

WFO-Advanced Overview

Table of Contents

[Title Page](#)

[1. Introduction](#)

[2. Design Drivers and Philosophy](#)

[2.1 Hierarchy of systems](#)

[2.2 Objective](#)

[2.3 History](#)

[2.4 Development Process](#)

[2.5 Design Driver](#)

[2.6 Implementation](#)

[2.7 Summary](#)

[3. Design and Development Methodology](#)

[3.1 Design Methods](#)

[3.2 Design Patterns](#)

[3.3 Programming Languages](#)

[3.3.1 C++](#)

[3.3.2 FORTRAN](#)

[3.3.3 Tcl/Tk](#)

[3.3.4 Other Scripting Languages](#)

[3.4 Incremental Development Methodology](#)

[3.5 Daily Builds](#)

[3.6 Configuration Management](#)

[3.7 Unit Testing](#)

[3.8 Integration Testing](#)

[4. Hardware Architecture](#)

[4.1 Introduction](#)

[4.2 Data Acquisition and Communication](#)

[4.2.1 Local Area Network](#)

[4.2.1.1 Function](#)

[4.2.1.2 Description](#)

[4.2.1.3 Backup](#)

[4.2.2 Satellite Broadcast Network \(SBN\)](#)

[4.2.2.1 Function](#)

[4.2.2.2 Description](#)

[4.2.2.3 Backup](#)

[4.2.3 WSR-88D Radar](#)

[4.2.3.1 Function](#)

[4.2.3.2 Description](#)

[4.2.3.3 Backup](#)

[4.2.4 Other Processors](#)

[4.2.4.1 Function](#)

[4.2.4.2 Description](#)

[4.2.4.3 Backup](#)

[4.3 Database and Application Computers](#)

[4.3.1 Function](#)

[4.3.2 Description](#)

[4.3.3 Backup](#)

[4.4 Workstation](#)

[4.4.1 Function](#)

[4.4.2 Description](#)

[4.4.3 Backup](#)

[4.5 Concluding Remarks](#)

[5. Software Overview](#)

[5.1 WFO-Advanced Subsystems](#)

[5.2 Interprocess Communications](#)

[5.3 Foundation Classes](#)

[5.3.1 Container Classes](#)

[5.3.2 Common Data Classes](#)

[5.3.3 Event Logging](#)

[5.4 Off-the-shelf Software Packages](#)

[5.4.1 OI](#)

[5.4.2 Tcl/Tk](#)

[5.4.3 DMQ](#)

[5.4.4 Informix](#)

[5.4.5 NetCDF](#)

[5a. Interprocess Communication \(IPC\)](#)

[5a.1 IPC Requirements](#)

[5a.1.1 Asynchronous Sends \(Fire & Forget\)](#)

[5a.1.2 Synchronous Sends](#)

[5a.1.3 Message Priority](#)

[5a.1.4 Message Selection](#)

[5a.1.5 Friendly to Event Multiplexors](#)

[5a.1.6 Unlimited Message Size](#)

[5a.1.7 Reliable and Maintainable](#)

[5a.1.8 Performance](#)

[5a.2 History of the IPC Library](#)

[5a.2.1 DEC Message Queue \(DMQ\)](#)

[5a.2.1.1 Design Overview](#)

[5a.2.1.2 Strengths](#)

[5a.2.1.3 Weaknesses](#)

[5a.2.2 Server Based Socket IPC](#)

[5a.2.2.1 Design Overview](#)

[5a.2.2.2 Strengths](#)

[5a.2.2.3 Weaknesses](#)

[5a.2.3 Thread Based Socket IPC](#)

[5a.2.3.1 Design Overview](#)

[5a.2.3.2 Strengths](#)

[5a.2.3.3 Weaknesses](#)

[5a.2.4 Requirement Comparisons](#)

[5a.2.5 Performance Comparisons](#)

[5a.3 Thread Based Socket IPC Design](#)

[5a.3.1 Consider This](#)

[5a.3.2 Object Interaction](#)

[5a.3.2.1 SocketConnection](#)

[5a.3.2.2 AcceptSocket](#)

[5a.3.2.3 DataSocket](#)

[5a.4 Thread Based Socket IPC Implementation](#)

[5a.4.1 Addressing](#)

[5a.4.1.1 Anonymous Addresses](#)

[5a.4.1.2 Named Addresses](#)

[5a.4.1.3 Assigning an Address to a Process](#)

[5a.4.2 Waiting for Events](#)

[5a.4.2.1 UNIX Select](#)

[5a.4.2.2 Waiting for IPC from any Source](#)

[5a.4.2.3 Waiting for IPC from a Particular Source](#)

[5a.4.2.4 Continually Waiting for All Events](#)

[5a.4.2.5 Using Tcl/Tk's Notifier for Event Waiting](#)

[5a.4.3 Client Interface for Sending a Message](#)

[5a.4.3.1 Packaging the Data](#)

[5a.4.3.2 Packaging the Metadata](#)

[5a.4.3.3 Initiating the Send](#)

[5a.4.3.4 Preparing for the Send](#)

[5a.4.4 Data Transmission](#)

[5a.4.4.1 Selecting a Socket](#)

[5a.4.4.2 Writing to a Socket](#)

[5a.4.4.3 Detecting Message Arrival](#)

[5a.4.4.4 Reading the Socket](#)

[5a.4.5 Client Interface for Receiving a Message](#)

[5a.4.5.1 Logical Modules and Message Types](#)

[5a.4.5.2 Use of Receiver Objects](#)

[5a.4.5.3 Delivering the Message to the Receiver](#)

[5a.4.5.4 Interpreting the Message](#)

[5a.4.6 Establishing and Breaking Connections Between Processes](#)

[5a.4.6.1 Making Room for a New Connection](#)

[5a.4.6.2 Initiating a New Connection Request](#)

[5a.4.6.3 Receiving a New Connection Request](#)

[5a.4.6.4 Breaking a Connection](#)

[5a.4.7 Waiting to Write to a Full Socket Buffer](#)

[5a.4.7.1 Detecting a Full Buffer](#)

[5a.4.7.2 Mutual Exclusion](#)

[5a.4.7.3 Synchronous Waiting](#)

[5a.4.7.4 Asynchronous Waiting](#)

[5a.4.7.5 Thread Management](#)

[5a.4.8 Signal Handling](#)

[5a.4.8.1 Registering for Signals](#)

[5a.4.8.2 Controlling Signal Delivery](#)

[5a.4.8.3 Synchronous and Asynchronous Signals and Re-entrancy](#)

[5a.4.8.4 Single Thread Implementation](#)

[5a.4.8.5 Multiple Thread Implementation](#)

[**5a.5 Conclusions and Future Direction**](#)

[**6 Data Acquisition**](#)

[**6.1 Introduction**](#)

[**6.2 SBN**](#)

[6.2.1 Design](#)

[6.2.2 Data Flow](#)

[6.2.3 Process Decomposition](#)

[6.2.4 Tables and Files Accessed](#)

[**6.3 Radar Ingest**](#)

[6.3.1 Class I interface](#)

[6.3.1.1 Design](#)

[6.3.1.2 Data Flow](#)

[6.3.1.3 Process Decomposition](#)

[6.3.1.4 Tables and Files Accessed](#)

[6.3.2 Dial-out Interface](#)

[6.3.2.1 Design](#)

[6.3.2.2 Data Flow](#)

[6.3.2.3 Process Decomposition](#)

[6.3.2.4 Tables and Files Accessed](#)

[**6.4 LDAD Ingest**](#)

[6.4.1 Introduction](#)

[6.4.2 Concept](#)

[6.4.3 CDoT](#)

[6.4.3.1 Design](#)

[6.4.3.2 Data Flow](#)

[6.4.3.3 Process Decomposition](#)

[6.4.3.4 Tables and Files Accessed](#)

[6.4.4 ALERT Decoder](#)

[6.4.4.1 Design](#)

[6.4.4.2 Data Flow](#)

[6.4.4.3 Process Decomposition](#)

[6.4.4.4 Tables and Files Accessed](#)

7. Data Management

7.1 Overview

[7.1.1 System Design](#)

[7.1.1.1 Data Processing and Storage](#)

[7.1.2 Data Retrieval](#)

7.2 Data Routing

[7.2.1 Router Processes](#)

[7.2.1.1 Description](#)

[7.2.1.2 Main objects](#)

7.3 Data Controller Processes

[7.3.1 Description](#)

[7.3.2 Interactions With Decoders](#)

[7.3.3 Main Objects](#)

7.4 Decoding

[7.4.1 METAR](#)

[7.4.2 Satellite](#)

[7.4.3 Models](#)

[7.4.3.1 GRIB](#)

[7.4.3.2 Grids](#)

[7.4.4 RAOB](#)

[7.4.5 Text](#)

[7.4.5.1 Communications Control Block \(CCB\)](#)

[7.4.5.2 WMO Header](#)

[7.4.5.3 Product Data](#)

[7.4.5.4 AFOS Identifier](#)

[7.4.5.5 TextDB Class](#)

[7.4.5.6 Collective Text Decoder](#)

[7.4.5.7 Standard Text Decoder](#)

[7.4.5.8 Data Tables](#)

[7.4.6 BUFR](#)

[7.4.7 Redbook Graphics](#)

7.5 Storage/Formats

[7.5.1 Radar](#)

[7.5.2 Satellite](#)

[7.5.3 Grid](#)

[7.5.4 Point Data](#)

[7.5.4.1 RAOB](#)

[7.5.4.2 METAR](#)

[7.5.4.3 Lightning](#)

[7.5.4.4 LDAD](#)

[7.5.5 Redbook Graphics](#)

7.6 Notification

[7.6.1 Notification Server Classes](#)

[7.6.1.1 class NotificationServer](#)

[7.6.1.2 The Notification List \(class NotificationListEntry\)](#)

[7.6.1.3 The Client List \(class ClientList\)](#)

[7.6.1.4 The New Data List \(Dict<DataAccessKey, SeqOf<DataTime> >\)](#)

[7.6.2 Key Notification Server Algorithms](#)

[7.6.2.1 Message Receipt and Dispatching](#)

[7.6.2.2 Data Notification Arrival](#)

[7.6.2.3 Depictable Registration and Cancellation](#)

[7.6.2.4 Timer Expiration and Low-Priority Processing](#)

7.7 Inventory

[7.7.1 Concepts of time in WFO-Advanced](#)

[7.7.2 Inventory Types](#)

7.8 Retrieval

[7.8.1 Data retrieval in depictables](#)

[7.8.2 Data retrieval in applications](#)

7.9 Purging

[7.9.1 Scour](#)

[7.9.2 Fxa-data.purge](#)

[7.9.2.1 Description](#)

[7.9.2.2 Purging criteria](#)

[7.9.3 Future Plans](#)

8. User Interface

8.1 WFO-Advanced UI Philosophy

[8.1.1 Layout](#)

8.2 User Interface Library

[8.2.1 Overview](#)

[8.2.2 Client Classes and Callbacks](#)

[8.2.3 Caveats - Tcl/Tk](#)

8.3 Software Design

[8.3.1 FXA Process Design](#)

[8.3.2 Classes/Objects](#)

[8.3.2.1 Tiled Display Window](#)

[8.3.2.2 UI Workspace Manager](#)

[8.3.2.3 Product Button Client](#)

[8.3.2.4 Product Button](#)

[8.3.2.5 Scale Menu Manager](#)

[8.3.2.6 Load Mode Manager](#)

[8.3.2.7 Control Classes](#)

[8.3.2.8 Clock Classes](#)

[8.3.2.9 Map Menu Manager](#)

[8.3.2.10 Announcement Classes](#)

[8.3.2.11 Procedure/Bundle Classes](#)

[8.3.2.12 UI KeypadClient](#)

[8.3.3 Interprocess Communication](#)

[8.3.4 Starting FXA and the Main Function](#)

9. Interactive Graphics Capability (IGC)

9.1 IGC Design Drivers

9.2 Process Environment

9.2.1 Child Process

[9.2.1.1 Invocation Arguments](#)

[9.2.1.2 User Interface](#)

9.2.2 Stand-Alone Process

[9.2.2.1 Invocation Arguments](#)

[9.2.2.2 User Interface](#)

9.2.3 Thread of Execution

[9.2.3.1 Initialization](#)

[9.2.3.2 Event Processing](#)

[9.2.3.3 Termination](#)

9.3 Overview of the Main Objects

9.3.1 EventDispatcher

[9.3.1.1 Process Context](#)

[9.3.1.2 Responsibilities](#)

9.3.2 IGC Impl

[9.3.2.1 Process Context](#)

[9.3.2.2 Inheritance](#)

[9.3.2.3 Responsibilities](#)

9.3.3 UI WorkspaceManager

[9.3.3.1 Process Context](#)

[9.3.3.2 Responsibilities](#)

9.3.4 LoopingMgr

[9.3.4.1 Process Context](#)

[9.3.4.2 Inheritance](#)

[9.3.4.3 Responsibilities](#)

9.3.5 DisplayMgr

[9.3.5.1 Process Context](#)

[9.3.5.2 Modes](#)

[9.3.5.3 Responsibilities](#)

9.3.6 DisplayPanel

[9.3.6.1 Process Context](#)

[9.3.6.2 Inheritance](#)

[9.3.6.3 Modes:](#)

[9.3.6.4 Responsibilities](#)

[9.3.7 ViewMgr](#)

[9.3.7.1 Process Context](#)

[9.3.7.2 Responsibilities](#)

[9.3.8 FrameSeq](#)

[9.3.8.1 Process Context](#)

[9.3.8.2 Inheritance](#)

[9.3.8.3 Responsibilities](#)

[9.3.9 Frame](#)

[9.3.9.1 Process Context](#)

[9.3.9.2 Responsibilities](#)

[9.3.10 GraphicDepictTuple](#)

[9.3.10.1 Process Context](#)

[9.3.10.2 Inheritance](#)

[9.3.10.3 Responsibilities](#)

[9.3.11 ImageDepictTuple](#)

[9.3.11.1 Process Context](#)

[9.3.11.2 Inheritance](#)

[9.3.11.3 Responsibilities](#)

[9.3.12 GraphicDepictSeq](#)

[9.3.12.1 Process Context](#)

[9.3.12.2 Inheritance](#)

[9.3.12.3 Responsibilities](#)

[9.3.13 ImageDepictSeq](#)

[9.3.13.1 Process Context](#)

[9.3.13.2 Inheritance](#)

[9.3.13.3 Responsibilities](#)

9.4 Walk-through of Loading a Product

[9.4.1 Load of a product to an empty display](#)

[9.4.2 Load of a product to a non-empty display](#)

[9.4.3 Image combo load](#)

[9.4.4 Multi-load](#)

[9.4.5 Product update \(auto-update\)](#)

9.5 Walk-through of Displaying a Product

[9.5.1 Displaying the current frame](#)

[9.5.2 Preparing frames in the background](#)

[9.5.3 Rendering a graphic tuple](#)

[9.5.4 Rendering an image tuple](#)

[9.5.4.1 Rendering a single 8-bit image](#)

[9.5.4.2 Rendering a single 24-bit image](#)

[9.5.4.3 Rendering an 8-bit image combo](#)

[9.5.4.4 Rendering a combo 24-bit image](#)

[9.5.5 Dimming and fading images](#)

[9.5.5.1 Calculating a combined color table](#)

[9.6 Walk-throughs of Other IGC Operations](#)

[9.6.1 Zooming and roaming](#)

[9.6.1.1 Zooming user interface](#)

[9.6.1.2 Roaming user interface](#)

[9.6.1.3 Zooming and roaming implementation](#)

[9.6.2 Toggling an overlay's visibility](#)

[9.6.3 Sampling](#)

[9.6.4 Stepping and looping](#)

[9.6.4.1 Stepping walk-through](#)

[9.6.4.2 Looping walk-throughs](#)

[9.6.5 Swapping](#)

[10. Depictables](#)

[10.1 What is a depictable?](#)

[10.2 The depictable class hierarchy](#)

[10.3 Support software](#)

[10.3.1 The Painter class](#)

[10.3.2 The Depictor class](#)

[10.3.3 The Xform class](#)

[10.3.4 The XformFunctions module](#)

[10.3.5 Remapping tables](#)

[10.3.6 Fortran Graphics module](#)

[10.3.7 Current usage of Fortran in depictables](#)

[10.4 How depictables work](#)

[10.4.1 How depictables create viewable displays](#)

[10.4.2 A specific example](#)

[10.5 Managing depictables](#)

[10.5.1 Context of depictables](#)

[10.5.2 Adding new depictables](#)

[10.5.2.1 Accessor for a new data type](#)

[10.5.2.2 Formally define a new depictable type](#)

[10.5.2.3 Code for the new depictable type](#)

[10.5.2.4 Enable instantiation of the new depictable type](#)

[10.5.2.5 Enable inventories for the new depictable type](#)

[10.5.2.6 New data key entries](#)

[10.5.2.7 New depictable key entries](#)

[10.5.2.8 New product buttons](#)

[10.5.2.9 Place new product buttons on the menu](#)

[11. Applications and Extensions](#)

[11.1 Applications](#)

[11.1.1 What is the Application Interface?](#)

[11.1.1.1 Interaction with D2D](#)

[11.1.1.2 Launching Applications](#)

[11.1.1.3 Application Characteristics](#)

[11.1.2 Implementation of the Application Interface](#)

[11.1.3 Application Requests](#)

[11.1.3.1 The Load Request](#)

[11.1.3.2 The Export Request](#)

[11.1.3.3 Miscellaneous Requests](#)

[11.1.3.4 Supporting New Requests](#)

[11.1.4 Application Notifications](#)

[11.2 Extensions](#)

[11.2.1 Extension Framework Classes](#)

[11.2.1.1 Interactive Depictables](#)

[11.2.1.2 Editable Elements](#)

[11.2.1.3 The Extension](#)

[11.2.2 Using the Extension Class](#)

[11.2.2.1 The Active State](#)

[11.2.2.2 Editable Element Notifications](#)

[11.2.2.3 Depictable Sequence Notifications](#)

[11.2.3 A Sample Extension](#)

[12. Monitoring and Control](#)

[12.1 Overview](#)

[12.2 Data Monitor](#)

[12.3 Ingest Process Monitor](#)

[12.4 System Performance Monitor](#)

[12.5 Prototype AWIPS Monitors](#)

[12.6 Concluding Remarks](#)

[13. Text Subsystem](#)

[13.1 Overview](#)

[13.2 Text Display User Interface](#)

[13.2.1 Small processes](#)

[13.2.2 Use of Tcl/Tk](#)

[13.2.3 Software Interface to the Text Database](#)

[13.2.4 Text Display Processes](#)

[13.2.5 Key Text Script Files](#)

[13.2.6 The textWish Tcl/Tk Interpreter](#)

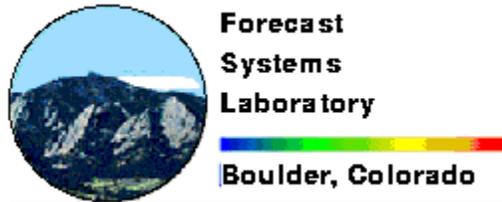
[13.3 Informix Database](#)

[13.4 Text Server](#)

[13.4.1 Text Read Server](#)

[13.4.2 Text Write Server](#)

NOAA Forecast Systems Laboratory Modernization and Systems Development Divisions



This document has been prepared by several members of the FSL WFO-Advanced development and management team. Contributing authors include:

Leigh Angus
Mike Biere
Carl Bullock
Darien Davis
Joanne Edwards
Jim Fluke
Chris Golden
Jennifer Gray
Herb Grote
Sean Kelly
Tom Kent
Sandy MacDonald
Gerry Murray
Scott O'Donnell
Jim Ramer
David Rubio
MarySue Schultz
Joe Wakefield
Denny Walts

CHAPTER 1 Introduction

In August of 1996, the National Weather Service senior management decided to immediately begin exploring ways to quickly exploit the capabilities of the WFO-Advanced system. A primary objective of this activity was to determine what would be required in order to make many of the WFO-Advanced functions available to NWS field offices earlier than previously planned via the current AWIPS schedule. To help understand the impacts of this decision on AWIPS development, schedule, cost, and plans, PRC was tasked with looking into the issues and quickly presenting their findings.

As part of this Task Order, FSL was asked to host a technical exchange session with a small group of engineers and others from both the AWIPS contractor and the government. The intent of this meeting was to promote a better understand of the design, software architecture, and implementation details of the WFO-Advanced system. This meeting was held September 17, 18, and 19, 1996. Participants included staff of FSL, PRC, NWS, and AAO.

A detailed summary of these nearly 3 days of meetings is included in the following document.

Some chapters have been modified to reflect changes in the system as FSL's software has evolved into AWIPS Builds 3.0, 3.1, and 4.0. Notably, Chapter 5a (so numbered for simplicity in updating the document) was added 1 October 1997, describing the complete re-write of the interprocess communication mechanism effected in mid-1997. In all cases, the date of most recent changes (with chapter-level granularity) is reflected at the bottom of each page.

CHAPTER 2 Design Drivers and Philosophy

2.1 Hierarchy of systems

The Forecast Systems Laboratory's (FSL) mission is to transfer science and technology to operational weather services. In its development of weather office information systems, FSL has taken a hierarchical approach. The generic name of FSL-developed, UNIX/X-based weather information systems is the FX Advanced. As shown in [Figure 2.1](#), the title comes from "FSL X-windows." There are "realizations" of the FX Advanced. For example, FSL is working with the Central Weather Bureau of Taiwan to develop a version of the system. Similarly, it has demonstrated a version of the system for the Air Force Global Weather Central. The most effort has gone into the WFO-Advanced, which is the realization of FX Advanced which was tailored for use in the Weather Forecast Offices of the National Weather Service. In the hierarchy below "realization" is "localization." This refers to the ability of the system to be easily tailored for use in a particular geographic area. For example, the WFO-Advanced being used at the Denver National Weather Service office has regional and local scales that are centered on the area of responsibility of the office; it is localized to the Denver area. Higher resolution satellite and radar data are available for the localized scales. It is planned that the forecaster can do a simple stop/restart of the system and localize it to another location, provided that the local data are available in the data base. For example, the Denver forecaster could localize to Tampa, Florida, and review a severe weather case. The other workstations in the office would be operating on the Denver localized real-time data, while the forecaster had a clock set to the case time (perhaps a few weeks earlier), and was viewing data from the Tampa WSR-88D, as well as other Tampa-based data and scales.

```
FX Advanced
-----
Generic name:
  FX Advanced
  "FSL X (-window) Advanced"
Realizations of FX Advanced:
  WFO-Advanced
  CWB Advanced
  AWC Advanced
  Spaceport Advanced
```

Figure 2.1 The FX Advanced family

2.2 Objective

As shown in [Figure 2.2](#), the FSL objective in the FX Advanced project was to develop a weather and hydrological information system suitable for use in operations. By system, it is meant the equipment, software, and even operational procedures needed in a weather forecasting office. In the case of WFO-Advanced, we assumed that the external data sources, such as radar, the satellite broadcast network, and the terrestrial network, are inputs to the system. The system itself includes the ingest processing, database, computers, networks, displays, etc. In short, we include everything needed within the weather office. Although there are many ways to accomplish a mission like "transfer science and technology into operations," FSL has always approached the problem by building and testing systems which are as close to operational as possible. Our experience over 16 years has shown that this is an effective way to identify and prioritize requirements, while at the same time finding the hardware and software combinations needed to satisfy requirements.

