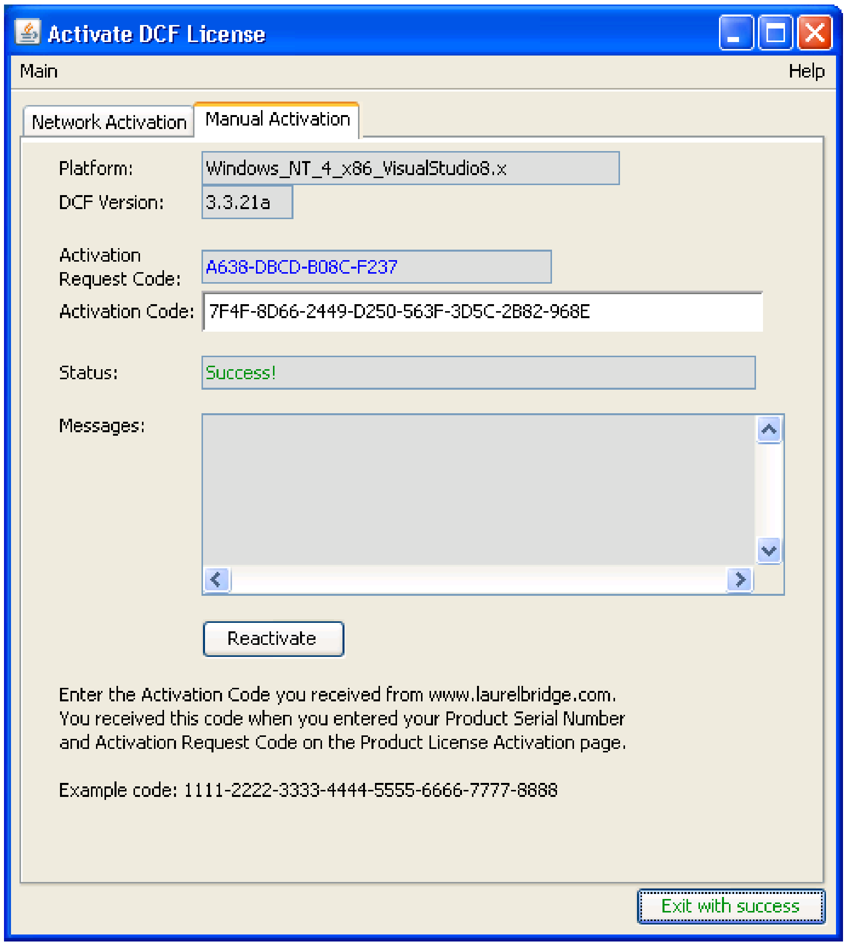
or click below to

Figure 47: Activation Code Display

Because this is the manual process, you must write down the license activation code (in this example, the string “`7F4F-8D66-2449-D250-563F-3D5C-2B82-968E`”).

Enter the provided activation code into the `ActivateDcfLicense` utility form on the system that is being activated.

After clicking “Activate”, that form should look similar to the following:



Platform: Windows_NT_4_x86_VisualStudio8.x
DCF Version: 3.3.21a
Activation Request Code: A638-DBCD-B08C-F237
Activation Code: 7F4F-8D66-2449-D250-563F-3D5C-2B82-968E
Status: Success!
Messages:

Enter the Activation Code you received from www.laurelbridge.com.
You received this code when you entered your Product Serial Number
and Activation Request Code on the Product License Activation page.

Example code: 1111-2222-3333-4444-5555-6666-7777-8888

Figure 48: Manual activation succeeded

After successful activation, the license file on the target system (`%DCF_CFG%\systeminfo`) will have been modified to include the activation code. The DCF software license should now be activated, allowing DCF-based software on the target system to be fully functional.

3.3. Administrative Issues

3.3.1. Reactivating a License

The activation process for a particular license on a particular machine can be repeated at any time if for some reason the license file (`%DCF_CFG%\systeminfo`) is damaged or lost. As long as the Product Serial Number matches a reserved license in the Laurel Bridge license server data base, and the Activation Request Code describes a system that is sufficiently similar to the system for which the original activation was performed, the activation process can be repeated.

License reactivation can be done using either the network or manual approach as described in the instructions above.

3.3.2. Transferring a License

If a customer system is being retired, or the system has been sufficiently modified such that a previously activated license no longer works, the license may be transferred. The Laurel Bridge Software customer web site provides instructions and a downloadable form to request transfer of a license from one system to another.

Once a license has been transferred, it will need to be activated just as if it were a new license. See activation instructions above.

3.3.3. Networking Issues Related to Network Activation

To use the network activation model, the end user system must have Internet access to the Laurel Bridge Software license server. This connection is managed by the license activation utility and is typically called during application installation and the access is invisible to the end user.

The URL accessed is: https://www.laurelbridge.com

The transactions use HTTPS for which the default port is 443.

If an end user’s firewall allows outbound connections using HTTPS, then the license activation utility should function as planned and intended.

If the required outbound HTTPS connection is not supported, then the Manual Activation process described above will need to be followed.

- End of Document -