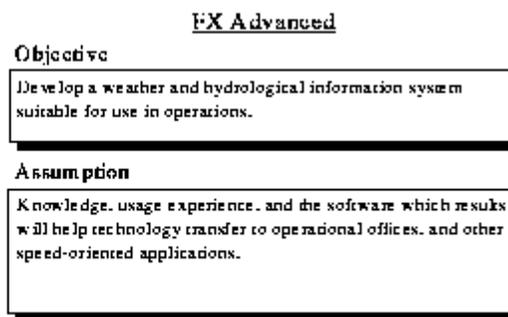


Figure 2.2 FSL's objectives for FX Advanced

2.3 History

The Program for Regional Observing and Forecasting Services, PROFS, began in 1980 to build weather information systems. PROFS developed an approach, now widely called "rapid prototyping," in which systems are built and tested in a real time exercise. The results of the test are used to build a new and better system. This approach results in a better understanding of requirements through time. The importance of various system aspects, such as performance and diversity of capability, can only be determined and prioritized by actual use. At the same time, the "rapid prototyping" methodology leads to progressively better technical and developmental approaches to satisfy the (prioritized) requirements.

[Figure 2.3](#) summarizes the history of system development in PROFS and FSL. It is evident that every aspect of the system has "evolved" over the 16 years. Before discussing the general design principles that came out of this extended evolution, a brief discussion of the evolution of each of the major system components is helpful. Refer to [Figure 2.3](#) for the following discussion:

Figure 2.3 (click to see a larger version)

Evolutionary History of FX Advanced - D2D

Year	'80	'81	'82	'83	'84	'85	'86	'87	'88	'89	'90	'91	'92	'93	'94	'95	'96
System	McIDAS	RT81	RT82	RT83		RT85	← DARE:1 →			← DARE:2 →					FX Advanced		
Interface	Command Line	Light Pen	Separate Touch Screen	Screen GUI		Dist control with mouse			On Screen, Multitouch GUI						Unified GUI		
Typical Display (x,y,z,n)	512x512x4 None built		512x512x4 Rampak			512x512x4 12x6			512x512x16x8						Five pane 900x900x24x12		
Hardware	Harris/6	PDP11	VAX														Hewlett Packard
Operating System	Harris/OS	RSX11	VMS						PC 1166								UNIX
Grid Data	Program Process	Predefined vector graphics															Volume Browser Product Maker
Data Base	60 MB	100 MB	200 MB	300 MB		600 MB	600 MB		1 GB		1 GB						12 GB
Innovation Example	Radar graphic satellite	Radar image	GUI	DRT Rampak fields		Point/Click interface	On screen GUI Family graphics		Grid to graphs Text editor				PUP functions Full ISPAN data access		Product Maker Volume Browser		

System - The first system at PROFS was a McIDAS on loan from Wisconsin's Space Science and Engineering Center. Another available system was the AFOS system of the National Weather Service. These systems were the starting point for design of the RT81 System. This had a command line interface, like McIDAS and AFOS, and included all types of data - such as radar that was not available on McIDAS; of course, the AFOS had no image data. The "RT" in the title refers to real time, with the approach being that the system was tested by bringing in NWS forecasters in the Fall of 1981 to use it, and make suggestions for improvement. The results of the RT81 test were used to completely design a new system for use in the Summer of 1982, designated RT82. This system was the first one that included a graphical user interface, which was built using the VAX/VMS architecture developed by Digital Equipment Corporation. The VAX/VMS systems were the primary development platform for PROFS/FSL through the late 1980s. A much more sophisticated system was built in 1983, which was oriented toward the ability to improve severe weather warning capability. Experience with this system showed that the need to issue convective weather warnings is a "design driver" for NWS weather information systems, a subject that will be discussed in Section 2.5. By 1986, a much more effective method of evolving the weather information systems was in place - a system called DARE was installed and operating at the Denver Weather Service Forecast Office. By being in operations, the system was used continuously, with the associated advantage of continuous

feedback to evolve improvements. Furthermore, the tests were not contrived to be close to operations - it was the actual operational environment toward which we were designing. DARE, including a significant upgrade in 1990, was in operation at Denver and Norman for many years, and is only now being decommissioned at Denver with the coming of WFO-Advanced. During 1987, work commenced on a PC-based system. This system was built with a UNIX operating system and used a special graphic board that plugged into the AT bus. The system helped FSL developers learn how to use UNIX effectively to get the performance needed for weather systems.

Interface - It became very obvious in the RT81 test that a key to weather information systems was development of a user interface that allowed the large and diverse variety of products and capabilities to be easily accessible. A command line may have worked for systems with limited products (McIDAS) or limited data types (AFOS), but a much more sophisticated approach was needed when diversity and complexity of products increased. The RT82 system had a separate display, equipped with a light pen, which was dedicated to the menu of products. A touch screen was used in RT83. Light pens and touch screens are limited in capability because the technology requires a comparatively large amount of screen space for each selectable item; later systems have used the now-common mouse/pointer mechanism. The RT85 system had a single high resolution dedicated menu screen that drove two display screens. This configuration, while an improvement over the touch screens, generated confusion and difficulty because of having to always be cognizant of which display would be affected by an interaction. It was determined that having the menus right on the display screen was preferable; this was accomplished in the development of DARE, which proved to be a great improvement over RT85. Tear-away menus allowed forecasters to configure the system and its menus to their preference and to tailor the menus to the particular situation they were facing. DARE had different menus for each scale, and floating menus for applications such as cross sections, interactive skew-T, warning generation, etc. The complexity of the menus became an issue, with orphan tear-always, and menus which were unrelated to the product which was displayed. For example, an interactive cross section generator could still hang around after the related cross section had been replaced by (say) a skew-T. Because of all these experiences, the FX Advanced menu was designed to be generally oriented to a single screen and unified through scales. Thus, whether one is on a very large scale, such as the hemispheric, or a small scale, such as the state scale, a single menu handles all access. If a product is unavailable, it is simply dimmed. In summary, the result of the experience was that menus have to tree rapidly and easily to a very large product set, but must be available at high level in simple form, and act unambiguously on the main display. WFO-Advanced does this. Although the reaction of many people to such a requirement is that this is obvious, most meteorological systems extant today grossly violate the rules of menu simplicity and directness.

Display - A salient feature of weather is that it is dynamic. Thus, it is not surprising that weather forecasters are always pushing for more and better motion capabilities. They want animations that load fast, have high resolution, and run long enough to see the phenomena of interest, which can range over time scales from minutes to weeks. A product, such as an infrared image of North America, can be derived from several megabytes of data which is transmitted from a

geostationary satellite. The display problem is to convert that data to a stream of images on the screen. With the computer technology of the 1980s and 1990s, doing this to the satisfaction of the users is difficult.

It was once stated that it was fortuitous that PROFS was able to find dedicated hardware that allowed easy and fast animation of high resolution images, and that such capability was not easy in a UNIX/X environment. This view reverses what actually happens in the design process. In the early 1980s, starting with the requirement, it was very difficult to see how the display requirements could be met. PROFS pioneered the use of the Ramtek frame buffer, with its size, pixel-replicate zoom, and rapid display list processing, to meet the requirement. PROFS engineers worked closely with Ramtek to improve their system designs to meet the weather information processing needs.

One of the first things to do in the design of a meteorological system is to start with the available hardware and software, and analyze the factors which will allow access, load speed, and animation which will meet requirements. In the case of WFO-Advanced, much of the database, including imagery and grids, is located on the database processor. Thus, the first access must bring the data over the network to the local memory. In the memory, it is stored in its database form; a pixmap suitable for X window display is created from the memory-resident data. The pixmap is typically not a one-to-one load from storage format. Rather, there will be operations to convert the existing file form to the appropriate scale, zoom, etc. Thus, the loading of (say) an image consists of many steps: determine where the disk file is, bring it over the network, access it in memory with a program that converts it to the pixmap appropriate for display, etc. In the design of FX Advanced, each of these steps was optimized and redesigned and optimized again in order to meet the performance requirements. We are continually looking for better and faster ways to perform these crucial operations.

As seen in [Figure 2.3](#), the original imagery (in the early 1980s) was presented as four-frame loops, with a resolution of 512 by 512 pixels. The depth of the frame buffers was typically 8 bits. During much of the 1980s, performance was achieved by specialized frame buffers, in which the entire animation sequence was stored (in DRAM) and operated on (e.g. pixel-replicate zoom). When forecasters were given a choice of how many frames to use in their animations, they would choose the longest animation that could be loaded in a "reasonable" amount of time. Thus, on DARE 1, they could load up to a 32-frame loop, but rarely did, because it would take minutes to load. The time one is willing to wait varies with individual, but it is typically tens of seconds. Thus, in the RT82 and RT83 systems, it took about 30 seconds to load a four-frame loop, and that was the typical usage. In RT85 and DARE 1, an 8-frame loop could be loaded in about 30 seconds, and that was the most common usage. DARE 2 allowed faster loads, partly due to an image disk, and partly due to other optimizations. The result were that more forecasters used longer (e.g. 16- and 32-frame) loops, although they sometimes chose shorter loops for a variety of reasons. The FX Advanced currently loads a 900 by 900-pixel image at about 0.8 seconds per frame on a cold hit (i.e. when the product is not available in RAM or disk cache), which means that a forecaster willing to wait 30 seconds might reasonably load a 24-

frame loop. Recent experience in Denver and Boulder has shown that the users are using 16 to 24 frame loops fairly commonly.

Hardware - The growth in the capability of hardware, such as processors and memory, is a well known but nevertheless amazing story during the last 16 years. PROFS started with a PDP 11 in the RT81 system, progressed to a VAX 750 in RT82, and stayed pace with the evolution of the DEC VAX series of processors during the 1980s. The PC 386 with an Omnicomp graphics board was used in the PC workstation. Hewlett Packard equipment, with the HCRX-24 graphics card, is being used in the FX Advanced. The most interesting thing to say about the hardware evolution of the PROFS/FSL systems is that large increases in raw computing power translates into relatively small increases of functional capability.

Operating System - The operating system used in the VAX systems was VMS. Starting with the PC workstation, FSL has used UNIX operating systems.

Grid Data - The early PROFS systems adapted from both AFOS and McIDAS. AFOS used predefined vector graphics, while McIDAS used grids to generate products "on the fly." During the early 1980s, PROFS used predefined vector graphics for a number of reasons, including performance (particularly load speed) and the availability of AFOS data in vector format. The use of grids is more flexible, both in the products that can be created, and how they can be tailored for use. For example, the use of 60 gpm contours on 500 mb charts is quite standard; for many years the fax products and then the AFOS products of NWS used these contours. Thus, when zooming into small spatial domains with data in vector format, the contours often disappear from the screen because the space between contours is wider than the geographic domain. With grids, we have the option of contouring on the fly, allowing the density of contours to increase as the user zooms in. In the DARE system, an application called "Grid to Graph" was used to generate a selection matrix for grid products. The system was limited in product selection, and cumbersome in that it often generated a large matrix of selectable products. The product was listed on the left side, with a "green time" (the time of the latest available product) for each prediction time listed along the abscissa of the selection matrix. The "Volume Browser" of FX Advanced represents a significant improvement because it simply shows an inventory of the prediction times. It is more organized in its call sequence. Furthermore, a new capability, called the "Product Maker" allows the user to use mathematical operations on one or more grids to generate fields of from 0 to 4 dimensions (x, y, z, t). A simple example of the use of the Product Maker would be to calculate the difference field between the relative humidities from two different models. It is clear that such operations require the availability of grids rather than vector graphics.

Data Base - It is clear from [Figure 2.3](#) that the history of the data base sizes has tracked the similar growth in disk storage available for a fixed cost. In real dollars, the cost of a 60 MB disk in 1980 is similar to the cost of 12 GB of disk storage in 1996. One use for the increasing disk space is to make more model grids available. The model grids require more space than vector graphics, and the models themselves have increased fairly rapidly in resolution, resulting in big increases in storage requirements. This trend will continue; a single run of a national 10 km

resolution meteorological model will generate about 10 GB of data (which of course could be compressed). Such models are now being tested at NCEP and FSL, and may be available operationally as soon as 2000.

A second increase in data storage that occurred was in the resolution of imagery (mainly satellite and radar) and the length of animations of these. As discussed above, in the early 1980s, a four-frame loop of 256 by 256 or 512 by 512 images has been replaced by typical loops being 900 by 900 pixels, with typical sequence lengths of 12 to 24.

2.4 Development Process

In the development of systems, FSL has promulgated a "stair step" approach, as shown in [Figure 2.4](#). Although a lot of effort is spent at the beginning on requirements and design, the detailed requirements and design are not completed initially. Rather, these flow out of the process of rapid prototyping, with a well defined sequence leading toward more complete and detailed requirements, design, and implementation. These three system attributes (requirements, design, and implementation hardware and software) are developed in parallel, and not sequentially. FSL's experience since the early 1980s has shown the power of this parallel approach for systems development is not well understood. As a further illustration, a sequential process is probably best for building a system which is very similar to previous ones for which the builders have much experience (e.g. an office building). The fundamentals of requirements and design are very well understood at the outset for construction of buildings. Conversely, the building of the first moon rocket or the first atomic bomb are examples where history has shown that the requirements, design, and implementation all developed in parallel. The latter case is well documented in *The Making of the Atomic Bomb* by Richard Rhodes. The FSL "stair step" approach is labeled above the line for the general case in the Figure, with the specific WFO-Advanced equivalent shown below the line.

As shown in [Figure 2.4](#), a system typically starts as a collection of software or subsystems that do parts of the job. These are chosen to test high risk areas, and to learn more about critical design questions. FSL demonstrated a limited system of this type at the American Meteorological Society meeting in January 1993. An intense effort which began in early 1994 culminated in a "working" system near the end of 1994. This was a development system including the major subsystems. The development systems were redesigned, improved, and rebuilt on an aggressive 2- month cycle during FY95. The purpose of this phase is to force an accommodation between the major subsystems. For example, the major processes like the executive ("fxa"), the database, the graphics controller ("IGC"), the interprocess communications, and the applications can be designed at a high level, and developed somewhat independently. The aggressive build and rebuild cycle of the development phase allows a natural global optimization of the system, in which each system is progressively modified to fit better in the environment created by the other subsystems. This approach often

results in improvements which would not have been possible if the system had been designed in detail at the beginning and built as if from a blueprint.

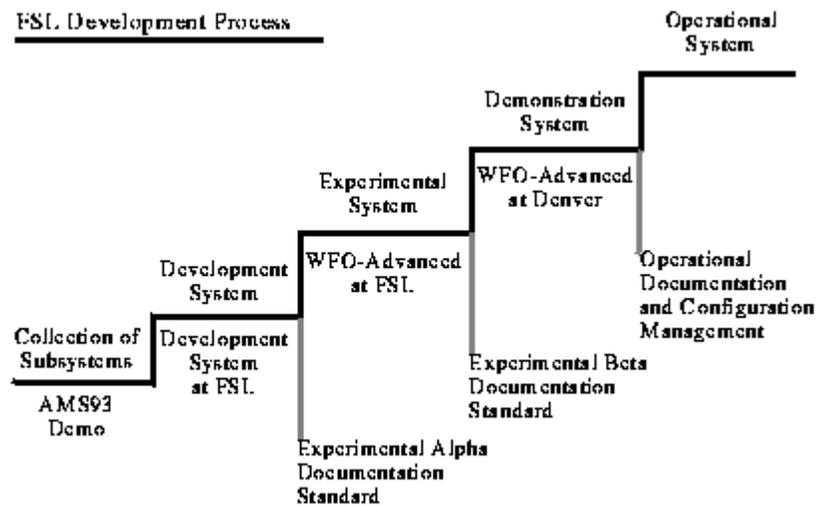


Figure 2.4 The FSL development process

The Experimental System for WFO-Advanced came into being in Fall of 1995 when it was tested extensively by bringing National Weather Service forecasters to FSL. The tests are documented in An Overview of Evaluation Results from the WFO-Advanced Real-time Forecast Exercise: RT95 by Roberts et al., 1996.

Between the development phase and experimental phase, FSL requires meeting of an Experimental Alpha Documentation Standard (essentially, in-line documentation), and similarly, the transition from Experimental System to Demonstration System requires meeting a more rigorous Experimental Beta Documentation Standard (adding more external documentation).

The Demonstration System was implemented at the National Weather Service Forecast Office in Denver in June of 1996.

2.5 Design Driver

With 16 years of experience at designing, building, testing and starting the process all over again, what has FSL learned? Surprisingly, there is a general principle, not immediately self evident from visiting weather offices, that seems to differentiate "good" weather information systems from "bad," where goodness and badness are related to the satisfaction of users in operational or simulated operational environments:

PRIMARY DESIGN DRIVER FOR WEATHER INFORMATION SYSTEMS

*AVERAGE TIME NEEDED TO DISPLAY A TYPICAL COMPOSITE, INTEGRATED PRODUCT - FROM
'INTENTION' TO 'AVAILABLE'*

In the above principle, a composite product is one which may have more than one field of the same type (e.g. 500 mb height and vorticity). Integrated refers to fields of different types (like satellite and surface observations) being displayed on the same screen.

A discussion of why this principle is so important as a design driver is given in *Design Considerations of Operational Meteorological Systems: A Perspective Based Upon the PROFS Experience*, by A. E. MacDonald, 1985. In that paper, particular importance was attached to speed of use as it related to the convective weather warning process. However, the principle stated above has been found applicable to all weather office operations in PROFS and FSL experience.

A few comments about the principle are necessary. First, the time needed between intention and available includes the Graphical User Interface (GUI) interaction. It was learned early in the PROFS workstation developments that there are many products for which the GUI access takes a very long time. For example, the forecaster wants a 300 mb vorticity chart; where in a complicated menu structure can it be found? The forecaster tries working down into the display tree of a particular model and discovers that there is no analysis, only 12, 24 and 36 hour progs. Trying another model, the vorticity analysis may be there for 500 mb, but not 300 mb. Complicated and difficult menus generate user confusion, and make their jobs much more difficult. In addition, the system may require a great deal of interaction, where the user specifies such things as display time, display mode, number of products in time sequence, etc. before the product is displayed. All these can result in the user being unable to rapidly tell the system the product which is desired. WFO-Advanced uses extensive and user-tested defaults to make minimal product specification the norm, while allowing extensive tailoring if needed.

Of course, once the product is selected (and specified), the system must display it. This is often a very complex process; for example, a 300 mb vorticity chart in WFO-Advanced is made from a grid, requiring data access through the network to two different fields (u and v component of the wind) stored on the database machine's disk, computations, contouring, and special display operations, all occurring on the fly before the product comes up. Furthermore, it is possible to call, in a single mouse click, upwards of 100 products (e.g. family graphics for a model run).

In short, there is much that happens between 'intention' and 'display.' Although things which typically show up in requirements, such as the speed of image loads, are quite important, other aspects, such as menu confusion and excessive specification demands (which are harder to cast as requirements), affect the usability of the system at a fundamental level.

It is important (at least conceptually) to define a metric which can be used to determine how well a weather information system meets the principle stated above. One characteristic of

operational weather information systems is that some products are used extensively tens or hundreds of times in a shift while others are used very rarely. (The fact that a product is used rarely should not be construed as a measure of its importance; when a dangerous ice storm threatens, the forecaster is likely to be looking at low level temperature and moisture products that would ordinarily be of little interest.) A measure of the speed of a weather information system should take into account the frequency of use and measure the display latency for an "average" product call. [Figure2.5](#) shows an example of how this could be done. First, the products are tabulated and placed in a histogram of usage based on actual logs. Then, a representative set of products (e.g. 100) is selected at random. Next, a typical user of the system (not the most adept, but rather the most representative) is timed in calling up the 100 products. The lower the time needed, the more suitable the system is for operational use. We believe that the WFO-Advanced performs very well on a test of this type.

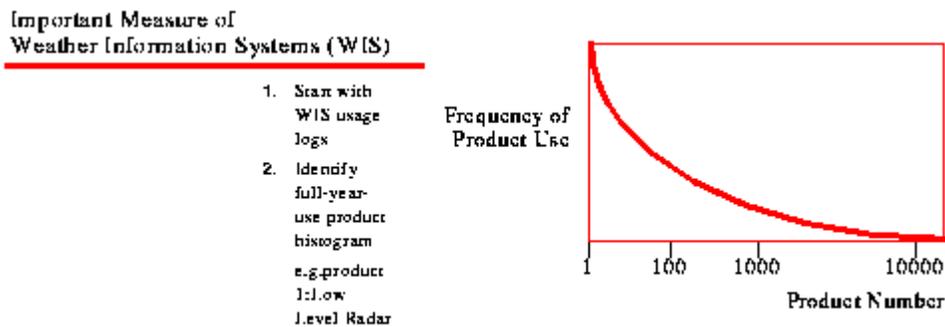


FIGURE D.1 A Measure of Weather Information Systems

Figure2.5

It is worthwhile to discuss briefly why the speed of display is so crucial in operational weather information systems. A good place to start is to observe how the forecasters use the system. A good way to distinguish weather office use from other computer users is to plot typical usage on a two axis display, as shown in [Figure2.6](#). The ordinate in this chart represents "Product Complexity", while the abscissa represents the "Display/Manipulation Ratio." Product complexity includes such things as the resolution of the images, the need to integrate different types of data (e.g. satellite images and model graphics), the use of motion, availability of domains or scales for display, etc. It is asserted that weather displays are extremely complex, as complex as any within the field of computer information systems. Although there are systems such as AFPS being developed for graphical forecast generation, the aspect of weather office usage which dominates is the study of the product displays. Thus, a forecaster trying to get ready for a possible severe convective storm will be looking at many products atmospheric stability, low-level helicity, the ever important doppler radar imagery, etc. This may go on for hours, with the culmination being a graphical specification of the warning area which occurs on a time scale of minutes. In current weather offices, the products are often generated on a word processor, which is the opposite of the weather information display aspect discussed in this section. Specifically, a word processor file is often quite small (a few kilobytes) and is usually manipulation-dominated (e.g. typing).

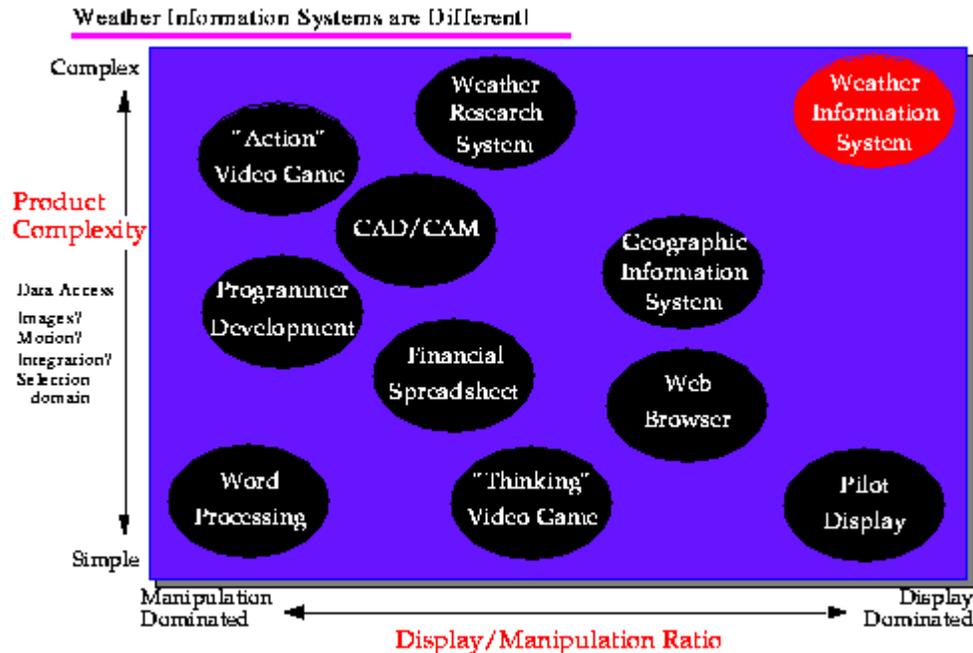


Figure 2.6 System complexity

Referring to [Figure 2.6](#), it is clear that weather information systems are quite different from other computer information systems. The experienced weather forecaster likes to look at a number of complex products and to rapidly assimilate the information available from them. A CAD/CAM system may be complex, but is generally the object of extensive manipulation. Geographic Information Systems have a lot in common with weather information systems, but the ground is static in comparison with the dynamism of the atmosphere. This atmospheric dynamism makes motion and other complexity necessary.

It is noteworthy that weather researchers do not typically use weather information with the rapid display needs typical of an operational weather office. Researchers are more likely to spend a long time on a product, studying it and analyzing complex features. Thus, the weather research community has often chosen flexibility over speed in the systems it has developed (e.g. McIDAS and GEMPAK).

2.6 Implementation

There are many ways to make a weather information system speedy and easy to use. For brevity, we will present only a few of the methods used in the FX Advanced system. First, we will discuss the use of coherence on display and the GUI, then we will discuss data access.

Space of Display Products



Coherence - users often want to view products which are related to previously viewed products

Note: FX Advanced takes advantage of *both* display buffer *and* GUI coherence

Figure2.7 Coherent organization of database

[Figure2.7](#) presents an abstract representation of a data base. In the Figure, the model data are shown in one area of the database, and the radar in another. A forecaster looking at a 500 mb height field prog valid at 12 hours is often going to want the 24, 36, 48 etc. prediction fields for the same product in sequence. The GUI in FX Advanced allows a single click to call the complete prognosis sequence. Further, since forecasters typically look at entire model runs, the Volume Browser and Family Graphics parts of the GUI make multiple product calls very easy. In this case, FX Advanced takes advantage of usage coherence to make GUI selection faster and easier. Another type of coherence is in the buffering of the products. FX Advanced brings a grid of data for a model field from the database into the local memory of the computer being used to display a product. The grid in memory is operated on to generate a pixmap which is used by X from main memory to generate the display. A gridded field is already available in main memory when a user request is generated; it displays much faster than if the system must go through the network to the database. It is faster still if the pixmap for the requested product is already in main memory. Like the cache model of computing memory, the FX is made faster by a hierarchy of product caching that is related to typical usage.

Another way that FX Advanced is speedy is the way the data are accessed from the disk. As shown in [Figure2.8](#), information concerning the location of the product on the disk is kept in the display machine's main memory. There is a data structure related to the CDL (descriptive text) files used by netCDF which has information about the grids, such as products available (inventory) and file names. Thus, a single access over the network is often all that is necessary to move a set of grids or other products from disk to main memory. This is made possible for grids by organizing the data in large files. As shown in [Figure2.8](#), a file for the Aviation 211 grid for a particular time (19/00Z) is organized with the pressure and theta levels varying most rapidly, the fields (u component of wind, v component of wind, temperature, etc.) varying next most rapidly, and the prediction times as the third level. A particular product or set of products is called with speed and simplicity by taking advantage of use coherence (i.e. only one file access gets the whole model run which is often used as a whole).

FX Advanced Data Access

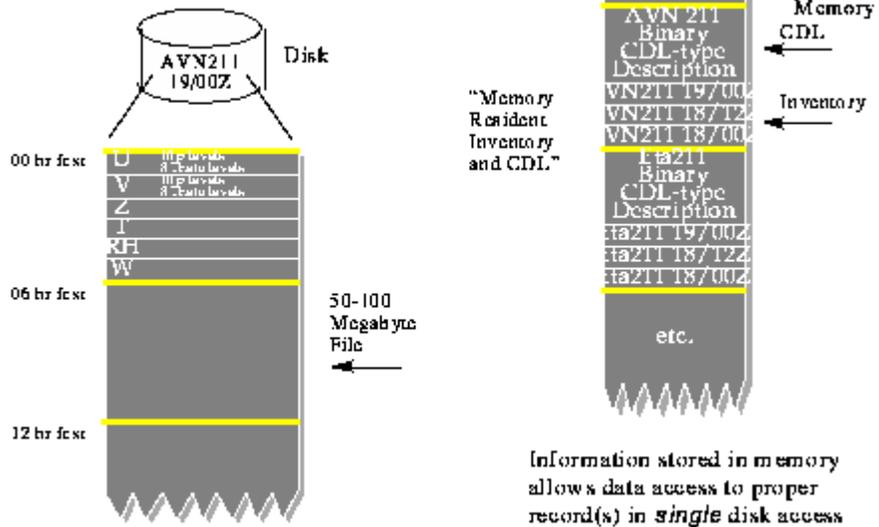


Figure 2.8 FX Advanced data access

2.7 Summary

A summary of the design philosophy of the FX Advanced is that it used an advanced GUI and designed its database and software to make access to complex products easier and faster.

CHAPTER 3 Design and Development Methodology

3.1 Design Methods

WFO-Advanced is a rather large system, encompassing data acquisition, storage and retrieval to a variety of file formats and to a relational database, user interface and display, and monitoring and control capabilities. No one design method handles all of these problem domains particularly well, so our guiding philosophy has been to use the design method appropriate to the problem at hand.

In general, this has resulted in dataflow design techniques being applied to the data acquisition components of WFO-Advanced, and object-oriented design techniques being applied to the display and user interface components, with a blend of the two being used in the data management area.

No single, formal object-oriented design method was applied to the object-oriented components of WFO-Advanced. Generally, our initial design efforts concentrate on the static aspects of design, establishing well-defined public class interfaces based on responsibility-driven class decompositions, and relying on well-known principles of data abstraction and information hiding. If the dynamic interactions between objects are a particularly important part of a design problem, a representative scenario will typically be introduced as part of the initial design discussions. Generally speaking, we try to avoid complicated inheritance relationships, particularly eschewing all but the most trivial uses of multiple inheritance

3.2 Design Patterns

The WFO-Advanced design has been influenced by the emerging discipline of design patterns, which provides a language for discussing reusable design components within the context of an object-oriented design. A design pattern is a stylized set of classes, objects, and interactions, working to solve a common problem. The language of design patterns provides a way for designers to discuss designs and reusability at a level of abstraction above that of an individual class.

For example, probably the most common design pattern used throughout WFO-Advanced is the Singleton pattern, which provides a standard way to initialize and provide global access to an

object, guaranteed to be unique throughout a process. When a developer identifies a particular class as a Singleton, a large amount of information is summarily transferred to other developers about the use of that class.

3.3 Programming Languages

3.3.1 C++

The C++ programming language is the primary language of WFO-Advanced. We use C++ both for its support of object-oriented programming, and as a "better C." Our use of inheritance is generally limited to the "forest" approach of using a large number of disjoint base classes, rather than the "tree" approach of having a system-wide set of base classes inherited from a single class. We rely heavily on the C++ template mechanism, using it as the basis for a set of common container classes.

3.3.2 FORTRAN

Particularly for meteorological calculations, we reuse a body of legacy FORTRAN libraries from previous development projects within FSL. Little new development within WFO-Advanced was done in FORTRAN.

3.3.3 Tcl/Tk

Tcl (tool command language) is a scripting language developed by John Ousterhout. In conjunction with a co-developed user-interface tool kit called Tk (tool kit), Tcl/Tk provides a powerful, easy-to-use environment in which to develop user interfaces. The text workstation user interface was developed with Tcl/Tk, and we plan on converting the graphic display interface to Tcl/Tk.

3.3.4 Other Scripting Languages

Some parts of WFO-Advanced, particularly startup scripts, and some background processes run as cron jobs, are written in various scripting languages including the Bourne UNIX shell (sh), the Berkeley UNIX shell (csh), and perl.

3.4 Incremental Development Methodology

The development model used by the WFO-Advanced project is one of iterative, incremental development cycles. Our approach is similar to that sometimes referred to as the spiral model of software development, although the spiral model is more risk-driven, while our development cycles are more requirement-driven.

Our normal development cycle is a period of two to four months. For each cycle, a set of requirements is established, generally as a delta set of functionality from the previous development cycle. The requirements are then prioritized, and a set of prioritized task

assignments is generated. Design authority is generally delegated fairly early in the process, so that task assignments encompass both design and implementation aspects.

3.5 Daily Builds

Rather than integrate as a final phase of the development cycle, WFO-Advanced integration is effectively a daily activity throughout the development cycle. When a developer has completed some set of functionality, the associated files are released by the developer from that developer's private staging area to a shared baseline area. A nightly "master build" is done on the baseline, ensuring that at least the interface specifications are consistent, and the syntax correct.

No automated integration testing of the resulting system is done as part of the daily build, but after a clean build, the resulting system is occasionally "pushed" to another system for manual integration testing.

3.6 Configuration Management

For source code configuration management, WFO-Advanced uses the BCS (Baseline Configuration System) software package. BCS uses RCS for individual file configuration management, but extends RCS to include the management of a shared baseline source directory, with each developer having a private staging area. Files which are locked by a developer are available for modification in that developer's staging area. Files which are not locked are represented in the staging area as symbolic links back to the real version, in the baseline directory.

3.7 Unit Testing

The extent and depth of unit testing is rather uneven across the project. The developer of a module is responsible for unit testing where feasible. In a number of cases, the interfaces to a module are so numerous or complicated that the development of unit-test scaffolding for that module is deemed too expensive, and testing is deferred to the integration testing level.

In general, there are unit test programs for testing inventory and retrieval for each WFO-Advanced data source. Most data decoders also have test drivers. There is also a suite of unit test drivers for checking the various static info files. There is little unit testing scaffolding in the

display and user interface components, due at least in part to the complexity and dynamic nature of the protocols between components.

3.8 Integration Testing

Integration testing of WFO-Advanced is a manual procedure. In short, someone sits in front of the system and uses it, noting any problems encountered. While this may sound inadequate, and does need be formalized a bit more than it is, it is surprisingly effective. Within FSL, daily weather briefings are given on the system, and new features and capabilities usually receive early attention as part of these briefings.

There is currently no automated integration or regression testing of WFO-Advanced. In a system like this, with flexible user requests and real-time data interactions, significant manual integration testing will always be a necessity, but some significant automated testing could be added. This is one of the anticipated benefits of redesigning the user interface with Tcl/Tk, which should make it easy to add at least some automated testing via scripts.

CHAPTER 4 Hardware Architecture

4.1 Introduction

The WFO-Advanced system installed in Denver in May 1996 comprises four full function graphics workstations, two text-only workstations, two data servers, an application processor, and several smaller processors for data acquisition. The system configuration is flexible and can easily be extended to include more workstations, application processors, or acquisition processors.

The hardware architecture for the WFO-Advanced system is illustrated in Figure 4.1. With minor exceptions, the system configuration and hardware for the WFO-Advanced are as specified for AWIPS by the AWIPS contractor. The function, description, and fail-over concept for each system component is described at a high level in this chapter.

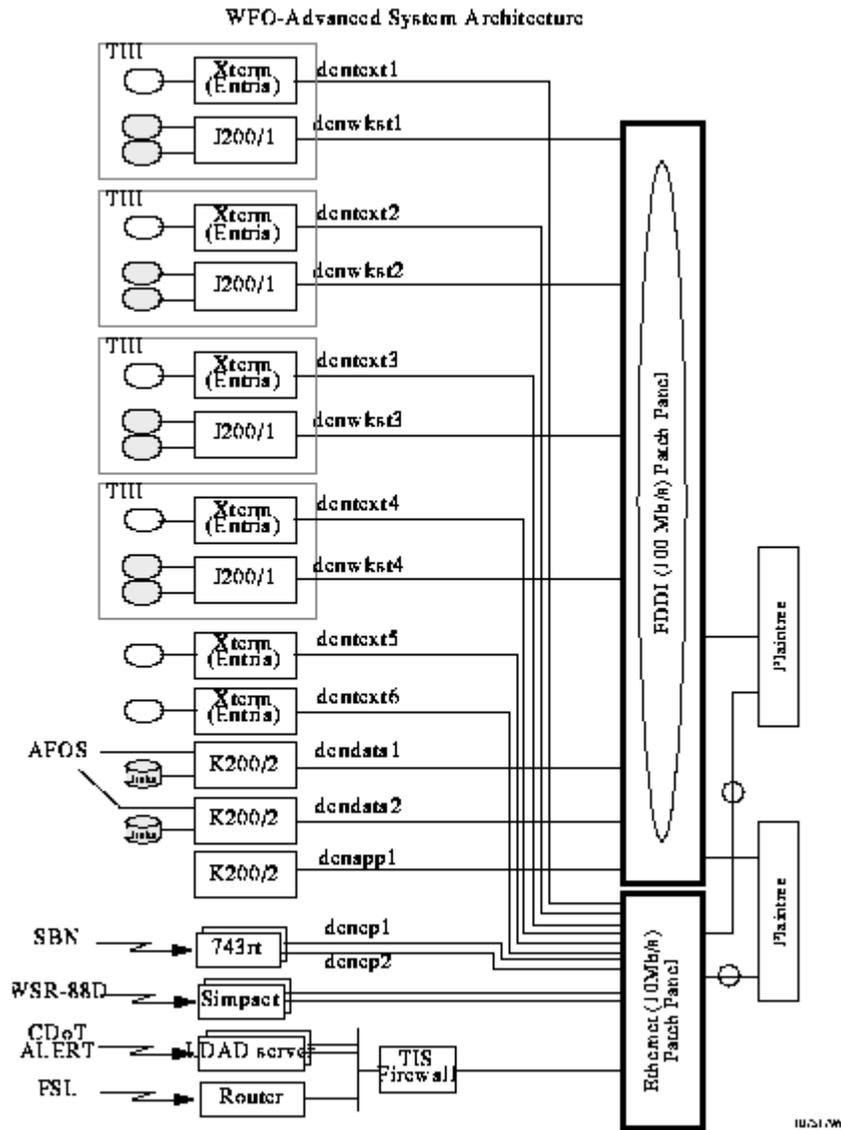


Figure 4.1 Advanced Hardware Architecture

4.2 Data Acquisition and Communication

4.2.1 Local Area Network

4.2.1.1 Function

The network provides the interconnection between the various processors in the system and for this discussion does not include external communications interfaces.

4.2.1.2 Description

The WFO-Advanced network consists of two major components: the 100 megabit/second FDDI ring and the 10 megabit/second ethernet bus. All of the graphic workstations, database and application computers, and Plaintree ethernet switches are connected to the FDDI ring. To simplify reconfiguration, the actual FDDI connections are made to a patch panel that is wired to provide the ring architecture. Optical bypass switches are provided for all computers (except the Plaintree switches) to allow FDDI communications automatically to bypass failed computers. The ethernet is implemented within a Plaintree ethernet switch box that also provides the interconnection between the ethernet and the FDDI.

4.2.1.3 Backup

Two Plaintree ethernet switches are connected to the network, each having connections to all ethernet devices. Should one switch fail, the second will automatically take over control of the network. Should failure of the ethernet switch not be detected by the backup ethernet switch, then the faulty switch may need to be manually shut down.

4.2.2 Satellite Broadcast Network (SBN)

Refer to Figure 4.2 for an overview and additional detail of the acquisition processors.

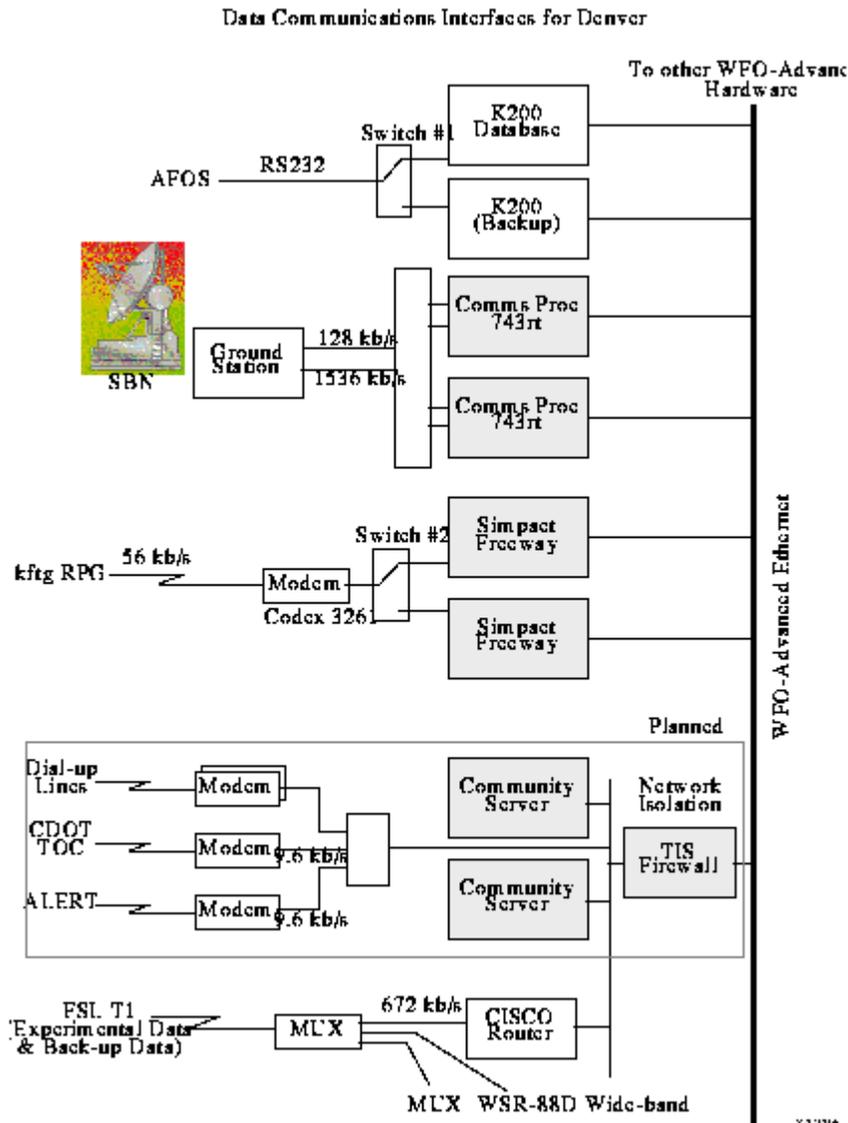


Figure 4.2 Advanced Hardware Architecture

4.2.2.1 Function

The SBN provides GOES satellite images and national guidance products to all AWIPS sites. The data are transmitted over two separate channels and are received at each site by a communications processor (CP) that handles the communications protocols and forwards the data to the database computer.

4.2.2.2 Description

The CP is a Hewlett Packard 743 computer executing the RT operating system. Inside the CP is a communications board manufactured by SBE that handles the incoming SBN protocol. Each CP performs communications functions for the NESDIS (GOES satellite images) and NWS/TG (national guidance products) channels. Data are transmitted from the CP to the WFO-Advanced database over the ethernet. The communications processor and software were provided by the AWIPS contractor. Figure 4.3 provides a detailed diagram of the SBN communications hardware and links.

SBN Data Acquisition

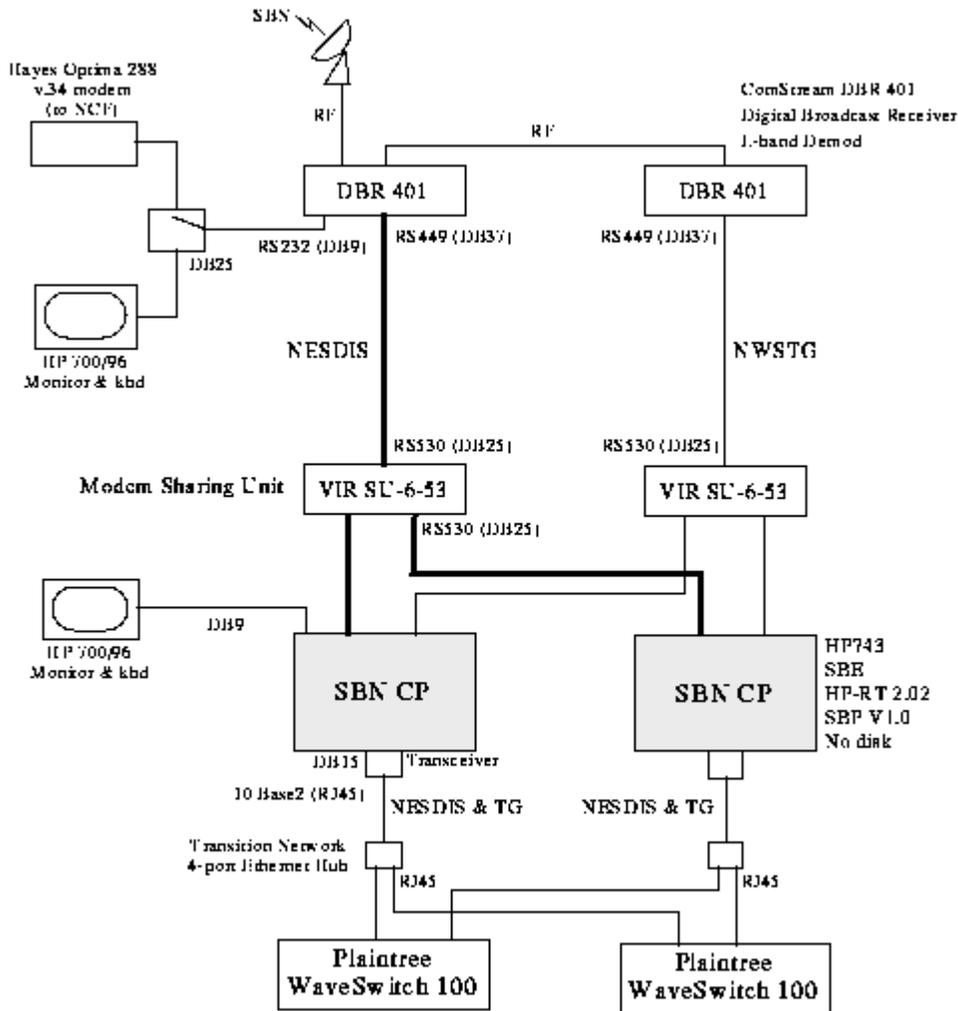


Figure 4.3 SBN data acquisition schematic

4.2.2.3 Backup

Two CP processors, each with its own connection to the incoming signal, provide the desired redundancy. Each processor routinely sends data to its assigned database. If a CP fails, data from the operating CP needs to be routed to the second database in order for both databases to have current data.

4.2.3 WSR-88D Radar

4.2.3.1 Function

The WSR-88D radar product generator is the source of radar data for the weather forecast office. The data are received and processed for storage and display by the WFO-Advanced system.

4.2.3.2 Description

The Simpact Freeway performs the X.25 communications protocol conversion. It is a commercial system that can receive data through several ports concurrently using a choice of protocols. In Denver, the Freeway handles the dedicated line to the primary WSR-88D radar and also dial-up lines to associated radars. Radar images are received from the radar product generator (RPG) in compressed form and sent over the ethernet to the WFO-Advanced database. Figure 4.4 provides the detailed configuration of the radar acquisition components.

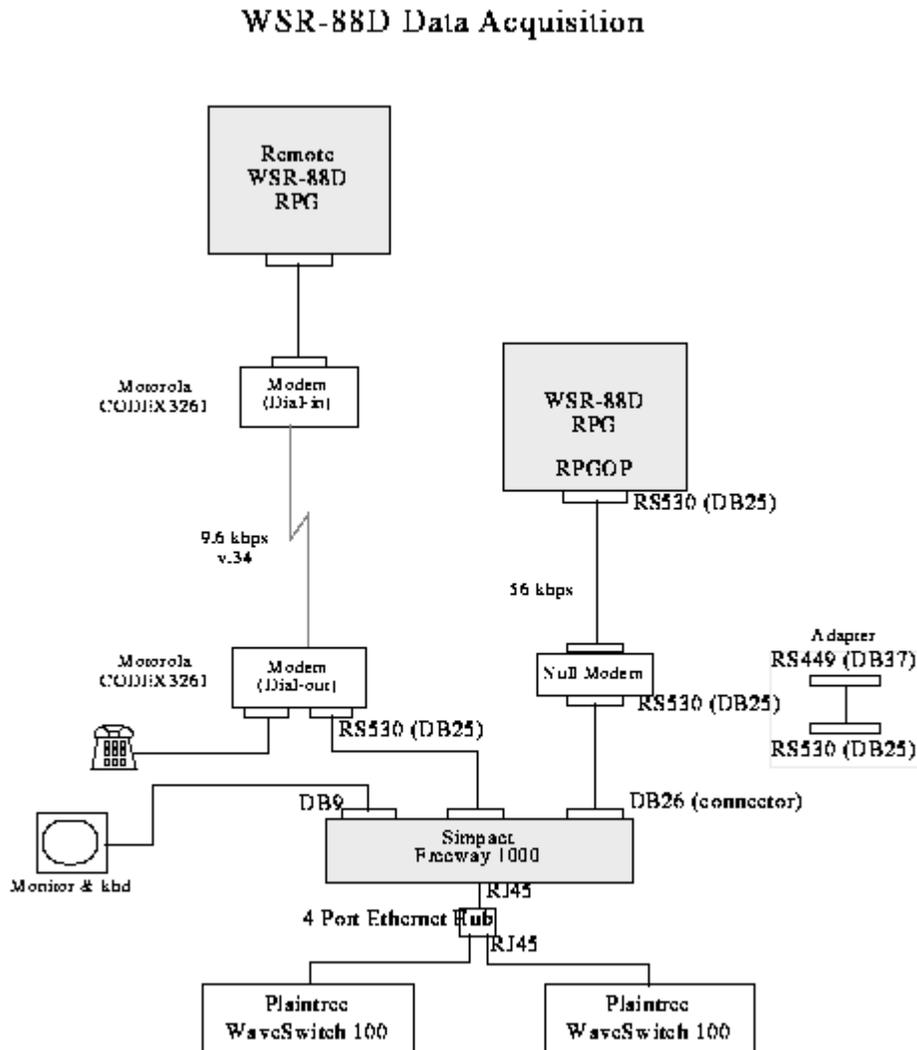


Figure 4.4 WSR 88D data acquisition schematic

4.2.3.3 Backup

A second Simpact Freeway is used to receive the input signal if the primary system fails. A manual switch is used to connect the radar signal to the back-up Freeway.

4.2.4 Other Processors

4.2.4.1 Function

WFO-Advanced has two other external links for receipt and transmission of data. Experimental observations and products are received from FSL over a T1 line, and forecast products prepared on the workstation are sent to other locations through the AFOS system. A Local Data Acquisition and Distribution (LDAD) system is scheduled to be installed in the near future. It will receive additional local data sets and distribute weather products to external users.

4.2.4.2 Description

An RS232 line connects the AFOS system to the WFO-Advanced database computer. The line can be switched manually to connect to either database computer. In addition, the output of local forecast models and other data collected at FSL is transmitted over a T1 link through a CISCO 4500 communications router to the WFO-Advanced ethernet. Routine transmissions from FSL through this link use the Local Data Manager (LDM) protocol developed by Unidata. The proposed LDAD hardware consists principally of two Pentium PC servers, two network servers, and several modems. In order to provide the required network security, the LDAD system and FSL's T1 link are planned to be isolated from the WFO-Advanced system through a TIS communications firewall.

4.2.4.3 Backup

Since FSL's data are considered experimental, no backup is provided for the link with FSL. The AFOS connection is critical for dissemination of forecast products, and the line has a switch that can connect AFOS to either one of the database machines. For the LDAD system, redundant processors and network servers will provide the desired backup capability.

4.3 Database and Application Computers

4.3.1 Function

The database computer performs all of the data management and some data acquisition tasks. In addition, each database computer runs the X server software for half of the text workstations. The purpose of the application processor is to execute meteorological applications such as event monitors, routine product preparation, data analyses, and local forecast models.

4.3.2 Description

Each database computer is a Hewlett Packard K200 server equipped with two PA7200 processors, 256 megabytes of random access memory, and 8 gigabytes of disk storage. Hydrometeorological products and data are allocated 6 GB. The application processor is identical except that it has only 128 megabytes of memory and 4 gigabytes of disk storage. These computers connect to the FDDI network. The Informix relational database management software is used to store a limited set of hydrometeorological data in the database.

4.3.3 Backup

To provide the desired redundancy, the system is configured with two database computers. Each stores a complete set of all data on its local disks. Half of the workstations (including text processors) use the database on one computer and the remainder use the database on the other. Should one computer fail, half of the workstations would temporarily be without data. A UNIX shell script needs to be executed to reconfigure the system so that the affected workstations can access the other database. An alternate configuration that used data mirroring and dual-porting of disks resulted in occasional catastrophic corruption of data on the primary and back-up disks. This resulted in neither database computer being able to access the database.

4.4 Workstation

4.4.1 Function

The workstation provides the capability for the user to display graphical information and interact with the data. Each workstation runs two independent copies of the menu and display software, one for each monitor. Each workstation includes an associated X terminal for display and editing of text products.

4.4.2 Description

The workstation consists of a Hewlett Packard J200 computer with two 19 inch color monitors. Each monitor is controlled by an HCRX24 graphics controller that supports four concurrent color tables: 24-bit true color, 8-bit image underlay, 8-bit graphics overlay, and 8-bit transparency. To support long animation loops, the workstation requires a minimum of 256 megabytes of random access memory. Each workstation is equipped with a four gigabyte disk drive for storing the operating system and workstation software. The user input devices are two sets of keyboards, mice, and touchpads. The workstation can easily be configured to operate with either a single set of input devices for both monitors or separate input devices for each monitor. The J200 connects directly to the FDDI network.

The text editor runs on a Hewlett Packard black-and-white X terminal with 12 megabytes of memory. The X terminal has an ethernet connection.

4.4.3 Backup

The two color graphics monitors on the workstation operate independently; each has its own input devices. If one display screen fails, the other screen can perform all of the workstation functions. In the event of a complete workstation or text processor failure, another workstation or text processor in the system would need to assume its functions.

4.5 Concluding Remarks

The WFO-Advanced hardware architecture is being evaluated at the NWS Forecast Office in Denver, Colorado.

As of this writing, several of the backup systems described in this section are not in place. The backup CP is currently undergoing testing in Boulder, and is expected to be installed in Denver before the end of January. At that time, the redundant data feeds and databases described earlier will be implemented, and the workstations will be split between the two databases, with two workstations and three text stations on each server.

It is likely that as the evaluation of the system proceeds, minor changes to the architecture may be made.

CHAPTER 5 Software Overview

At the highest level, the WFO-Advanced system can be decomposed into a number of major software subsystems: Data Acquisition, Data Management, Workstation Display, and Text. Each of these subsystems is described in more detail in one or more chapters of this document. The intent of this chapter is to give the big-picture overview of the system, and also to describe some services available to each of these subsystems, such as interprocess communications and the foundation class library. Please note that this is a rather cursory overview.

5.1 WFO-Advanced Subsystems

The following are the major subsystems of WFO-Advanced discussed in this document.

- The Data Acquisition subsystem is responsible for receipt and storage of all meteorological data sets, including any communications interfaces or required decoding. Data Acquisition uses services provided by Data Management for storage and notification.
- Data Management is responsible for storage, retrieval, and maintenance of meteorological data sets, including purging and product notification. Data Management also provides access to static tables of system configuration and other information. Since data is the lifeblood of WFO-Advanced, all other subsystems make use of services provided by Data Management.
- The Workstation Display subsystem provides graphical display of meteorological data, including the user interface for data selection, examination, and manipulation. Workstation Display relies on Data Management for meteorological data access and notification, and static configuration tables. Major components of Workstation Display are the User Interface, the Display, and Depictables. Each of these has a separate chapter in this document.
- The Text subsystem provides for display, editing, and dissemination of textual meteorological products. In this document, text data acquisition and data management are considered part of the Text subsystem, as well.

5.2 Interprocess Communications

The WFO-Advanced project has developed a software framework to handle interprocess communications (IPC). The currently implemented mechanism is based on TCP sockets, but the design will accommodate the addition of alternative communication mechanisms. IPC functionality depends upon inheriting from or using the following classes in the ipc source directory.

- **ArgPkg:** The ArgPkg class is the base class for a set of template classes that provides for marshalling of arguments into a sendable byte stream, and back.
- **Connection:** The Connection class represents the mechanism used to deliver a message from one place to another. Currently, SocketConnection inherits from Connection, and is the only implemented mechanism.
- **Dispatcher:** The Dispatcher isn't abstract, or generally overridden, but rather implements the dispatching of messages received by a particular process to the appropriate Receiver.
- **IPC_Target:** The IPC_Target represents the target process of a message.
- **Message:** This is the Message itself the data moving from one process to another. New kinds of messages are implemented by inheriting from Message.
- **ParameterizedMessage:** Messages that use the ArgPkg argument marshalling mechanism inherit from ParameterizedMessage, which in turn inherits from Message.
- **Receiver:** A Receiver represents a target within a particular process that should receive a particular type of message. When new modules need to receive new types of messages, a new class is created as a descendent of Receiver. The Dispatcher is responsible for routing a message to the appropriate Receiver.

Note: The original DMQ-based IPC software was replaced in the AWIPS Build 3 handoff system (WFO-A version fxa-3.0, completed 2/97) with a socket-based system. See the design document for more information.

5.3 Foundation Classes

A library of low-level utility classes has been assembled in our foundation directory, including a heavily-used set of container classes, and a number of common data classes.

5.3.1 Container Classes

WFO-Advanced makes heavy use of a set of container classes, implemented as template classes. The dictionary and sequence classes are the most used of these.

- **The Dictionary Classes.** These templates implement associative arrays parameterized by both key and value. Specialized variants exist to reduce template code bloat for combinations where the key or value or both are pointers.
- **The Sequence Classes.** These templates implement ordered sequences of arbitrary parameterized classes. A specialized variant exists to reduce template code bloat for sequences of pointers.

5.3.2 Common Data Classes

- **AbsTime.** This class, built on UNIX time concepts, implements the concept of a particular date and time. It provides an easier and less error-prone interface to use than the UNIX time library calls.

- `TextString`. This is a powerful and easy-to-use class for text string manipulation, much less error-prone than using C strings.

5.3.3 Event Logging

WFO-Advanced has a standard C++ interface, the `LogStream` class, for writing messages to log files. To the user, the log request looks just like a standard `iostream` write. Multiple message types are available, and the set of messages logged can be tailored at run time.

5.4 Off-the-shelf Software Packages

Both commercial off-the-shelf (COTS) and freely-available software packages are or have been used in WFO-Advanced. (Those in *italics* are no longer in use.)

5.4.1 *OI*

OI (Object Interface) is a commercial C++ X11 user interface widget software library. The Build 3 user interface library module (`uiLib`) was built on top of OI. In late summer 1997, we completed converting the entire UI to Tcl/Tk. This eliminates the `uiLib`, as well.

5.4.2 Tcl/Tk

Tcl/Tk (Tool Command Language/Tool Kit) is a freely-available scripting language (Tcl) and a User Interface tool kit (Tk). Tcl/Tk provides a very high-level user interface development language, and a powerful set of UI widgets, and supports rapid development of flexible user interfaces.

5.4.3 *DMQ*

DMQ (DEC Message Queue) is a commercial message-based communications product. It is a descendent of PAMS, a DEC VMS messaging product used successfully in the DARE workstation project. While the product has a number of virtues, its proprietary black-box nature made it difficult at times to diagnose interprocess communications bottlenecks and other problems. During 1997, we rewrote our interprocess communication (IPC) routines to replace DMQ with TCP sockets. The IPC software is fully described in CHAPTER 5a.

5.4.4 *Informix*

Informix is a commercial relational database product, used in WFO-Advanced for the text and hydrology databases.

5.4.5 *NetCDF*

NetCDF is a freely available data storage and retrieval software interface and file structure. NetCDF provides portable, self-describing data files, particularly well-suited to uniform multi-dimensional data sets such as gridded data and satellite images. WFO-Advanced makes extensive use of netCDF for gridded data, satellite images, and some point data sets.

CHAPTER 5a Interprocess Communication (IPC)

A running program -- a process -- may well be functional by itself. But it's when processes communicate with each other that we take full advantage of the power of today's computing systems. As a Sun Microsystems president once said, "The computer is the network." Modern computing applications fetch data from remote sources, distribute tasks between clients and servers, and present results to users sitting in front of networked, remote displays.

The D2D component of the WFO-Advanced system is a "modern computing application" in the foregoing regard. Our software reads data from remote sources (the data ingest processes), distributes tasks between clients and servers (the communications router, the notification server, or the IGC_Process, for example), and presents results to users on workstation and X terminal displays.

Interprocess communication (IPC) of this sort is such a fundamental part of the WFO-Advanced D2D software that we have developed an object-oriented library to make the sending and receiving of data relatively simple. In fact, 60% of the applications in the D2D software link with our IPC library.

The IPC library developed for the D2D has three major components:

- *Message Encapsulation:* Software that encodes and decodes various data types (including objects) into a binary byte stream. This byte stream is combined with message attributes (priority, sync/async, etc.) to create an abstract message object that all transport mechanisms can send and receive.
- *Addressing Mechanism:* Software that encapsulates the address of a sending and receiving process.
- *Transport Mechanism:* Software that does the actual transmitting (sending and receiving) of binary message data.

Recognizing that various IPC transport mechanisms may exist simultaneously, and that some are better suited for certain communications tasks than others, the D2D IPC library includes support for multiple, concurrent IPC mechanisms. The library hides the underlying IPC transport from an application programmer and automatically chooses the best one for each message to be sent.

Currently, the software's decision of which mechanism to use is made quite easy; our IPC library has never had multiple transport mechanisms. Since the birth of D2D, we have used three different transport mechanisms. Each successive mechanism has satisfied the requirements better than the previous mechanism so we haven't found a need to support two or more mechanisms concurrently. This may change as more reliable and high performing mechanisms emerge.

The rest of this chapter:

- Describes the major requirements of IPC which have driven the design of all our implementations;
- Provides an historical evolution of our IPC library, focusing on the strengths and weakness of the three different transport mechanisms that have been used;
- Provides a software overview and implementation details for our current mechanism, Thread Based Socket IPC;
- Summarizes this chapter by exploring the issues that should be considered for future IPC development.

5a.1 IPC Requirements

This section describes the major requirements for IPC that surfaced in earlier incarnations and prototypes of D2D. Our first two implementations did not sufficiently satisfy all of these requirements, and this was the main impetus for evolving to our current implementation.

5a.1.1 Asynchronous Sends (Fire & Forget)

Clients of the IPC library should not be involved in the intricacies of queueing and resending data if the receiving process is not ready to accept and process messages. The sending process may have some critical processing to perform and cannot wait for the message to be received. The transport mechanism must hold on to messages until they can be sent without blocking the execution of the sending process. The mechanism should also be able to detect when a receiving process is not responding because of a normal termination, a crash, or a hang (infinite loop, for example). In that case, the transport mechanism should abort the sending of all the queued messages.

This requirement is crucial. In fact, the D2D data ingest system relies on this feature -- 14 of the 24 processes that comprise the data ingest system will not function without it.

5a.1.2 Synchronous Sends

In some rare cases, a client may not want to resume processing until it is certain that the IPC message has arrived at its destination and is processed. The transport mechanism should not queue any additional messages until the initial message arrives.

5a.1.3 Message Priority

With our IPC library, a client can specify two levels of priority for a message: high and normal. If a receiving process is responsive, then priority is not much of an issue because a message will be received almost immediately after it is sent if a fast transport mechanism is used. However, if the process is not responsive, high priority messages should be queued, sent, and received before messages with normal priority.

Currently, the vast majority of our IPC messages are specified to be at normal priority. In fact, the only type of message that is currently of a high priority is for relaying radar alert data.

5a.1.4 Message Selection

A client may want to select which messages to receive and process, and defer the processing of messages that may arrive before a selected message. Our IPC library allows a client to select messages from a particular sending process.

For example, a text depictable in an IGC_Process may need data from a TextDB process running on some remote host. The IGC sends an IPC message to the TextDB requesting the data. The IGC does not want to do any more processing until the reply from the TextDB arrives, so it chooses to receive messages only from the TextDB process.

5a.1.5 Friendly to Event Multiplexors

Event multiplexors wait for input, output, or exceptional conditions on a number of data sources at the same time. These are essential to efficient operation of D2D software, particularly the display software which needs to multiplex I/O with the X window display, the keypad, external applications, timers, and the IPC library.

In a UNIX environment, most event multiplexors are implemented with a `select()` call which uses file descriptors to identify the devices to multiplex. Therefore, the IPC transport mechanism must provide file descriptors for the devices that receive IPC messages.

The alternative to using an event multiplexor is to poll each of the input sources separately in a loop. This wastes CPU time. Considering that a single D2D graphics workstation runs 24 (or more as users start extensions and applications) processes -- each of which has to poll for input -- the waste is astronomical.

5a.1.6 Unlimited Message Size

The client should never have to worry about whether an IPC message is too big for a transport device. If it is, the transport mechanism should fragment the message and then reassemble the fragments on the receiving end.

The number of bytes for most IPC messages passed between D2D processes is rather small. However, some extensions such as `warnGen` and the `alertAreaEditor` can potentially pass large messages (> 64k) to the IGC process.

5a.1.7 Reliable and Maintainable

The IPC mechanism should always be accessible and require no special permissions to use except to provide security or authentication. It should take as few process slots as possible to run, preferably none. The only reasons a message cannot be sent may be that the receiving process is down or hung, the operating system is down, the machine on which it runs is down, or the network is down.

If one or more auxiliary processes are used to implement the IPC mechanism, then these processes should be able to restart without having to restart the client processes. In other words, these processes should be able to save and restore their state. The IPC mechanism should be able to detect an unresponsive IPC server process, and restart that process if necessary.

As far as being maintainable, the mechanism should not be a black box. It should either have a well-documented theory of operation or provide source code to its implementation.

The mechanism should require little system administrator assistance. It should use familiar forms of addressing.

5a.1.8 Performance

Initially, the needs of the IPC library were modest; today, we are sending messages all the time. In just the Denver WFO, literally hundreds of processes are involved, passing messages with a variety of sizes to each other. Here are some examples where fast IPC performance is of vital importance.

Users expect that green times displayed on the user interface product menus or the display of a product on an IGC will be updated as soon as new data for that product arrives. The expediency of auto-notification depends heavily on IPC. The decoders notify the notification server, which in turn notifies the IGCs, the UI, and the volume browser. The notification server sends and receives thousands of messages per hour; some of these messages can be quite large.

Remember, the UI and the display of data is divided into separate processes, although the user may not be aware of this. If the user selects a product for display, or a pane to swap with the large pane, he/she expects the results almost immediately. An average user response time of more than two seconds would register serious complaints among our users. Since the IGC may spend a good portion of the two seconds reading and displaying data, rapidly sending the message from the UI to the IGC and back to the UI is crucial.

Like the UI and IGC, extension interactivity is partitioned between the IGC and an extension process. One of the most time-critical tasks for a meteorologist is the generation of warnings, which is managed by the warnGen extension. Potentially very large IPC messages are sent between the IGC and warnGen processes. If the messages take too long to send and receive, the user will be bogged down with sluggish responsiveness.

While it would be nice to state this requirement as "throughput of at least 30 kilobytes per second," or some other figure, this requirement depends on the user's perceptions of system responsiveness, and the speed of non-IPC processing.

5a.2 History of the IPC Library

Since the advent of D2D, the transport and addressing mechanisms of the IPC library have evolved quite dramatically during our short development history. However, the message encapsulation component of the library has not changed nearly as much. This section will focus on the three transport mechanisms and the reasons for switching mechanisms. Addressing mechanisms are dependent on the transport mechanism, and have been changed only to accommodate a new transport mechanism.

Granted, approaching UNIX IPC can be a bit difficult because of all the different mechanisms it provides. Since UNIX had a colorful evolution, the IPC features it provides are equally colorful. There are a number of IPC mechanisms directly provided by the UNIX operating system, each with its own features and blemishes. There are also a number of packages, commercial and free, that layer themselves atop the UNIX IPC mechanisms. These "middleware" packages make using UNIX IPC easier or provide additional features not directly provided by UNIX itself. Since the beginning of our development, we have used three distinct mechanism implementations: one utilizes a middleware package, while the other two have been built on top of native UNIX mechanisms.

5a.2.1 DEC Message Queue (DMQ)

The first IPC implementation was built utilizing a commercial middleware package called MessageQ, DECmessageQ, or DMQ, made by Digital Equipment Corporation (DEC).

5a.2.1.1 Design Overview

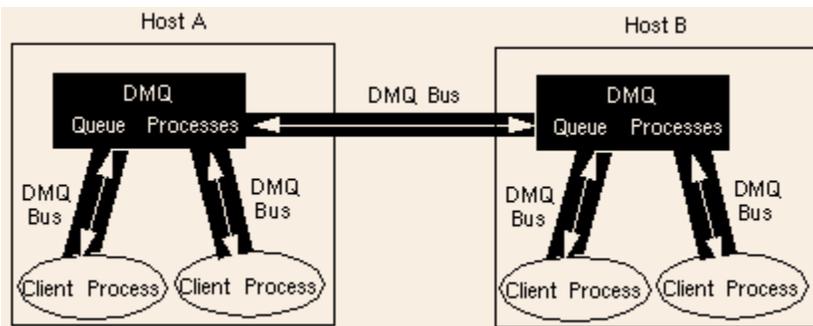


Figure 5a.1 DMQ Transport Design

To send a message, a client process makes a call to the DMQ application programmers interface (API), specifying the address, the message attributes, and the message data. If the send is synchronous, the API routine will return when the message has been dequeued by the destination process. If the send is asynchronous, the routine will return immediately. The message is placed on the DMQ bus and sent to one of the DMQ queuing processes. The queuing processes examine the address and determine whether the message is for a process on the local host or some remote host. If it is for a local process, it is added to the back of the queue dedicated to the destination process. For a remote process, the message is sent via the

bus to a queuing process on the remote host. That queuing process then inserts the message into the proper queue. The destination process does not have to be running.

To receive a message, a client process makes a call to the DMQ API, passing a time-out value, in deciseconds. The routine will not return until a message has been received or the time-out has been exceeded. Optionally, the client can pass an address, which tells the routine to return a message only from a process with that address. The DMQ routine makes a request via the bus to a queuing process for a message. If there is a message on the queue for the receiving process, it is sent back via the bus.

5a.2.1.2 Strengths

Because a queue is maintained in another process, a client process can send to a target and not wait for the destination process to receive the message. The DMQ queuing process is even smart enough to write the message to disk if the destination process is not running, and then restore and send the message when the process does come back up. Optionally, a sending process can block its execution until the message has arrived at the receiving process. Therefore, this transport mechanism successfully satisfies the requirement of both synchronous and asynchronous sends.

DMQ supports queuing processes with multiple queues, which makes it possible to separately receive high- and normal-priority messages, satisfying another important IPC requirement.

Receiving specific messages distinguished by address is another strong feature of the DMQ transport mechanism. The client process can ask the DMQ queuing process for a message with a particular address, leaving all the other messages still queued.

5a.2.1.3 Weaknesses

In Figure 5a.1, main components of the DMQ architecture are depicted as black boxes since DEC has supplied minimal information about the internals of these pieces. For example, the DMQ bus might be a TCP socket, a UNIX Domain socket, or something entirely different. Likewise, all we know about the queuing processes is that there are at least three running simultaneously, and sometimes up to six processes. Little is known about what these processes do, or the internal details of the queues; this makes this transport mechanism extremely difficult for D2D developers to maintain.

Another major weakness is that the DMQ transport is definitely not event multiplexor friendly. A UNIX `select()` call cannot be used to determine when data has arrived on the DMQ bus, since DEC does not supply a file descriptor for the bus. This has tremendous performance ramifications on the D2D display processes, which receive events from multiple sources: IPC, `stdin` and `stdout` (application interface), X, timer events, etc. These display processes are forced to resort to polling, which consumes a lot of unnecessary CPU time.

Unfortunately, the DMQ bus has a limitation of 32 kilobytes. The software we wrote around DMQ does not support automatically fragmenting and reassembling messages, so the

responsibility falls to the users of the IPC library. This is not desirable since IPC clients have enough to worry about.

Besides requiring that system administrators install and maintain the product, DMQ layers an additional level of addressing on top of that already provided by every UNIX workstation. This makes debugging of IPC problems more difficult because of the additional lookup or memorization needed to map from DMQ host number to host name or Internet Protocol (IP) address. Users who initiate processes using IPC are also burdened with setting an environment variable to the DMQ host number for the host of the process.

Using separate processes to queue messages allows this transport mechanism to satisfy our important requirement of asynchronous sends. However, this strategy adversely affects performance and reliability. The dynamic memory for the binary byte stream has to be allocated in the sending process, receiving process, and the intermediate queuing process. The extra context switch to the queuing process increases transmission time, especially for large messages. For reliability reasons, a queuing process is not optimal since it can crash. Even worse, DMQ queueing processes do not save their state upon a crash, which means that all client processes will need to be restarted. This is problematic since the workstation takes considerable time to initialize.

Finally, this middleware product costs money. But UNIX provides several IPC mechanisms, and we already pay for UNIX. Why shell out more money?

5a.2.2 Server Based Socket IPC

Our next IPC mechanism, delivered to the fxa-3.0 (AWIPS Build 3) release of D2D, was designed to address many of the problems discussed in Section 5a.2.1.3. It was our first attempt at using the native UNIX IPC mechanisms, such as UNIX Domain (local) sockets, and RPC.

5a.2.2.1 Design Overview

In the UNIX parlance, sockets are an endpoint of communication. After a process creates a socket, it can then connect its socket to another process's socket and exchange data. The other process may run on the same host or across the world on the Internet, depending on the type of socket used. Sockets are easy to use; the same read and write calls that work with files also work with sockets. Because they're ubiquitous, well-documented, and relatively easy to use, sockets are an attractive IPC option.

UNIX Domain sockets, also known as local sockets, enable processes on a single UNIX system to communicate. These kinds of sockets are identified by creating nodes in the file system, usually under the /tmp directory. The X Window System uses UNIX Domain sockets: programs running on the same host as the X server can connect to the X server on display 0 by using the socket /var/spool/sockets/X11/0. This kind of socket features an easy addressing scheme (nodes in the filesystem) but can transfer data between processes on a single system *only*.

Sun's Remote Procedure Call system, or RPC, is an interface layered above sockets and is free with HP/UX. The client sends a message by making what appears to be a procedure call. The

parameters passed into the procedure are the message data. The return value of the procedure is an optional reply message from the destination process.

In order to implement asynchronous sends (fire and forget), an IPC server daemon runs on every host where our client processes need IPC. The server maintains a queue for every running client process on the server's host, which is indexed by the process address (host IP address and process ID). Each queue maintains a local socket that is connected to a client process. RPC calls are used for communication from a client process to the server. Local sockets are used for sending queued messages from the server to the client. This design is very similar to how DMQ supports synchronous sends except for some notable improvements:

- Only one server daemon is needed per host. DMQ has at least three (sometimes six) daemons running on a single host.
- The server can restore its queues (both messages and addresses) if it crashes or is restarted. DMQ queuing processes do not have this feature.

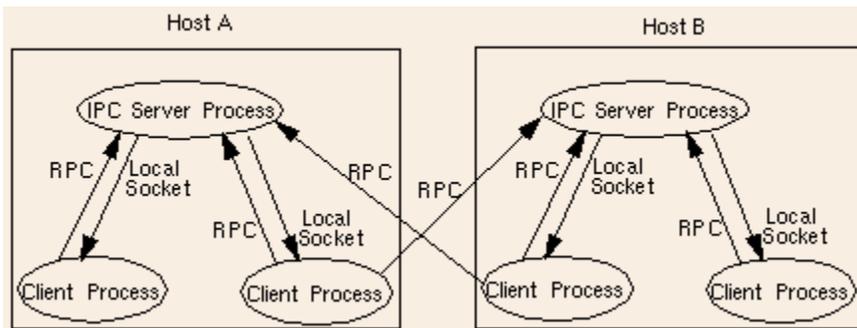


Figure 5a.2 Server Based IPC Transport Design

Here is the basic algorithm for this transport mechanism. Figure 5a.2 illustrates this algorithm.

1. The IPC server (process name: `rpc.ipcd`) is invoked and initialized. Its initialization includes registering with the RPC system, and restoring its queues from the crash recovery file, if there is a file. Stored for each queue is the IPC address of the queue's process, the pathname of the process's socket, and any messages that still need to be sent. When the queue object is recreated, the server creates a socket and tries to connect to the process's socket. If this fails, the queue is removed. This scenario is very probable, especially if an IPC server daemon is down for a fairly long time.
2. After initialization, the IPC server enters its event multiplexor. The UNIX `select()` call is used to determine when the RPC sockets are ready for reading, and when the local sockets for each queue are ready for writing.
3. An IPC client process (an executable linking with the IPC library) initializes by creating a local (UNIX Domain) socket, and then binds that socket with some file whose name includes the process ID. It then registers with the server via RPC, passing its IPC address (host IP address, and process ID), and the pathname of the local socket. The RPC will return a status indicating

whether the server was able to create a queue for this process and connect to the local socket. If the client cannot register with the server, then this client process will not be able to perform any IPC during the life of the process.

4. When an IPC client wants to send a message, a message object is created with the binary stream and some message attributes. An RPC message is then sent to the IPC server that is running on the host of the destination process.
5. An IPC server receives the RPC message sent in step 4. It then tries to queue the message using the IPC address to identify the queue object to use. If it can't find the queue, or the queue is full, then a status is delivered to the sender through the return value of the RPC. Each queue object has two queues, one for each possible priority. The priority attribute of the message is used to decide which queue to use.
6. If the queueing of the message is successful, the server will be notified via the `select()` call that there is now a message to send. The selected message will be from the head of the high priority queue. If that queue is empty, then the message will be from the normal priority queue. Using the UNIX `write()` call, the message's binary stream is placed on the local socket for the destination process.
7. The event multiplexor for the destination process will detect that the local socket connected to the server is now ready for reading. It will read the binary data from the socket using the UNIX `read()` call, and then pass the message to the dispatcher. If the client is waiting for a message from a particular target, the dispatcher will queue the message until the desired message arrives. Once a message is ready to be dispatched, it is sent to the appropriate receiver object.
8. When a client process exits normally or abnormally, an RPC message is sent to the IPC server running on the same host as the client. This message tells the server to delete the queue for that client, and any pending messages in the queue.
9. When the IPC server exits normally or abnormally, it closes the local socket for every queue. It then saves the state of every queue into a crash recovery file.

5a.2.2.2 Strengths

Because a queue is maintained in another process, a client process can send to a target and not wait for the destination process to receive the message. The only caveat is that the destination process has to be running, which is a minor blemish compared to DMQ. This transport mechanism successfully satisfies the requirement of asynchronous (fire and forget) sends. It is possible that the queueing process could be busy doing something else and not be responsive to the sending client, but that is unlikely since the server is not processing messages -- it is just forwarding the messages to other clients.

Local sockets and RPC are friendly to event multiplexors since a file (socket) descriptor can easily be obtained. And because this mechanism is event multiplexor friendly, a display process can truly sleep while waiting for an IPC, X, or a timer event arrive. No CPU intensive polling is necessary.

Both UNIX domain sockets and RPC support a message size that is limited only by the amount of virtual memory to which a process has access. Although this mechanism has code to fragment and reassemble messages, it is rarely used.

The addressing for this mechanism is fairly universal. Every UNIX process has a process ID, and every UNIX host has an IP (Internet Protocol) address. This is a big improvement over DMQ.

The IPC server has separate queues for IPC messages with different priorities, satisfying an important IPC requirement.

5a.2.2.3 Weaknesses

As shown in Figure 5a.2, local sockets are used as only a one-way communication link from the server to the client. However, local sockets can support two-way communication. Thus, this transport design is using only half of the available bandwidth of the socket.

The requirement of synchronous sends is not completely satisfied with this mechanism. When the sending process returns from its synchronous send, it knows that the message has been received at the server process running on the host of the target process. However, the message has not arrived at the destination process yet; the destination process might be in the middle of some intensive processing and cannot read the local socket immediately.

Although not as mysterious as DMQ, RPC is somewhat of a black box. Unfortunately, it is difficult to tell how many internal sockets are used for its transport. That information is pertinent to the server daemon since UNIX has a limit on the number of open sockets and files a process can have. In addition, HP's implementation of RPC was flawed with HP-UX 10.10: occasionally, strange values were returned from RPC calls. The problem seems to be have fixed with HP-UX 10.20.

Using separate processes to queue messages allows this transport mechanism to satisfy our important requirement of asynchronous sends. However this strategy adversely affects performance and reliability. The dynamic memory for the binary byte stream has to be allocated in the sending process, receiving process, and the intermediate queuing process. The extra context switch to the queuing process increases transmission time, especially for large messages. For reliability reasons, a queuing process is not optimal since it can crash. Fortunately, unlike DMQ, our server can restore its state after a crash, eliminating the need for clients to be restarted.

This design is fairly complex, and so is the resulting code which deplorably affects maintainability. The code to generate RPC calls is particularly complex. Initially, we planned to use the utility `rpcgen` to generate the RPC code. Unfortunately, it produced code of dubious quality.

The message selection requirement is not completely satisfied with this mechanism. The local socket connected to the server may contain many messages from different target processes. There is not a way to read from the middle of a socket and leave the rest of the socket data intact. Thus, other non-desirable messages have to be read before the desired message can be read. This implementation defers the dispatching and processing of the non-desired message until the desired message arrives. This could be time consuming and memory exhaustive if the desired message takes a long time to arrive and a lot of unwanted messages arrive in the meantime. DMQ and Thread Based Socket IPC provide better solutions for satisfying selective reception of messages.

5a.2.3 Thread Based Socket IPC

Our latest and greatest IPC mechanism, delivered to the november97 (AWIPS Build 4) release of D2D, was designed to address the performance and reliability problems discussed in Section 5a.2.2.3. This time we opted for a different UNIX IPC mechanism, TCP sockets. Also, in order to support asynchronous sends, we needed a library that supplied multiple threads of execution. Our choice was HP's implementation of the DCE pthread library (not that we had much choice).

5a.2.3.1 Design Overview

Transport-control protocol, or TCP, sockets are another type of network-capable socket. TCP sockets establish a virtual connection between peer processes wishing to communicate. Most of the popular Internet services are implemented using TCP sockets: rlogin, telnet, http, ftp, X Windows, mail, print spooling, net news, kerberos, and more. TCP works by layering a reliable, flow-controlled, data stream protocol on top of the Internet Protocol (IP). TCP sockets are quite easy to use because they are interchangeable with the file I/O interface used to read and write data to disks and terminals. TCP sockets enable processes to communicate whether they're on a single host, on the local network, or on the Internet; the programmer doesn't have to change the program to support all these types of connections.

Instead of a queuing process, we opted for a multiple thread environment in order to support asynchronous sends. Maurice Bach in *The Design of the UNIX Operating System* defines a thread as the following:

An independent flow of control within a process, composed of a context (which includes a register set and a program counter) and a sequence of instructions to execute.

Until now, most if not all applications developed here at FSL have used single-threaded processes, which have a single flow of control through program code. Processes linking with our IPC library will potentially be multi-threaded with multiple flows of control. A multi-threaded process can achieve significant performance gains through the use of concurrent thread execution. This means the two or more threads are in progress at the same time. For some HP hosts with multiple processors, such as the K series, two or more threads can be executed simultaneously.

This mechanism uses multiple threads of execution to implement asynchronous (fire and forget) sends. Here is a possible scenario without using multiple threads. Suppose the notification server process is trying to send a message to an IGC process, but the IGC is really busy doing some other processing, perhaps constructing some radar tables. The UNIX buffer between the two sockets has become full, so when the notification server tries to write to the socket, the UNIX write() call blocks execution until the IGC reads some bytes from its end of the buffer. Since execution is blocked, the notification server cannot process other notifications, which will delay the notifications sent to other workstations who have IGCs that are responsive.

Now, consider the above scenario in a multi-threaded process. When the notification server detects the socket to the IGC is full, it creates a new thread of execution whose job is to keep writing to the socket until the entire message is sent. This thread will be blocked inside its write() call. Meanwhile, the main thread can proceed with the business of the notification server. When the IGC removes some bytes from the socket buffer, the socket writing thread wakes up, and receives the CPU so it can write to its socket. If the notification server keeps trying to send messages to the busy IGC, more socket writing threads may be created. Thus, the process could have a main thread and many socket writing threads. Keep in mind that thread creation does carry some performance overhead, so we want to initiate threads only when the socket buffer is full.

The real strength of this design is the elimination of the queueing (server) process. The socket writing threads (threads are often referred to as lightweight processes) take the place of the queueing process and are used only when really needed. The result is a dramatic performance increase over our previous two transport mechanisms. See Section 5a.2.5.

The following diagram depicts the general data flow for this mechanism. Each client process maintains a socket for every other process which which it wants to communicate. The bidirectional TCP socket can be used for both sending and receiving data on the same host or remote hosts. Unfortunately, the number of files (and/or sockets) a process can have open is limited (most of our hosts are configured to have 60, although this is a tunable kernel parameter at the expense of increased memory use). In an effort to be considerate of our clients' needs, the maximum number of file descriptors that the IPC library consumes is a third of the system max. In order to accomplish that goal, each socket object will record the time it was last accessed for either writing or reading. If a request for a new connection will exceed our 20 (or so) socket limit, the least recently used socket will be destroyed.

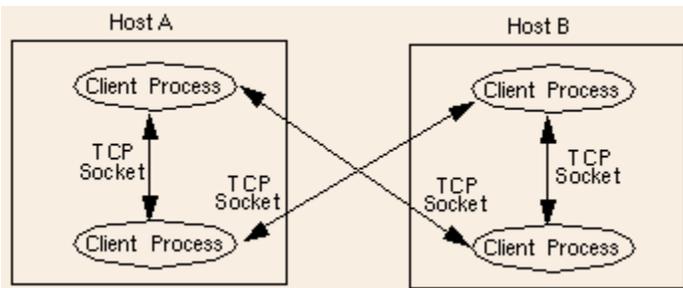


Figure 5a.3 Thread Based IPC Transport Design

5a.2.3.2 Strengths

TCP sockets are friendly to event multiplexors since a file (socket) descriptor can easily be obtained. And because this mechanism is event multiplexor friendly, a display process can truly sleep while waiting for an IPC, X, or a timer event arrive. No CPU-intensive polling is necessary.

As mentioned already, not having a third party server process involved will increase performance and reliability.

The use of threads fully satisfies the asynchronous send requirement. And the real advantage of using threads is that they are used only when really needed; only when the destination process is not responsive. Performing a send without a thread constitutes a synchronous send, which satisfies another important requirement.

The default buffer size for a TCP socket is 32K (32,768) bytes, but can be configured to the maximum size of 256K (262,144) bytes. This mechanism uses the maximum buffer size, but that is a tunable parameter, specified in our config file (ipc.config). The IPC library does multiple socket writes and reads in order to send messages of unlimited size, satisfying another requirement of our IPC library.

Our design allows for client processes to have two sockets between them, one for normal and one for high priority messages, although the extra socket will be created only as needed. Currently, very few clients send or receive high priority messages, although that could certainly change. Our event multiplexor has been enhanced so that it will flag high priority sockets as readable before normal priority sockets.

Since each target has its own socket, it is relatively easy to implement the requirement of selecting messages from a particular destination. The algorithm for doing this is explained in Section 5a.4.2.3.

Because we are using threads only when needed and eliminating an extra process context switch, the performance of this mechanism is quite impressive. In the words of Beavis and Butthead, "this system rocks, man!"

5a.2.3.3 Weaknesses

Like RPC, the HP DCE pthread library is somewhat of a black box, although not quite as mysterious as DMQ. We have encountered several HP bugs thus far, and there is probably more to uncover. One particularly amusing bug that HP has since fixed successfully was a math error when running an optimized multi-threaded executable on their latest and greatest(?) hardware. Another bug is **not** so amusing since HP refuses to fix it. The pthread library places wrappers around most system calls. The wrapper around the select() call does not work according to the specification in the man page. If select() returns due to an error, it is supposed to clear the bit masks which are returned to the caller, indicating which file descriptors are now ready. The pthread select() call does not do this, which causes some serious problems with the Tcl library which uses select(). Fortunately for us, the Tcl consortium agreed to apply a patch which has just been released in version 8 of Tcl/Tk.

We know that doesn't sound promising for a critical library of an operational forecast system, but we are pretty optimistic about HP's future plans for operating systems that support the multi-threaded programming model. HP just announced their plans to support kernel threads in

HP-UX 10.30 which are entities that are visible to the OS kernel, as opposed to user threads which exist in user space and execute user code. Kernel threads promise to make signal handling in a multi-threaded environment more robust. We shall see about that. As we free our system from commercial off-the-shelf software (COTS), it will become more portable and will have more flexibility in pursuing superior thread libraries such as the one offered by Sun Microsystems.

5a.2.4 Requirement Comparisons

The following table compares how well the three mechanisms satisfy our requirements.

- **YES** implies that the requirement has been successfully met.
- **YES-** implies that the requirement has been met with several caveats.
- **NO+** implies that the requirement has been met on some levels, but generally is not satisfactory.
- **NO** implies that the requirement has not been met.

Table 5a.1 IPC Requirement Comparisons

	DMQ	Server Socket	Thread Socket
Asynchronous Sends	Yes	Yes	Yes
Synchronous Sends	Yes	No+	Yes
Message Priority	Yes	Yes	Yes
Message Selection	Yes	No+	Yes
Friendly to Event Multiplexors	No	Yes	Yes
Unlimited Message Size	No	Yes	Yes
Reliable and Maintainable	No+	Yes-	Yes
Performs Well	No+	Yes-	Yes

5a.2.5 Performance Comparisons

As development aids, we have implemented two test programs which are ideal for evaluating performance. The sending program sends a series of text string messages to a specified receiving process. The receiving program receives that text string, and then sends a reply back to the sending process. A timer is started when a message is sent, and is stopped when a reply for that message is received.

For the performance tests, we used an asynchronous, normal priority message containing the string, "This is a performance test". Each test sent the message 200 times; the time displayed in the following table is the average of those 200 times. We performed the test five times: twice for each of the three transport mechanism; once for a local transmission, and once for a remote transmission (the sender and receiver running on different hosts within the development network). Unfortunately, we were not able to perform a remote test for DMQ due to address configuration problems. The same hosts were used for all the tests, and performed when the hosts were relatively idle.

Here are the results from those tests.

Table 5a.2 IPC Performance Comparisons

Test Description	Time it took in seconds to send a message and receive a reply from that message
------------------	---

DMQ (local)	0.0596364
DMQ (remote)	Not Available
Server (local)	0.0168488
Server (remote)	0.0160134
Thread (local)	0.0022547
Thread (remote)	0.0104642

5a.3 Thread Based Socket IPC Design

After several failed attempts at describing the Thread Based Socket IPC implementation, we decided to rely on the power of analogy for delivering the "big picture" view of this transport design followed by a description of the main objects and their interactions.

5a.3.1 Consider This

Suppose you are a software engineer working for the hippest 3D visualization animation software producer in the industry. And your company is desperately trying to escape a hostile take over, and you're in charge of researching financial trends in the industry. You decide to enlist the help of some hot shot financial analyst, and unfortunately you find my number, an analyst with a shady Wall Street company who has been involved in covert financing of the Irish Republican Army.

Anyway, your desktop phone has multiple line capability (up to 10); each line is accessible by a button which flashes when an incoming call arrives on the line. So with my number in hand, you select a line on your phone; wait for the dial tone; dial; and voila, you are talking to my receptionist, 1800 miles away. You ask the receptionist for my extension. I have a similar phone and one of the buttons is flashing. I push the button and start listening to your company's woes.

What does this have to do with our design? With a little a bit of imagination, it's not hard to see at all. The companies for which you and I work are analogous to computer processes. Obviously, communication between other companies is part of our business, but our companies have specific missions and methods for accomplishing those missions. Certainly the notificationServer and the IGC_Process are processes needing communication with other processes, but both have specific reasons for being beyond IPC. You and I are analogous to objects in a process, each having a specific task in the context of a larger mission. The multi-line phone manages connections and thus would be considered a connection manager. Our design has a connection manager except we call it a SocketConnection object. Each button on the

phone manages a communication endpoint. That certainly fits the definition of a socket. Between my button and your button is an electronic telephone line (or maybe fiber-optic, but neither of us really cares about the details). Between every two sockets is a two-way socket buffer; the actual implementation of this buffer can be many things, all transparent to us.

What about the receptionist? I'm glad you asked. No actual data is transferred between him/her and you. You're just requesting a connection between me and you. The receptionist uses the same phone we are using, so s/he has access to a button/socket also. Except this button is used only for accepting and forwarding connections. Every company usually has some sort of receptionist, albeit some are automated, but usually there is only one per company. Our design has a receptionist also, except we call ours an `AcceptSocket` object. Like the receptionist, there is only one per process, and its sole function is to accept and forward connections.

Back to me and you. As I talk, you listen and vice versa. This is a polite conversation, even though I have a Type A personality. The words I use are converted to electronic signals; sent over the phone line; and then converted back to words. Our IPC messages start out in data formats that the client object understands, but are converted to a binary stream that garners host/architecture independence. Once the message is received, it is quantized back into words for the benefit of the client objects that process the incoming message. Once you hear my message, I may look up some figures on my PowerBook, and then reply. Our communication is two way, and we are exchanging data back and forth. Just as the receptionist's button is analogous to a socket, it had a special purpose, so we gave it a special name, `AcceptSocket`. My button and your button are also analogous to sockets. In our design, TCP sockets that are endpoints between two-way communication lines that transfer actual data messages are encapsulated in what we call `DataSocket` objects.

A `SocketConnection` object can have multiple `DataSocket` objects; each connected to a different process. However like our phones, there is a limit to how many active connections a process can simultaneously maintain. What if I needed to call you, and all my lines were being used? One possible solution is to hang up on the person whose line has been idle the longest. Not a solution that will give you style points or good karma, but at least it will free up the line for that important phone call to me. Our design does the same thing. When a process has reached its limit of outside connections (usually about 20), then the `SocketConnection` object looks for the least recently used `DataSocket` object, and destructs the object which is very analogous to hanging up on someone. But in the digital world, it is not quite as rude.

Ok, we admit it. Here's where the analogy gets kind of creepy. Suppose that I try to call you. I get through to the receptionist, but since you are using the last line for some other conversation, I get put on hold. Perhaps, you are talking to your boss about a raise, or maybe to your significant other about some new way of (never mind...). Meanwhile, I have this important message to give you, and even though the music to the Hawaii Five-O TV show is very refreshing, I have some important work to do, and can't do it effectively being on hold. So what do I do? I clone myself. My clone waits for your line to become available, and then gives you my message and kills himself. Very tragic, I know, but at least I was able to get some work done

while my clone waited for you to get off the line. This sounds very similar to the problem of a process sending a message to another process that has a full socket buffer because the destination process is busy talking to its significant other, the meteorologist. Our solution is similar to the cloning idea although it doesn't have as many moral ramifications. We cut a thread whose job is to wait for the destination process to read some of the socket buffer. The thread then writes the message and exits once the message is complete. Cloning can be expensive (new technology), so we want to clone only when absolutely necessary. Likewise, thread creation carries some overhead with it, so we want to cut a thread only when the socket buffer is full.

If you can indulge this metaphor a little longer, we have one more point to deliver. Suppose you are on the phone with your significant other, and your boss knocks on your door. BUSTED! He interrupts you with some important information about a tech review that you couldn't care less about. You have a great idea. You use a clone that will receive your boss's information. Then the clone calls you up, and waits on hold until you get off the phone. The clone then gives you that very important information, but he doesn't kill himself because he may be useful later on. We handle asynchronous signals in very much the same manner. Suppose the IGC is busy rendering a depictable and a SIGUSR1 signal comes in. We have a signal thread that is always running which receives the signal. The thread then writes the signal to a pipe managed by a SignalPipe object. A pipe is very similar to a socket, and in fact is identical to a socket as far as the EventDispatcher is concerned. The IGC reads the pipe and processes the signal once the depictable rendering is complete.

5a.3.2 Object Interaction

Now that we introduced the basic concepts, this section describes the major objects of the IPC library and how they interact.

5a.3.2.1 SocketConnection

SocketConnection objects manage a group of sockets of a particular priority. These objects are derived from the Connection class. This is to support a future possibility that our IPC library will have another transport mechanism in addition to TCP sockets.

Currently we support only two levels of message priorities, normal and high. At static initialization, a process creates a SocketConnection object for normal messages. A second SocketConnection object is created only if the process sends or receives high priority messages.

5a.3.2.2 AcceptSocket

The AcceptSocket encapsulates a TCP socket whose sole responsibility is to listen for and handle connection requests from other processes. Its parent is the normal priority SocketConnection object. There is only one AcceptSocket object active in a process. It handles connection requests by creating a new socket and passing it on to its parent.

5a.3.2.3 DataSocket

The DataSocket also encapsulates a TCP socket; however, its use differs from the AcceptSocket. DataSocket objects manage a data transmission endpoint between processes. These objects are capable of both receiving and sending data to/from a DataSocket object living in another process. Each is managed and owned by one of the two possible SocketConnection objects. Only 19 DataSocket objects can live in a process simultaneously. If the limit has been reached, and a new one is needed, then one of the two SocketConnection objects will destroy the least recently used DataSocket object.

DataSocket objects can be constructed in two ways depending on whether the object is used to initiate a connection to another process or if the object was created to receive a connection request initiated by another process. Both constructors perform some mutual initialization tasks which are executed by the private member, DataSocket::initialize() that both constructors invoke. These tasks include initializing the data structures needed for thread management, and adjusting the socket buffer size to the value specified in the configuration file, localization/nationalData/ipc.config.

The DataSocket object also manages the threads that may be used to send to the connecting process. If a DataSocket object is destroyed, running threads are also terminated. The object also maintains mutual exclusion locks (mutexes) to ensure that only one thread at a time will be writing to a socket.

5a.4 Thread Based Socket IPC Implementation

In describing the implementation, we've focused on the major tasks of the IPC library and how each task is accomplished. In the interest of brevity, we have omitted some of the less important details which are documented with the code. We have identified the essential tasks of the IPC library as the following:

- Managing IPC addresses and assigning addresses to processes.
- Waiting for IPC and other events to arrive.
- Client Interface for sending a message.
- Data transmission.
- Client Interface for receiving a message.
- Establishing and breaking a connection to another process.
- Synchronously and asynchronously waiting to write to a full socket buffer.
- Signal handling in a single and multiple threaded process.

5a.4.1 Addressing

In order to connect to a TCP socket in a remote process, the local process must know the IP (internet protocol) address of the remote host, and the port number assigned to the socket inside the AcceptSocket object of the remote process. This information is encapsulated into an IPC_Target object and stored internally using a UNIX struct, sockaddr_in (defined in /usr/include/netinet/in.h) which is the

data format passed to UNIX socket system calls. This class has methods to encode and decode to/from a binary stream. An `IPC_Target` object can be constructed with either a text string or a `sockaddr_in`. A text string representation of the address is useful for logging and for passing the parent address as a program argument to a child process. The string format is dependent whether the process is anonymous or named.

5a.4.1.1 Anonymous Addresses

Anonymous IPC addresses are usually used by transient processes, such as a process awaiting a reply from a daemon, or two processes (usually in a parent/child relationship) that need to converse, but don't need to identify themselves to the rest of the system or network. `WarnGen`, `fxa`, `fxaWish`, `IGC_Process` are all examples of anonymous processes.

Anonymous processes from the same executable will each have different IPC addresses since the port number will be generated by UNIX when the `AcceptSocket` object is created. This allows many instances of the same executable to be running simultaneously; each instance can be addressed independently.

The text string representation of an anonymous address has the following format: `<host name>/<port number>/<process id>`. Host name can either be in fully qualified domain format: `vulture.fsl.noaa.gov` or as a dotted IP address: `127.0.0.1`. The process ID is not used in the internal address representation and is not needed to connect to a socket. However, it is useful for logging and debugging purposes since it is much easier to identify a process by its process ID than by its port number.

5a.4.1.2 Named Addresses

Named addresses are usually used by processes that provide a well-known service. Such processes are usually daemons, awaiting a request, performing some action, and sometimes sending a response back. `NotificationServer`, `CommsRouter`, and `RadarTextDecoder` are all examples of processes that use named addresses.

Only one instance of an executable with a named address can be running on the network at one time. Named processes must run on the host specified in the system configuration file (`ipc.config`) and their `AcceptSocket` object will use the port number specified in the config entry for that process.

A config entry for a named process has the following format: `<process name> <host name> <port number>`. The process name has no restrictions but usually coincides with the executable name. The text string representation of an address for a named process is simply the process name specified in the config file. As with anonymous targets, host names can either be in fully qualified domain format or as a dotted IP address. The port number can be any positive number less than 32K, but all the named processes running on the same host must have unique port numbers.

5a.4.1.3 Assigning an Address to a Process

Both anonymous and named processes must be assigned an IPC address before any messages can be received or sent. A client of the IPC library can query the process address by calling the static method, `Connection::myTarget()`.

How a process is assigned its address depends on whether it's an anonymous or a named process:

- For an anonymous process, the assignment happens at static initialization time (before the process executes `main()`). A `SocketConnection` object for managing normal priority sockets is constructed. Its constructor creates an `AcceptSocket` object, which will be the only one in this process. The `AcceptSocket` constructor will create a TCP socket and bind it to a null address. When passing a null address into a UNIX `bind()` call, UNIX generates the port number. The constructor then obtains the address of the socket by calling UNIX's `getSockName()` and writes the address to a static variable in the `Connection` class. Once the address is assigned, we put the socket in a mode where it will listen and be ready to accept up to 20 connections.
- For a named process, the address is assigned in `main()`. Before any IPC is sent or received, the static method `Connection::setMyTarget()` is called by passing an `IPC_Target` object with an address for a named process. It is not possible for a client of the IPC library to assign the address of an anonymous process. Before the address is assigned, we check to see if the process is running on the host specified in the address. Also, we check to see if there is another process running with the same address by trying to make a connection using that address. See Section 5a.4.6 for the details of establishing a connection with another process. If a connection is made, then we break the connection immediately and abort the address assignment. If we failed to connect, the static variable maintaining the process address is updated. The `AcceptSocket` object created at static initialization time now has an address for an anonymous target. The parent `SocketConnection` object will destroy the existing `AcceptSocket` and create a new one, passing the named address to the `AcceptSocket` constructor. The UNIX `bind()` call will not generate a port number but rather use the port number in the named address. Other than that, the socket is created and prepared in the same way as the anonymous case.

5a.4.2 Waiting for Events

Most D2D display and ingest processes initialize and then wait for events or UNIX signals to arrive. As the event arrives, it is processed by the object interested in the event. This continues until an event or a signal causes the process to terminate. Objects interested in events fall into two categories.

- An object wants to be notified when its I/O device has become ready for reading, writing, or an error condition has occurred. In order to accomplish this, the object derives from the `DescriptorEventClient` abstract base class. By overriding several virtual functions, it tells the IPC library which routine to call when the device is ready, the UNIX descriptor for the device, the type of I/O that should be monitored, and the priority of the device. The IPC library has three classes of objects that fall in this category: `AcceptSocket`, `SignalPipe`, and `DataSocket`. Objects that manage X events, the keypad, application streams, etc. also fall into this category.
- An object wants to be notified when the current time has exceeded a specified time. In order to accomplish this, the object derives from the `TimerEventClient` abstract base class. By overriding

several virtual functions, it tells the IPC library which routine to call when the time has been exceeded, and the time to check against the current time. No IPC objects fall into this category but an example of an object that does is the one that manages looping inside the IGC_Process.

The EventDispatcher singleton object maintains three sets of client objects.

1. All objects derived from DescriptorEventClient that are currently active in the process.
2. All DescriptorEventClient objects that are part of the IPC library. This is a subset of the first set.
3. All objects derived from TimerEventClient that are currently active in the process.

In order for the EventDispatcher to maintain these sets, client objects must register with the singleton during construction and cancel their registration during destruction.

A process can wait for events to arrive in four different ways.

- Wait for a single IPC message arriving from any process.
- Wait for a single IPC message arriving from a specific process.
- Continually wait for all types of events to arrive.
- Have Tcl/Tk continually wait for IPC messages along with other events registered with Tcl/Tk.

5a.4.2.1 UNIX Select

No matter which way a process waits for events, the UNIX select() routine is used to wait for the devices to become ready until a time-out has been reached. This routine is particularly efficient since processes truly sleep until an event arrives.

The IPC library contains a global function, selectDescriptorEvents, which is a wrapper around the select() call. It is passed a set of DescriptorEventClient objects and a pointer to UNIX struct that represents time with a granularity of microseconds. A null pointer can be passed in which indicates that the select() will never time out. This routine blocks its process until one of the devices is ready or the time-out value has been exceeded. It uses the objects to construct three sets of file descriptors indicating which devices are waiting for reading, writing, or an exception. It passes these sets and the pointer to the time struct to select(). If select() returns because one or more devices are ready, selectDescriptorEvents determines which objects are ready by examining the descriptor sets set by select(). It then invokes the callback, DescriptorEventClient::handleEvent() for each object that is ready. If two or more objects are ready at the same time, the objects with higher priority devices will be notified first. SelectDescriptorEvents() will return an enumerated value depending on the four possible results of calling select() which are:

- select() was interrupted by a signal.
- select() failed because of invalid file descriptors or time-out value.
- None of the specified devices was ready in the time allotted.
- One or more devices is now ready for the specified operation.

5a.4.2.2 Waiting for IPC from any Source

In order to wait for a single message arriving from any process, clients of the IPC library should call the static method, `Connection::waitForMessage()` and pass in a time-out value. The time-out value is specified in deciseconds (a relic from our DMQ days we can now support time-out granularity up to microseconds, but we decided not to change the interface since developers are used to it). The client can also use the enumerated values: `IPC_Types::NO_WAIT` and `IPC_Types::WAIT_FOREVER` as the time-out value.

If the process wants to wait continually for IPC messages to arrive, this code fragment might be used in `main()` after all initialization is complete.

```
while (Connection::waitForMessage (IPC_Types::WAIT_FOREVER) ==  
      IPC_Types::IPC_SUCCESS);
```

A caveat of using this approach is that only IPC devices are monitored. Objects that manage non-IPC devices will not be notified when their devices are ready. Also, `TimerEventClient` objects will not be notified when their timer has expired. If the process has these kinds of objects, then the approach described in Section 5a.4.2.4 should be used.

Here is how `Connection::waitForMessage()` implements this approach.

1. Converts time-out value to the UNIX struct representing time in microseconds.
2. Asks the `EventDispatcher` object for the set of `DescriptorEventClient` IPC objects.
3. Calls `selectDescriptorEvents()`, passing in the data obtained in steps 1 and 2.
4. If a signal arrived or if `selectDescriptorEvents()` times out, return `IPC_Types::IPC_TRY_AGAIN_LATER`. If invalid arguments were passed in, return `IPC_Types::IPC_HOPELESS`. Otherwise, return `IPC_Types::IPC_SUCCESS`.

5a.4.2.3 Waiting for IPC from a Particular Source

In order to wait for a message from a particular process, the client also calls `Connection::waitForMessage()`. In addition to the time-out value, the client passes a pointer to an `IPC_Target` object containing the address of the source process.

Here's how we implemented this approach.

1. Two static variables are set. One contains the address of the source process, and the other is a flag indicating whether the desired message has arrived. Initially it is set to false.
2. It performs the first three steps of the algorithm described in the previous section. However, step 3 is timed using a `StopWatch` object.
3. Each `DataSocket` object that gets notified that its socket is ready for reading will check to see if the address of the process to which it is connected matches the address in the static variable set in step 1. If not, no data is read from the socket. If so, once the complete message is read, the flag from step 1 is set to true.
4. Check the return status of `selectDescriptorEvents()` call. If it timed out, `Connection::waitForMessage` returns `IPC_Types::IPC_TRY_AGAIN_LATER`. If one of the sockets

was able to read, check the flag to see if the socket connected to the source process received a message. If so, we are done, so return `IPC_Types::IPC_SUCCESS`. If not, subtract the time it took to execute `selectDescriptorEvents()` from the time-out. If there is still time remaining, repeat steps 2 - 4.

5a.4.2.4 Continually Waiting for All Events

Display processes that need to collectively wait for IPC, X, stream, keypad, and timer events must apply this approach. After process initialization, `main()` should call `EventDispatcher::enterDispatchLoop()`. This method will not return until an object handling an event or signal calls

`EventDispatcher::exitDispatchLoop()`. Typically, process cleanup code is found after a return from `EventDispatcher::enterDispatchLoop()` since the process is about to terminate. The dispatch method operates on all the active `DescriptorEventClient` objects and `TimerEventClient` objects that have been registered during process initialization or in response to an event callback.

The `EventDispatcher` supports multiple interchangeable dispatch engines. Even though we currently have only one variety, we may have several in the near future, one for Tcl executables and one for non-Tcl/Tk executables. To specify which engine to use, a client passes one of these enumerated values into `EventDispatcher::enterDispatchLoop()`: `EventDispatcher::GENERIC` or `EventDispatcher::TCL`. If no engine type is specified, then the generic engine is assumed.

Note: Tcl/Tk *interpreters* built in our software tree do not use our event dispatcher at all but use the Tcl event notifier directly. Tcl/Tk interpreters are executables that are used to interpret Tcl scripts; Tcl/Tk executables are programs which have Tcl/Tk built into them, but otherwise are used for their own purposes instead of for script evaluation. These programs would use our `EventDispatcher` with the `TclDispatchEngine`, which is yet to be developed.

We will discuss only the generic dispatch engine since the Tcl engine has not been implemented yet. The implementation is very similar to what was described in Section 5a.4.2.2. The dispatch method will continue to loop until an internal flag inside the `EventDispatcher` singleton is set to false by calling `EventDispatcher::exitDispatchLoop()`. Inside the body of the loop, the method does the following:

1. Invokes the callback for any registered timer objects that have expired by checking the current time against the timer's expiration time.
2. While looking for expired timers, we also identify the timer object that is about to expire next. We subtract that timer's expiration from the current time and then convert the difference into the UNIX struct that represents time in milliseconds.
3. `SelectDescriptorEvents()` is then called by passing all the registered `DescriptorEventClient` objects (not just the ones dedicated to IPC) and a pointer to the time-out that was calculated in step 2. It is quite possible that there are no timer clients that are ready to expire. In that case, a null pointer to the time struct is passed in, instructing the `select()` call to never time out.
4. If `selectDescriptorEvents()` has timed out, then the callback for the timer object found in step 2 is invoked.

5a.4.2.5 Using Tcl/Tk's Notifier for Event Waiting

Tcl/Tk interpreters and scripts use the Tcl/Tk event notifier for dispatching X events, file and device I/O, and timer and idle events. In order for those executables to use our IPC library, all the objects that manage sockets and pipes must register with this mechanism so that these will be notified when their devices are ready.

For most of our interpreters, IPC is considered a module and is initialized by the global function, `IPC_Init()`. This function creates a Tcl command that a script can invoke which simply invokes either `Connection::myTarget()` and `Connection::setMyTarget()` which were explained in Section 5a.4.1.3. More importantly, all the `IPC DescriptorEventClient` objects that were registered with the `EventDispatcher` at static initialization are now registered with Tcl. This is done by calling `Tcl_CreateFileHandler()`, passing the UNIX file descriptor, a mask describing the type of I/O to wait for, a callback, and a pointer to the `DescriptorEventClient` object. Unregistering with Tcl is done with `Tcl_DeleteFileHandler()`. When Tcl invokes the callback, it passes in the pointer to the `DescriptorEventClient` object. Thus, the callback can call the callback method of the object, `DescriptorEventClient::handleEvent()`.

After `IPC_Init()` is executed, socket and pipe objects will register with Tcl when they also register with the `EventDispatcher` singleton. Likewise, when these objects are destructed, they cancel their registration with both the `EventDispatcher` and Tcl.

5a.4.3 Client Interface for Sending a Message

This section covers how clients of the IPC library specify and send messages to other processes. We'll also discuss the implementation details of how a message is packaged and prepared for sending. The nitty gritty details of selecting a socket and writing to it are deferred to Section 5a.4.4.

5a.4.3.1 Packaging the Data

Clients who want to send a message must first pack the data comprising the message into an `ArgPkg` object. `ArgPkg` is a set of template classes that provides for marshalling of objects or variables into a sendable byte stream, and back. There is a class for each possible quantity of data objects up to 11 objects. There is also an abstract base class from which each of these 12 classes is derived. For example, if a client wanted to send a message that contained a `DateTime` object, a text string, and a double precision floating point value, the client would instantiate an `ArgPkg3` object using the constructor that accepts the three pieces of data. The declaration might look like this:

```
DateTime aDateTime;  
TextString aString;  
double aNumber;  
ArgPkg3<DateTime, TextString, double> args (aDateTime, aString, aNumber);
```

An important caveat is that each data type must have an associated `serialize()`, `serialLength()`, and `quantize()` routine.

To convert from data to binary, the following set of functions is used:

```
byte* serialize(const <some data type>& arg, byte* addr)
```

The data to write (`arg`), and the binary stream to write to (`addr`) are passed in. The routines return the pointer to the byte on the stream after the data just written.

To determine how many bytes a piece of data occupies on the byte stream, the following functions are used:

```
long serialLength(const <some data type>& arg)
```

The object or variable is passed in (`arg`) and the number of bytes it will take to encode onto a binary stream is returned.

To convert from binary to data, the following set of functions is used:

```
byte* quantize(<some data type>* arg, byte* addr)
```

A pointer to the data to read (`arg`), and the binary stream to write to (`addr`) is passed in. The caller is responsible for allocating memory for the data. Quantize methods do not allocate any memory. The routines return the pointer to the byte on the stream after the data just read.

Fortunately, the conversion functions for most all of the common data types have been written. Occasionally, a client may want to send an unusual class of object across process boundaries and will have to write these conversion routines. Complex data types are converted by calling `serialize`, `serialLength`, or `quantize` on the individual data members that are to be converted. The author of the `serialize/quantize` functions for a particular class can choose which data members will be encoded/decoded to/from a binary stream. For atomic data types, we use the XDR package for our conversion needs. XDR was a good choice since it's available on a great number of UNIX and non-UNIX systems. In fact, any system that has NFS had better have XDR.

With an `ArgPkg` object, data is converted to binary by simply calling its `serialize()` method. This method in turn calls the `serialize()` routine for each of the packaged data types. An `ArgPkg` object also supports a `serialLength()` method that is implemented in the same recursive fashion.

5a.4.3.2 Packaging the Metadata

Once the data is packaged, the next step is to instantiate a `ParameterizedMsg` object. This is constructed with an `ArgPkg` object and the following metadata that describes how the message should be sent and received:

- **delivery mode:** This enumeration describes if a message is to be sent synchronously or asynchronously. The difference between synchronous and asynchronous is described in detail in Section 5a.1.1 and Section 5a.1.2.
- **priority:** This enumeration describes the priority of a message for both sending and receiving. Currently, only two levels of priority are supported: normal and high.

- **module:** Every one of our IPC messages is grouped into a logical categorization. For example, all messages that pertain to the file access controller are assigned the module identifier: `FAC_MODULE`.
- **type:** A module can contain one or more different types of messages. The client usually defines an enumeration that identifies the types for the module. This attribute is one of those values.

The delivery type and priority default to `IPC_Types::ASYNC` and `IPC_Types::NORMAL_PRIORITY`, so these can be omitted by the client.

Quite often, a client develops a class that inherits from `ParameterizedMsg`. This is usually convenient, but not necessary unless the client wants to directly control how the message data is translated to binary. In this case, the client passes a null pointer for the `arg` object when constructing the message object. The derived class should then provide implementations for the virtual methods: `structToByteStream()` and `byteStreamLength()`.

5a.4.3.3 Initiating the Send

Now that the data and the attributes are packaged, the client can initiate the sending of the message by calling one of the two send methods, passing either a pointer to a `ChildProcess` object or an `IPC_Target` object (described in Section 5a.4.1), indicating where the message should be sent. These methods will return one of the enumerations of `IPC_Types::ErrorCodes`. If the delivery type is `IPC_Types::SYNC`, then these methods may take a while to return, especially if the destination process is not responsive, or network traffic is heavy. These methods will return immediately if the transmission is asynchronous. Clients should check the return values of these methods. If the methods return `IPC_Types::IPC_TRY_AGAIN_LATER`, then the destination is unreachable. If `IPC_Types::IPC_HOPELESS` is returned, then there is something wrong with the destination address that was passed in or some of the meta-data values are invalid.

As soon as one of the send methods return, the `ParameterizedMsg` object is no longer needed unless the client wants to send the same message at a later time.

5a.4.3.4 Preparing for the Send

Here is a peek at the implementation of the two `ParameterizedMsg::send` methods.

Most of the real work is done by `ParameterizedMsg::send (const IPC_Target *target)`. `ParameterizedMsg::send (const ChildProcess *process)` simply extracts the `IPC_Target` from the process object and then calls the other send method which follows the following algorithm.

1. Check the `IPC_Target` for validity. If it is null, we return immediately with the value of `IPC_HOPELESS`.
2. Obtain a `Connection` object by passing the message priority to `Connection::getConnection()`. If this fails, then the message priority value is invalid so we return immediately with the value of `IPC_HOPELESS`. If the priority is normal, then the `SocketConnection` object created at static initialization is returned. If this is the first high priority message sent or received by the process,

then a new `SocketConnection` object dedicated to high priority messages is constructed and returned.

3. Construct the message header struct. This struct has the following fields:
 - o `_length`: The number of bytes for the entire message, data and header. The size of the data can be obtained by calling `ParameterizedMsg::byteStreamLength` which simply asks the `arg` object to return its serial length.
 - o `_sourceTarget`: An `IPC_Target` object containing the address of the process sending the message. This can be obtained by calling `Connection::myTarget()`.
 - o `_module`, `_priority`, `_type`: This is the message metadata described in Section 5a.4.3.2 that will be needed to receive the message. Notice the delivery mode is not part of the header since that attribute is used only in the sending of the data.

The message header struct also has `serialize`, `quantize`, and `serialLength` routines, since the header will be encoded and decoded to/from the binary byte stream.

4. Dynamically allocate a byte stream (an array of 8 bit integers) big enough to hold the data and the header.
5. Write the header to the front of the byte stream by calling the `serialize` method for the header struct. Write the message data to the byte stream behind the header by calling `ParameterizedMsg::structToByteStream()` which simply calls the `serialize` method for the `arg` object.
6. Initiate the transmission by calling the `send` method for the `Connection` object obtained in Step 2, passing the priority, delivery type (`sync` or `async`), the destination address, and the byte stream. Return to the caller the `IPC_Types::ErrorCodes` value that is returned from `Connection::send`. Keep in mind that `Connection::send` is a pure virtual method. Currently, only `SocketConnection` is derived from `Connection`, although it is possible that we may someday add additional derivations of `Connection` objects.

The message is now completely packaged, converted and ready to transmit!

5a.4.4 Data Transmission

This section describes how a binary stream of bytes representing an IPC message is transmitted between two processes. Here we will assume that the connection between processes has already been made. Section 5a.4.6 will describe how connections are established. We are also assuming that the socket buffer between the processes is not full. If it is, then sending the message becomes much more complicated; this will be covered in Section 5a.4.7.

5a.4.4.1 Selecting a Socket

When a client initiates a send, `SocketConnection::sendMsg()` eventually gets called with the byte stream to send, the address to send to, and instructions on how to send (`sync` or `async`). A `SocketConnection` object maintains a dictionary of `DataSocket` objects indexed by the address (`IPC_Target` object) of the connecting process. This method uses the specified address to locate a `DataSocket` object from the table. If one is found, then we need to confirm that the socket managed by the object is still connected. We do this without actually sending or receiving data by using the UNIX routine, `getpeername()`. If that call fails, or we couldn't find an associated `DataSocket` object, then we should try to establish a

connection. If we cannot establish a connection, then it is safe to abort the send with a status of `IPC_Types::IPC_TRY_AGAIN_LATER`. If `getpeername()` is successful, then we have found a socket and we can begin writing to it.

5a.4.4.2 Writing to a Socket

After `SocketConnection::sendMsg()` has identified the socket object to use, the byte stream and a pointer to the data socket object are packaged into a `PendingMsg` object. `PendingMsg` is a fairly simple class of objects that supply a method for writing to a socket (`PendingMsg::send()`) and methods indicating whether the send is finished and whether it is successful.

`SocketConnection::sendMsg()` then invokes the `DataSocket::writeMsg()` method on the selected socket object, passing the delivery mode and the `PendingMsg` object. `DataSocket::writeMsg()` will return a status indicating whether the write was successful; that status value is then returned by `SocketConnection::sendMsg()`.

The first task for `DataSocket::writeMsg()` is to record the current time. This will be useful in determining which is the least recently used socket.

Next, the socket object places the socket in non-blocking mode. This means that if we try to write to a full socket, the `write()` call will return immediately with a status of `EAGAIN`. We use `UNIX fcntl()` calls to toggle between blocking and non-blocking mode.

As we mentioned earlier, the `PendingMsg` object will perform the write to the socket. The constructor of this object determines how many bytes are in the message stream by reading the first four bytes of the binary stream. `PendingMsg::send()` will call the `UNIX write()` routine, which does the socket writing. `Write()` will return the number of bytes written to the socket. At this point, there are three possibilities:

- If the entire message was written to the socket, then the `PendingMsg` object sets flags indicating that the send is complete and successful.
- If `write()` failed because of a broken pipe error, the object sets flags indicating the send has completed and is not successful.
- If a portion of the message was written, then a pointer is set to the unsent portion of the byte stream, and the number of bytes written is subtracted from the number of bytes left to write. The `PendingMsg` object sets the "send completed" flag to false.

Because of that third possibility, it may take several calls to `PendingMsg::send()` before the complete message is sent.

After the `DataSocket` object places the socket in non-blocking mode, the `PendingMsg` object is asked to try to write its message. The socket is then immediately placed back into blocking mode. If the write has completed either successfully or unsuccessfully, then we are done. Yippee! If the write did not complete, the receiving process must be busy with some other task which caused the socket buffer to fill up. With that being the case, the situation becomes much more complicated and is explained in detail in Section 5a.4.7.

After the write completes, the PendingMsg object will be destructed causing the binary stream to be deallocated (which can be a fair chunk of memory for large messages).

DataSocket::writeMessage() will then return a status indicating whether the write has succeeded or not. If SocketConnection::sendMsg() gets a status back indicating the write has failed, we can assume that the connecting process no longer wants to engage in conversation. As a result, we destroy the DataSocket object which closes our end of the connection, and then remove the socket object from the table of connections.

5a.4.4.3 Detecting Message Arrival

The sending process has just successfully placed a complete binary stream message on the socket buffer. The destination process must punctually remove and process this message in order to avoid slowing down the sending process. Let's assume that the destination process is now idle waiting for events to arrive. SelectDescriptorEvents() (mentioned in Section 5a.4.2.1) detects that there is data to read on one of the socket devices that it is monitoring. It tells the object managing the socket by calling the object's DataSocket::handleEvent() method.

5a.4.4.4 Reading the Socket

The primary task of the method DataSocket::handleEvent() is to remove binary data from a socket and to assemble that data into a message that can be received and processed by clients of the IPC library. It is possible that the message will be sent in fragments, requiring multiple calls to this method in order to read the entire message. The following algorithm implements this method.

1. We record the current time. This will be useful in determining which is the least recently used socket.
2. If the client is waiting for a message from a particular process, and this socket is not connected to that process, return immediately without reading any data. See Section 5a.4.2.3.
3. The DataSocket object maintains a pointer to a buffer capable of containing a binary stream. If that pointer is null, we are about to read a new message. If it is not null, we are about to read a fragment of a message that we started reading in an earlier invocation of this method. If we are reading a new message, read the first four bytes of the socket using the UNIX read() function which will return the number of bytes to read (the length of the message). If the read returns 0 bytes, that is a sign that the connecting process has broken the connection. In response to that, we close our socket by asking the parent SocketConnection object to destroy this socket object. Since the length is now known, we allocate a byte buffer big enough to hold the message and copy the length to the front of the buffer. Some data members are also initialized, recording the message byte length and how many bytes have been read thus far.
4. We call UNIX read() again, attempting to read the rest of the message, passing a pointer to the unfilled portion of the read buffer, and the number of bytes still left to read. Again, if read() returns 0, we close our socket. Once read() returns, we update the number of bytes read thus far. If we haven't read the entire message yet, we return here and wait for DataSocket::handleEvent() to be called again.
5. Once the entire message has been read, the binary byte buffer is passed to the static method, Dispatcher::route() which will pass the message on to the client. The Dispatcher singleton is

responsible for deleting the byte buffer once the client is finished processing. The data members that keep track of the reading progress are initialized to null values.

5a.4.5 Client Interface for Receiving a Message

The previous section described how the IPC library receives a binary byte stream that represents a message arriving from some other process. This section describes how the binary message is passed to client code, and how the byte stream can be converted into a meaningful data representation that the client can use.

5a.4.5.1 Logical Modules and Message Types

IPC messages for a particular purpose are grouped into a logical module. For example, logical modules might contain messages that pertain to data notifications, log stream, the file access controller, extensions, etc. A process can have as many logical modules as required. Each logical module has an identifying enumerated value associated with it, defined in `IPC_Types::Modules`. As mentioned earlier, the metadata passed with every message includes this logical module identifier.

In addition to the logical module, every message has a type associated with it. The type tells the client what the format of the data is, and what the message should be used for. For example, one type of message could be an instruction for the IGC to load a color table. This type of message contains one piece of data, a color table key. A logical module may contain one or many types of messages. Like the logical module, a type has an enumerated value associated with it that is passed with every message as part of the metadata.

5a.4.5.2 Use of Receiver Objects

For every logical module, the client must instantiate a receiver object which receives all the messages for that module. Ideally, this object should be created during initialization before a process begins waiting for events. This object must be derived from the abstract base class `Receiver` containing a single pure virtual function, `receive()`, for which the client must provide an implementation. When the IPC library calls this method, it passes the byte stream, the number of bytes in the stream, the address (`IPC_Target` object) of the sending process, and the message type.

5a.4.5.3 Delivering the Message to the Receiver

As mentioned earlier, a process can have multiple logical modules and thus multiple receiver objects. The IPC library manages the multiple receiver objects with a singleton `Dispatcher` object (not to be confused with the `EventDispatcher`). The `Dispatcher` maintains a dictionary of receiver objects indexed by the logical module identifier. Thus, receiver objects must register with the `Dispatcher` soon after they are created. Lots of our receiver objects are instantiated and registered at static initialization time. The advantage of a static declaration in the `.C` file is that a client process just has to link with the `.o` file, and the receiver is all ready to do its thing.

Once a `DataSocket` object receives an entire message, it passes the byte stream to the static method, `Dispatcher::route()`. This method will then extract the message header from the byte stream, and use the logical module ID in the header to locate an associated registered receiver object. The receive method for that object is then called, passing the message byte stream without the header, the size of the header-less byte stream, and some other metadata that was included in the header. Once the receive method returns, `Dispatcher::route()` deallocates the memory for the byte stream.

5a.4.5.4 Interpreting the Message

A client must provide an implementation for the method that receives different types of messages for a particular logical module. Typically, a receive method will contain a `switch` statement, selecting on the various types of messages for that module. Each case in the `switch` should convert the binary stream to some meaningful data types and then pass that data to some object or module that can process the request and data. To convert from binary to random data types, an `ArgPkg` object is used. `ArgPkg` template classes were introduced in Section 5a.4.3.1 for converting multiple objects and variables into a binary stream. They can also be used to convert a binary stream into objects and variables. For example, if a client wanted to decode a message that contained a `DateTime` object, a text string, and a double precision floating point value, the client would instantiate an `ArgPkg3` object using the constructor that accepts a binary byte stream. The individual pieces of data can then be obtained by calling the object's `param1()`, `param2()`, etc. methods. Here is a code fragment that illustrates this example:

```
ArgPkg3<DateTime, TextString, double> args (binaryByteStream);
DateTime aDateTime (args.param1());
TextString aString (args.param2());
double aNumber = args.param3();
```

5a.4.6 Establishing and Breaking Connections Between Processes

With our IPC system, a process can have up to twenty connections to other processes. Each connection is two way, allowing both the receiving and sending of messages. Connections are created as needed, and maintained until one of the processes decides to break a connection. Each process transmits data through these connections using a TCP socket, managed by a `DataSocket` object. This section describes how the IPC system manages connections between processes.

5a.4.6.1 Making Room for a New Connection

The IPC library has a self-imposed limit of allowing up to twenty connections. Since a process has a UNIX-imposed limit of sixty active I/O devices, we decided that the IPC library should at the most use a third of that valuable resource, leaving plenty of I/O devices for the client to use.

Whenever a process initiates or receives a connection request, we count the number of `DataSocket` objects that each `SocketConnection` object owns. Then we add two to the count since every process has at least one `SignalPipe` and `AcceptSocket` object also.

If our count is over twenty, we have to close down one of our existing connections. There are a number of ways we could choose the unlucky connection, all equally appealing. We opted for the least-recently-used approach. Each time a socket is written or read, we record the current time inside the DataSocket object. We destroy the DataSocket object with the oldest recorded time which will close the socket and allow us to have a new connection without exceeding our limit.

5a.4.6.2 Initiating a New Connection Request

When a process wants to send a message to another process, the SocketConnection object associated with that message's priority tries to locate a DataSocket object that is connected to the destination process. If one can not be found, then the sending process will initiate a new connection request.

To initiate a new connection request, the SocketConnection object makes room for the new socket if necessary. It then constructs a DataSocket object by passing the address of the destination process. This DataSocket constructor creates a TCP socket and then calls the UNIX routine `connect()`, passing the address. The address of the process to connect to is actually the IP address of process host and the port number of the TCP socket managed by the connecting process's AcceptSocket object. As long as the accept socket has been created, and a listen request has been submitted, a connection can be made, even if the connecting process is suspended or busy processing. Generally, `connect()` will determine quickly whether the connection is feasible, although it is possible that `connect()` could block the initiating process if network traffic is heavy. For now, we decided to allow the blocking because during our testing, `connect()` always returns almost immediately. If it is a problem later on, we can install a time-out value on the connection. This can be done by placing the socket in non-blocking mode, and then issuing a connect request. If `connect()` returns `EINPROGRESS`, invoke a `select()` call, waiting until the socket becomes writable or the time-out has been exceeded.

If `connect()` returns `ECONNREFUSED`, we know that the connection cannot be made because the address was bad, the network or the connecting host is down, or the connecting process is not running. The DataSocket constructor then closes the socket and sets the file descriptor data member to -1.

After the DataSocket object is created, the SocketConnection object checks to see if the object contains a valid file descriptor. If so, the object is added to the dictionary of connected DataSocket objects maintained by the SocketConnection object. If not, the DataSocket object is destroyed, and a problem is logged with LogStream, indicating that the connection attempt has failed.

5a.4.6.3 Receiving a New Connection Request

The previous section discussed the chain of events inside a process that tries to connect to some other process. This section describes how that other process receives the request and completes the connection.

Every process capable of IPC has a single `AcceptSocket` object that manages a TCP socket whose sole purpose is to accept connection requests. When a request arrives, this socket becomes ready for reading. `SelectDescriptorEvents()` detects this and calls the `AcceptSocket::handleEvent()` method for this object. This method will tell the owner of this object, the normal priority `SocketConnection` object, to make room for a new socket if necessary. Next, the method calls the UNIX routine `accept()`. This creates a new socket that is connected to the socket managed by the `DataSocket` object living in the process that initiated the connection. `Accept()` returns the file descriptor of this socket, which is used to construct a `DataSocket` object. This constructor doesn't have much to do since a socket was already created. It just saves the file descriptor passed in, and then performs the initialization common to both constructors. The new `DataSocket` object is then given to the parent `SocketConnection` object.

At this point, the `SocketConnection` object managing normal priority sockets cannot add this new `DataSocket` object to its dictionary of connected sockets because we do not know the address of the process to which the new socket is connected. We also do not know the priority level of this socket. The `accept()` call can optionally return the address of the peer socket in the other process; however, this won't help us, since that address contains the port number of the socket managed by a `DataSocket` object. The address of the connecting process must contain the port number of the socket managed by the `AcceptSocket` object. Fortunately, the process address and the priority is sent in the header of every message, so the mystery surrounding this new socket will be resolved once its first message arrives. In the meantime, the `SocketConnection` object adds the socket object to a set of sockets that do not know yet who their connecting processes are. `DataSocket::handleEvent()`, the method that reads messages from a socket, will check to see if the connection address is known. If it isn't, the read method reads the header from the front of the byte stream, looking for the address of the sending process and the priority. The address and priority now known, this `DataSocket` object can now be placed properly in the `SocketConnection` object that manages the priority of this socket. Also, this `DataSocket` object can be removed from the set of mystery sockets maintained by the normal priority `SocketConnection` object.

5a.4.6.4 Breaking a Connection

A connection is broken between processes when one of the processes terminates either normally or abnormally. Also, a process can break a connection if it needs to make space for a new connection. Usually, one process breaks the connection, so the other process needs to detect the break so it can clean up. We can detect a broken connection with several different scenarios:

- While reading the data from a socket buffer, the UNIX routine `read()` returns 0.
- While verifying the socket is still connected, the UNIX routine `getpeername()` fails.
- While writing to a socket buffer, the UNIX routine `write()` returns -1 and the error status is set to `EPIPE`.

In order to clean up a broken connection, one of the `SocketConnection` objects destructs the `DataSocket` object and removes it from its dictionary of known sockets or its set of unknown

sockets. The DataSocket destructor closes the socket, freeing up the file descriptor for other incoming connections. Also, any threads waiting to write to this socket are terminated.

5a.4.7 Waiting to Write to a Full Socket Buffer

It's easy to imagine a scenario of how a socket buffer between two processes can become full. One process may be continuously writing to the buffer, while the other process is suspended, really busy processing, or initializing and is not able to read the buffer. If the socket buffer is in blocking mode, the UNIX write() routine will block until there is space on the buffer. Blocking is generally bad. Even though it is important to send the message, the sending process has more important things to do than to wait for a message to be sent. If the socket buffer size is small, then this blocking scenario is more likely to occur. Currently, we set our buffer size to be the maximum that TCP allows: 256K. We reduce the risk of filling the socket buffer, but at the cost of increased memory use. The socket buffer size can be easily configured; there is an entry for it in ipc.config. Even with a very large socket buffer, we have seen instances where a program fills the buffer. For example, the buffer between the notification server and an fxa process usually fills while the fxa process is initializing. Since a full buffer cannot be avoided, we offer two approaches for dealing with a full buffer.

- Synchronous wait: The sending process waits a certain time for space on the buffer to become available. While waiting, all processing stops. If the time-out period is exceeded, the message send is aborted.
- Asynchronous wait: For each message that will be written to the full buffer, the sending process invokes a separate thread of execution that will wait for space on the buffer to become available. The sending process can then continue with its execution. With this approach, it is possible to have many threads executing concurrently.

The best approach to use is left up to the client of the IPC library. If the client needs to know that its message has been completely received and processed, then the synchronous wait should be used.

This section discusses how to detect if these approaches are needed, and the implementation of both approaches. However, before discussing the approaches, we discuss how our design utilizes mutual exclusion mechanisms.

5a.4.7.1 Detecting a Full Buffer

Most of the time, a message can be sent without blocking. Since both of the above approaches incur overhead and ramifications, they should be used only if necessary. Before sending a message, we apply two full buffer checks.

First, we count the number of active threads that the DataSocket object has invoked. If there is at least one, we know that the socket buffer is still full, since a thread will terminate as soon as it can do its write, and a thread is created only in response to a full buffer.

If there are no threads active, the buffer could still be full. For the second check, the socket buffer is placed in non-blocking mode and the PendingMsg object will attempt to write the

entire message to the buffer. In non-blocking mode, the UNIX `write()` routine will return the number of bytes written, even if it wasn't able to write the requested number of bytes. If the entire message was not written, the socket buffer is full. After the write attempt, the socket buffer is returned to blocking mode. One hopes that the complete message will be written to the buffer, and neither of the following two approaches needs to be applied.

5a.4.7.2 Mutual Exclusion

For both approaches, this design makes use of the mutual exclusion mechanism (also called mutex locks) provided by the pthread library. Mutex locks are used to ensure that only one thread of execution has access to a data structure or an operation that is global to all threads. Thus, many threads may be waiting for a single thread to finish an operation like writing to a socket buffer. Once that thread is finished, it unlocks the mutex, and one of the waiting threads is then granted access. The choice of which thread goes next is left up to the pthread library. A first come, first served approach might make sense, but the library's choices seem to be random.

Each `DataSocket` object manages two mutex lock mechanisms. The first ensures that fragmented messages are not interleaved with other messages. Remember, sometimes it takes several `write()` calls to send a complete message. Without a locking mechanism, thread A might write 20 percent of its message causing the `write()` to return since the buffer is now full. Thread B is given access to the socket buffer and writes a different message. The `DataSocket` object in the receiving process has trouble reading the message from thread A, since it has no way of telling that a new message is interleaved.

The second mutex lock is used to protect the `DataSocket` object's thread table, a dictionary that keeps track of the running threads. If a lock were not used, it would be quite possible that the main thread of execution is reading or writing to the table while a sending thread was also writing to the table.

5a.4.7.3 Synchronous Waiting

For a synchronous message send, we not only have to wait for available space on the socket buffer, but we have to wait our turn for access to the buffer along with any other active threads. This is done by trying to lock the `DataSocket`'s mutex lock for sending. It is possible that the main thread of execution could be blocked, waiting for other threads to do their writing.

Once the main thread of execution obtains the mutex lock, it can proceed with writing the message. However, we don't want to block indefinitely! The number of seconds to wait for space to become available on the buffer is specified in the configuration file, `ipc.config`. Current configuration is set for 3 seconds. Before trying to write to the socket, the UNIX `select()` routine is used to determine if the socket is ready for writing. If `select()` returns 1, we know that there is space on the buffer and the `write()` call will not block. If `select()` returns 0, our patience has reached its limit. At this point, it is safe to assume that the receiving process is truly unresponsive, so we should break our connection with the receiving process.

The synchronous approach may sometimes be used even though the client requested the asynchronous approach. Some processes link with middleware libraries that register handlers for asynchronous signals without using our signal catcher mechanism. If a sending thread is invoked, then those signals will not be delivered consistently to the library. This is the case with our D2D executables that link with the Freeway library (wfoApi, dialRadar, etc.). The best solution is to prevent these executables from ever creating threads. This can be done by calling the static method, `Connection::preventThreadCreation()`.

5a.4.7.4 Asynchronous Waiting

With asynchronous waiting, a separate thread of execution waits for space on the socket buffer while the main thread of execution continues processing. As mentioned in Section 5a.4.4.2, a `PendingMsg` object encapsulates the byte stream for a message and coordinates the socket buffer writing. A `PendingMsg` object also has a pointer back to the `DataSocket` object managing the socket. The sending threads all execute the same routine, `doThreadSend()`, which takes a `PendingMsg` object as a parameter.

Here is the algorithm for `doThreadSend()`

1. We try to lock the mutex lock that protects against interleaved message data. This thread will probably block until the lock is obtained.
2. Keep invoking `PendingMsg::send()` on the message object that was passed in until `PendingMsg::sendCompleted()` returns true.
3. Unlock the mutex so that some other thread can now write to the buffer.
4. Tell the `DataSocket` object that this thread has just completed. The `DataSocket` object will then update its thread table, and deallocate the message byte stream. Also, if the send was not successful because the connecting process has broken the connection, a flag is set to true indicating that a thread send has failed. With every future message send, the flag is checked. If the flag is true, we destroy the `DataSocket` object, which will abort all pending sends.

This approach is far more efficient and sexier than the synchronous approach, but the client sending the message may not care about sexiness or efficiency.

5a.4.7.5 Thread Management

Every `DataSocket` object is responsible for managing the threads that are waiting to write to the object's socket. Each thread has an identifier assigned to it by the `pthread` library. As mentioned in the previous section, each thread is also passed a `PendingMsg` object, encapsulating the message to be sent by the thread. In order to keep tabs on the running threads, each `DataSocket` object maintains a dictionary of pointers to `PendingMsg` objects that have been passed to a thread. The dictionary is indexed by the identifier of the thread holding on to the associated `PendingMsg` object. As mentioned in Section 5a.4.7.2, access to this dictionary must be protected by a mutex lock since multiple threads may be accessing the dictionary simultaneously.

A thread is invoked by calling the `pthread` library routine `pthread_create()` by passing the thread routine `doThreadSend()` and a pointer to the `PendingMsg` object. The creation routine returns an identifier which can be used to make a new entry in the table.

We made a design decision that we should somehow limit the number of threads running in the system because an abundance of pending threads means that potentially a lot of message memory is allocated. Also, a plethora of running threads implies that the receiving process is not responding or is possibly hung. After mulling over several possibilities, we came up with the approach of counting the number of bytes for all the pending messages whenever we are about to create a new thread. If it is over a certain threshold value defined in `ipc.config` we will break our end of the connection, which involves destructing the `DataSocket` object and terminating all the running threads for that `DataSocket`.

Under normal circumstances, a thread terminates under its own control, and then tells its `DataSocket` object that it has finished. However, whenever a `DataSocket` is destructed, it needs to tell all the running threads to terminate, ASAP. This is done by using the `pthread` library routine `pthread_cancel()`, passing the thread identifier. Unfortunately, `pthread_cancel()` is just a request, and the return of this routine does not guarantee that all running threads have been cancelled. As a result, the destructor has to wait for all the threads to terminate before it can continue execution. It does this by trying to lock the sending mutex lock. Once it obtains the lock, we can be sure that all the threads have terminated. When a thread receives a cancel request, it does not automatically give up any mutexes that it locked. Fortunately, the `pthread` library calls a callback routine whenever a thread is about to be cancelled. Our callback routine, `doThreadCancel()`, simply unlocks the sending mutex.

5a.4.8 Signal Handling

Keep in mind that a process using this IPC library is essentially a single-threaded process until at least one message-sending thread has been invoked. Since a new thread of execution will be created only as a result of sending a message to a process whose socket buffer is full, many processes will never become multi-threaded. However, as soon as a process becomes multi-threaded, the algorithm for delivering signals in a single-threaded process no longer works. Thus, our signal catcher has been enhanced to deliver signals in both single- and multiple-threaded environments.

This section describes how a client process can register for signals and also control their delivery. It then explains the difference between synchronous and asynchronous signals and how these kinds of signals can cause re-entrancy problems. Finally, this section discusses the implementation of delivering signals in both single- and multiple-threaded environments.

5a.4.8.1 Registering for Signals

A client process handles signals by instantiating an object derived from the `SignalClient` class, and then registering that object by calling the static method, `SignalCatcher::registerSignalClient()`. The `SignalClient` class contains five virtual methods, each representing a logical grouping of signals. A client process can handle signals from one or more categories by providing an implementation for one or more of the corresponding virtual functions. Notice that these virtual functions are not pure. If the derived class does not override the implementation, then the base class will provide *reasonable* behavior for handling that category of signals.

A client process does not have to provide a `SignalClient` object; one with default behavior is created and registered at static initialization. Should the client process register several `SignalClient` objects, signals will be passed to only the most recently registered client.

Signals are categorized in the following way:

- *Shutdown*: Signals that cause the client process to terminate. A shutdown can be either clean or dirty (caused by a fault). Clean shutdowns are triggered by `SIGQUIT`, `SIGTERM`, and `SIGINT`. Dirty shutdowns are triggered by `SIGILL`, `SIGFPE`, `SIGTRAP`, `SIGSEGV`, `SIGEMT`, `SIGBUS`, and `SIGABRT`. The default behavior is to log a clean shutdown as an event and to log a dirty shutdown as a bug, which will also display a stack trace. In addition, the process is exited by calling either `EventDispatcher::exitDispatchLoop()` or the UNIX routine `exit()`.
- *Reset*: Signals that instruct the process to reset some state such as rereading a configuration file. Currently, this category contains only `SIGHUP`. The default behavior is to log the arrival of the signal as an event.
- *Alarm*: Signals that are generated whenever some timer expires. This category includes `SIGALRM` and `SIGPROF`. The default behavior is to invoke a dirty shut-down.
- *IO*: Signals that are generated when some I/O condition occurs such as the closing of a socket (`SIGPIPE`), or when the lock on a file is lost (`SIGLOST`). The default behavior is to log and otherwise ignore the signal.
- *Child Died*: Signals that indicate that a child process has just terminated, either normally or abnormally, or its state has changed. Currently, this category contains only `SIGCHLD`. The default behavior is to notify the `ChildProcessSet` singleton that a child process has died.
- *LogStream*: Signals that are interpreted as instructions to the D2D logging mechanism. `SIGUSR1` is used to break the log files and `SIGUSR2` is used to toggle the log filtration. Client processes can not override the handling of these signals.

5a.4.8.2 Controlling Signal Delivery

Some of our D2D processes link with middleware packages that will not function properly if a handler is installed for the `SIGCHLD` signal. Tcl/Tk is an example of such a package. Thus, client processes can instruct the signal handling software to keep its grubby little paws off of `SIGCHLD` by calling the static method `SignalCatcher::preventChildSignalHandling()`.

A few of our D2D processes such as the `acqserver` receive IPC messages from clients that do not link with our IPC library. As a result, these processes do not wait for events using the approaches described in Section 5a.4.2. Since our approach for handling asynchronous signals safely is contingent on the process using one of our event-waiting approaches, these processes must handle asynchronous signals at the risk of re-entering code that is not designed to be re-entered. If a client process calls the static method `SignalCatcher::useUnsafeSignalDelivery()`, then asynchronous signals will be handled as soon as they are delivered. Normally, their delivery is deferred until the client process is ready to handle them.

5a.4.8.3 Synchronous and Asynchronous Signals and Re-entrancy

Even in a single-threaded process, delivery of signals differs depending on whether the signal was generated synchronously or asynchronously because of the risk of re-entering non-re-entrant code.

Synchronous signals are the result of some error condition that occurs inside a process, and are delivered synchronously with respect to that error. For example, if a floating point calculation results in an overflow, a SIGFPE (floating point exception signal) is delivered to the process immediately following the instruction that resulted in the overflow. Our signal catching software handles the following synchronous signals: SIGILL, SIGTRAP, SIGABRT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, SIGPIPE, and SIGSYS.

Synchronous signals can be handled immediately without the danger of re-entering code since the signal was in response to an error in the code that it is interrupting.

Asynchronous signals are normally the result of an event that is external to the process, and are delivered whenever during the process execution such an event occurs. For example, when a user running a program types the interrupt character at the terminal (generally <ctrl-c>), a SIGINT (interrupt signal) is delivered to the process. Our signal catching software handles the following asynchronous signals: SIGHUP, SIGINT, SIGTERM, SIGALRM, SIGUSR1, SIGUSR2, SIGCHLD, SIGLOST, SIGPROF, and SIGQUIT.

If asynchronous signals are handled as soon as they arrive, it is quite possible to re-enter code that doesn't support re-entrancy, such as malloc() which may cause memory errors or a process crash. For example, suppose a depictable is busy allocating memory for a radar table and a SIGCHLD signal is delivered to our process, signaling that one of our child processes has died and needs to be restarted. While restarting the child, some memory is allocated and the process crashes due to a segmentation violation. Why? Malloc() accesses some shared global data structures. The second malloc() call during the signal handling was issued before the first malloc() call from the depictable had a chance to complete.

In order to avoid re-entrancy problems, our signal catching software delivers async signals using a strategy of deferred delivery. The vast majority of D2D processes alternate in a loop between waiting for events and handling new events. An event can be a timer expiring, an IPC message, a mouse button click, etc. As we saw with the previous example, async signals can be delivered while a process is busy handling an event. However, we defer the handling of that signal until the process has finished its event handling and is now waiting for a new event to arrive. The new event could be that deferred signal. We procrastinate on signal processing by using a pipe which is UNIX's implementation of a FIFO (first-in, first-out) queue. The details of using a pipe are covered in the next section.

5a.4.8.4 Single Thread Implementation

The SignalCatcher singleton object solicits signals from UNIX and passes them on to the registered SignalClient object by calling the appropriate method that matches the incoming signal.

Signal solicitation is done during static initialization time before the process executes its main() routine. The UNIX routine sigaction() is used to install one of two signal handling routines for each signal that we are handling. When one of those signals is sent to our processes, UNIX will call one of those routines, passing the signal number.

Also, during static initialization, the SignalCatcher will create a SignalPipe object that encapsulates a UNIX pipe (FIFO queue). During construction, the object creates the pipe and saves the two file descriptors for reading and writing to the pipe. It also registers with the EventDispatcher as a DescriptorEventClient so that the object will be notified when its pipe is ready for reading.

The signal handling routine for synchronous signals is called handleSignal(). This routine contains one large switch statement with cases for each supported signal. Each case will invoke the appropriate method for the currently registered SignalClient object. Thus, synchronous signals are not deferred; they are handled as soon as they arrive.

The signal handling routine for asynchronous signals is called receiveAsyncSignal(). If the client process instructed the SignalCatcher object not to defer signal delivery, then this routine will call handleSignal() which will handle the signal immediately. However, most processes will not do this since they do not want to risk a crash because of re-entrancy. Instead, receiveAsyncSignal() will tell the SignalPipe object to write the signal number to its pipe. When the process has finished its processing of other events, and is idle waiting for an event to arrive, selectDescriptorEvents() routine (covered in Section 5a.4.2.1) will determine that the pipe managed by the SignalPipe object is now ready for reading, and will call SignalPipe::handleEvent(). This method will read all the data in the pipe into an array of integers. Each cell in the array will contain an async signal that has been delivered but not handled. Looping through the array, handleSignal() is called for every cell in the array. Thus, the pipe is used to defer the delivery of async signals until a process is ready to handle them.

5a.4.8.5 Multiple Thread Implementation

In a multiple thread environment, synchronous signals are delivered to the thread that generates the signal. All other threads continue as if nothing happened. With our design, a process can have a single main thread that does most of the process work, and one or more threads that are waiting to write to a full socket buffer. For the main thread, we use sigaction() to install handleSignal() as the handler for all synchronous signals, just as we do in the single-threaded implementation. For sending threads, we rely on the default UNIX handling for synchronous signals except for SIGPIPE. The default UNIX behavior for SIGPIPE is to exit the process. That is not a good action for us. If another process closes its end of the connection, a SIGPIPE is generated; that's a signal for our process to close our socket but not to exit the process. The first thing that every sending thread does is to install a handler for SIGPIPE. This handler

doesn't do anything useful, but if the sending thread tells UNIX to ignore SIGPIPE, then the thread will not be able to detect when a connection is closed.

Handling asynchronous signals in a multi-threaded environment is fairly complicated since an async signal can be delivered to any one of the running threads. The decision of which thread gets the signal is dependent on the implementation of the DCE pthread() library. Also, this library does not allow clients to install handlers for async signals with sigaction() once the process becomes multi-threaded.

After consulting with HP, we developed the following approach for soliciting and delivering async signals in a multiple threaded environment. Right before the first sending thread is invoked, the SignalCatcher object will start a thread whose sole purpose is to listen for async signals. This async signal thread will block all other threads from receiving the supported set of asynchronous signals using the UNIX routine, sigprocmask(). It then calls another UNIX routine, sigwait(), that will block the thread's execution until one of the async signals arrives and returns the signal number.

At this point, if the client process instructed the SignalCatcher object not to defer signal delivery, then the thread will call handleSignal() which will handle the signal immediately. This is very dangerous since the signal handling will occur in a different thread but might possibly simultaneously access some of the same data structures that the main thread is accessing. Fortunately, only a few D2D processes require the immediate delivery of async signals.

More likely, once the async signal thread receives a signal from sigwait(), it uses the SignalPipe object to write that signal number to the pipe. The signal is then read by the main thread of execution without interrupting any of the main thread's processing. The signal is then handled and processed by the main thread by calling handleSignal(). This approach not only addresses the re-entrancy problem, but also the issue of two threads accessing the same data structures simultaneously.

5a.5 Conclusions and Future Direction

Clearly, interprocess communication in the D2D and data components of WFO-Advanced (FX-Advanced) is a complex issue. Even though we now have quite likely the clearest IPC implementation so far, it is still a daunting prospect to completely understand what's all involved. That is, of course, the entire reason for this document, which describes the IPC system in its entirety.

In this document, we have covered the requirements and history of IPC in WFO-Advanced, including descriptions of the two predecessors to the current thread-based implementation. We learned how well each product worked and what features made the previous systems

unacceptable. We also examined the current thread-based implementation and discussed its performance gains, its simplicity, its complexity, and its weaknesses.

By using an analogy of a business telephone system, we discovered what components existed in the thread-based implementation by correlating them with real-world objects. The analogy served as a focal point for the discussion of the implementation itself.

The majority of this document exposed the thread-based IPC implementation in great detail, enabling even the passive observer to understand and maintain the system. The discussion included addressing, event dispatching, message receipt, message transmission and metadata, synchronous and asynchronous sends, the use of threads, signal handling, and most of the classes and objects involved.

The WFO-Advanced IPC library continues to be open-ended and still allows for multiple concurrent transports, with threaded IPC coexisting seamlessly with any future transports. Any future work on the IPC library might include the addition of a new transport, but this is doubtful as the thread-based implementation satisfies all of our requirements quite well and in a highly advanced fashion. Future development on the thread-based implementation may evolve as vendors' thread libraries and kernel thread support improve. And although our use of threads doesn't involve concurrency, multiprocessor versions of UNIX may have some interesting (if minor) impacts on our library's performance. We look forward to better thread support as threaded programming becomes more and more the norm for software engineering.

CHAPTER 6 Data Acquisition

6.1 Introduction

The WFO-Advanced team has developed software to acquire data to support the forecasting functions of the WFO. Software has been developed to acquire data from or via the WSR-88D radar, the NOAAPORT data feed, and various local data sources unique to the WFO at the Denver site.

NOAAPORT, also known as the AWIPS Satellite Broadcast Network (SBN), provides a central data stream through a PRC-developed firmware product known as a Communications Processor (CP). The CP acquires data from the demodulated signal off a satellite broadcast feed, and concatenates and distributes data to several data server machines. WFO-Advanced sees the CP software as a "black box;" software development has been directed towards creating a process to acquire data from the CP (simulating a DWB-like connection) and assimilating these data into the WFO-Advanced data acquisition scheme (see Section 7.1.1, "System Design").

The WSR-88D interface is an extension of work performed over the last several years at FSL. FSL was able to develop certified Class I interfaces for radar ingest and display on the DARE II operational system at the Denver WSFO and the Pre-AWIPS system installed at the Norman, OK WSFO. Open-systems-compliant hardware was used for the WSR-88D interface. The ongoing efforts include a multiple-radar interface as well as a dial-out capability to acquire data from non-associated radars.

The third effort of the data acquisition system includes the ability to acquire data from local data sources. Although the Local Data Acquisition and Dissemination (LDAD) function is still being defined for AWIPS, WFO-Advanced includes a subsystem which is complemented by an effort within FSL which performs LDAD functions. This effort includes the use of PCs (outside of the AWIPS firewall) to acquire data. FTP is used to move the data between the PCs and the WFO-Advanced system. These data are currently being decoded for use by the graphics display. Efforts are underway to define a generic database for storage and retrieval of these data.

Future data acquisition and dissemination projects include the acquisition of ASOS data from a dedicated link, as well as asynchronous acquisition of ASOS data. NOAA Weather Wire dissemination will also be added to WFO-Advanced, as will the distribution of data over the AWIPS Communication Network (ACN).

6.2 SBN

The WFO-Advanced acquisition server process (acqserver) is the interface to the NOAAPORT Receive System (NRS) Communications Processor (CP). The NRS reads data streams from the AWIPS Satellite Broadcast Network (SBN) and sends them to a configured host data server over an ethernet based Local Area Network (LAN). The acqserver process runs on the data server host machine. The two data streams being sent from the CP are GOES_WEST (either EAST or WEST can be configured) imagery data and NWSTG (NWS Telecommunications Gateway, including text, graphics, observations, and model grids). These data streams are routed through the AWIPS Network Control Facility (NCF).

The GOES channel uses a T1 connection which has a data rate of 1.536 Mbps while the NWSTG data channel is 128 kbps. The NRS uses a HP 9000/747 computer running the HP-RT real-time operating system. HP-RT provides a UNIX-compatible environment including the UNIX shells, most commands, and IPC facilities (shared memory, semaphores, and message queues), as well as networking services such as FTP and NFS. The NRS software performs receive-only data acquisition, packet reassembly, product error control, SBN monitoring, and product transfer to configured host data servers. Its functionality and configuration is covered in the NRS User's Manual provided by PRC.

6.2.1 Design

The acqserver process acquires weather data from the CP across a direct socket connection to the acq_client process of the NRS. This ties into the NRS system's DWB transactions to set up the connection versus the standard configuration of using either NFS or FTP transfers (as documented in the NRS User's Manual). The acqserver process was adapted from a sample server implementation sent to us from PRC. The main added functionality was product identification and storage into our hierarchical directory structure. A product pattern-matching design was implemented. A configuration file is used to specify where a product matching a regular expression pattern needs to be stored in the data management directory hierarchy. Products are sent from the CP in 4000 byte packets which contain a header identifying the product type and category. The current types being used are NWSTG and GOES_WEST. The product categories are:

- TEXT
- GRAPHIC
- GRID
- POINT
- OTHER (text)
- IMAGE

The acqserver process reads the patterns configuration file (acq_patterns.txt) to determine where products that match a pattern are stored. The directories are relative to the start of the data acquisition directory structure (\$FXA_DATA). A product matching more than one pattern may be stored in multiple directories. Hard links to the different directories are used if possible (when they are within the same

file system) if multiple patterns match. Using hard links saves on disk I/O and disk space. Here are some sample lines from the configuration file:

```
GRAPHIC ^[PQ].* /ispan/graph
IMAGE ^T.* /goes
GRID ^[YZ].[A-MO-SU-Z].* /Grid/SBN/Raw
POINT .*IUPT(0[1-4]).*|.*IUPT40.* /ispan/bufr
POINT ^IUST[0-9][0-9].* /ispan/bufr/raob
TEXT ^SFUS41.* /ispan/binLightning
```

The 4000-byte packets sent by the CP are written (appended) to a time-stamped file as they arrive. The file name also contains the WMO header identification string. The acqserver process assumes the data are being sent in proper sequential order; no validation of the header sequence numbers is performed. After the last packet of a particular product arrives, the file is closed and a notification is sent to the Comms Router process.

The patterns configuration file can be changed at run time and will be re-read by the acqserver process on receipt of the HUP signal.

6.2.2 Data Flow

Products are sent from the CP to the host running the acqserver processes. The data are sent in 4000-byte packets plus a header. The header contains an indication of the first and last packet of every product sent (as well as sequence numbers for every intermediate packet.) The primary function added to the original PRC sample server implementation was storeProduct() which performs the pattern matching and appropriate product storage. After complete products are stored, a notification is sent to the CommsRouter process which through the DataControllers delivers the notification to the Decoders. The storeProduct function is called for each received packet and is passed the header, the buffer containing all or part of a product, and the time it was received. The first 11 characters of GOES products are used to perform pattern matching. For NWSTG products, the product identification begins at the 24th character of the first product buffer. This is part of the WMO header which follows the CCB header. If a match is found then a file name is generated which consists of the full directory name from the patterns configuration file, part of the product identification from the (WMO) header, and a time stamp with millisecond resolution. Sometimes the product identification from the WMO header contains embedded blanks. These are first stripped before generating the file name.

For GOES, NWSTG/TEXT, NWSTG/POINT, and NWSTG/OTHER products, the time stamps are from when the acqserver process started receiving the product. For NWSTG/GRID and NWSTG/GRAPHIC products, part of the time stamp is taken from the WMO header, specifically the day of month and the hour. The rest of the time stamp is from when the product is received by the acqserver process. The acqserver process uses the WFO-Advanced LogStream facility which allows logging messages of various severity levels to a log file or the screen. If VERBOSE level logging is turned on then each product read by the acqserver process from the CP is identified and those that matched a pattern are displayed with where they will be stored. If a received product matches multiple patterns then each file pathname (storage location) is displayed in the log. A few sanity checks are made for garbled headers or unknown product types. These are logged as well. All products are first completely written to a temporary file

location and then renamed to the proper directory. This prevents the decoders from reading a partial product file. Also, if an NWSTG/TEXT or NWSTG/OTHER product cannot be stored because the file system is full, for example, a notification is still sent down the pipeline to the CommsRouter process. The notifications for the NWSTG/TEXT and NWSTG/OTHER products involve sending the whole product contents rather than just a product identification string from the WMO header.

A configurable delay parameter (in milliseconds) is used by the acqserver process to delay the time between disk writes of GOES image product packets. This was introduced to avoid congesting the LAN where the CP and data servers are connected. The delay parameter is read from (and will be re-read on receipt of a HUP signal) the acqTimingParam.txt file.

6.2.3 Process Decomposition

The acqserver process is broken up into a parent process which waits for connect requests from a client process running on the CP and two child processes. The parent process spawns child processes to handle each data stream (GOES_WEST and NWSTG). The parent process starts by creating a listening socket and then waiting on an accept for the CP client process (acq_client) to connect to it over a "well known" port. The acq_client process on the CP sends a DWB transaction message requesting a direct connection to the acqserver for the purpose of sending data over one of the two data streams. When the acqserver process receives the connect request it forks (with no exec) a child "worker" process to handle reading that particular data stream from the CP. The parent acqserver process goes back to waiting for additional connect requests from the CP. After the CP starts sending over both data streams there will be three acqserver processes running on the data server: the parent process and the two child worker processes.

6.2.4 Tables and Files Accessed

The regular expression pattern/directory storage location file is named acq_patterns.txt. The timing delay parameter used to avoid network congestion when reading large GOES image files is stored in acqTimingParam.txt. Both configuration files are re-read when the child acqserver processes are sent a Hang Up (HUP) signal.

6.3 Radar Ingest

6.3.1 Class I interface

The radar ingest system for the WSR-88D class I interface is responsible for requesting data from one or more associated Radar Product Generation machines (RPGs), receiving data from the RPGs and making the data available to the D2D workstations. More specifically, this system is responsible for initiating and maintaining connections with multiple RPGs, forwarding data requests from the workstations to the RPGs, receiving and processing data and status messages from the RPGs, and forwarding status messages to the workstations.

The radar ingest system manages three types of requests: the Routine Product Set (RPS) list, the one-time request, and the alert request.

- The RPS list contains the products to be sent routinely to the workstation. The list is sent to the RPG on startup of the system, when the mode of the radar changes, or when a forecaster at the workstation modifies one of the adaptation RPS lists and sends that out to the RPG.
- The one-time request enables a forecaster at the workstation to request that a product be sent on a one-time basis only. The forecaster can also request that the product be sent for more than one volume scan. When the request has been honored, the request is cancelled.
- An alert request is generated at the workstation when the forecaster wants to be notified of an alert condition. When the threshold values specified in the request are exceeded, the RPG sends back an alert message. The forecaster can also request that a product be sent back along with the alert.

6.3.1.1 Design

The radar data ingest system is composed of two major processes: wfoApi and Radar Server (see Figure6.1). wfoApi makes and maintains connections with the RPGs using software libraries provided by Simpect to manage the transmission and receiving of messages from the RPG. There is one wfoApi process for each associated RPG. The RadarServer provides an interface between wfoApi and the workstations, routing data requests from the workstations to the appropriate wfoApi processes, and forwarding RPG status messages from wfoApi to the workstations. The RadarServer is also responsible for routing incoming data to the main data ingest system where data processing takes place. There is one RadarServer process in the radar data ingest system for handling interactions with associated RPGs.

6.3.1.2 Data Flow

On startup, wfoApi establishes a connection with the RadarServer process using inter-process communications. The Radar Server then sends out the "current" RPS list which wfoApi formats and sends on to the RPG.

When the RPG sends back a message, wfoApi receives it from the Freeway Call System Application Program Interface (CS-API) in 1024-byte packets. As each packet is received, wfoApi assembles the message based on the message length from the header. When the message is complete, wfoApi stores it in raw files and sends a notification to the Radar Server. If the message is a General Status Message, Alert Adaptation Parameters, an Alert Message, or a Product Request Response, the notification contains the actual message, and the Radar Server sends the message on to be processed for the workstation. If the message is a product, wfoApi sends a "data" notification to the Radar Server, which also sends a "data" notification to the Data Processing Subsystem. Within the Data Processing Subsystem is the RadarStorage process which moves the data from the "raw" data files to "product" directories. A high-level data flow is given in Figure6.1.

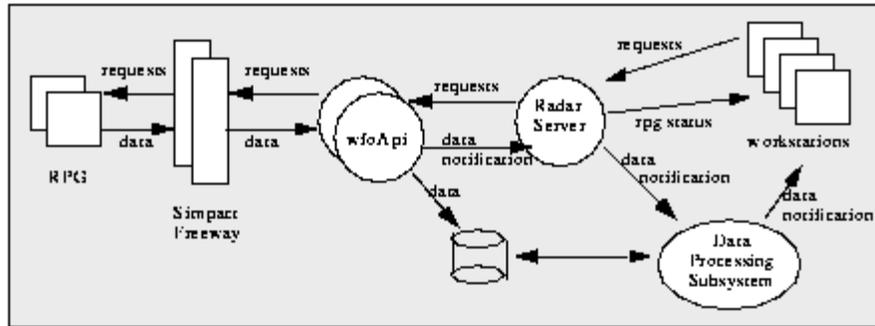


Figure 6.1 Radar Process Architecture, Data Ingest System (Class 1)

6.3.1.3 Process Decomposition

wfoApi

wfoApi is responsible for maintaining the connection with the RPG, and passing information between the RPG and the rest of the data ingest system. wfoApi has a direct interface with the Simpact's Freeway 1000 system using high-level CS-API library functions. The Freeway 1000 system provides serial communications services access to client processes using the Transport Control Protocol/Internet Protocol (TCP/IP) over an Ethernet local area network. In order to handle both normal and high priority data such as alerts, wfoApi establishes two permanent virtual circuits (PVCs). Once wfoApi has connected to the PVCs, data transfer can take place.

wfoApi has a very simple design. It responds asynchronously to external events, and uses tables and configuration files for initialization. wfoApi is started by a parent process called syncComms, the synchronous communications server. syncComms is patterned after Simpact's x25_manager described in the Freeway X.25/HDLC Configuration Guide. On startup, syncComms configures the links, and then passes control to wfoApi.

Some of the asynchronous events that occur are:

- **CS_ATTACH_SUCCESS:** occurs when wfoApi starts up and attaches to the specified communications server, and the attach has completed successfully. In this state, wfoApi binds the application to the X.25 protocol, establishing the connection.
- **CS_BIND_SUCCESS:** occurs when the bind completes successfully. When this happens, wfoApi sends a registration message to the Radar Server, followed by a "connection up" message.
- **CS_READ_COMPLETE:** occurs when the CS-API has completed reading a message from Simpact's Intelligent Communications Processor (ICP) board. The CS API sends the event to wfoApi. When wfoApi gets this event, it reads the data from the appropriate port on the Freeway. When all of the data are read in for a particular message, that message is stored and the Radar Server is notified. If the message is a General Status Message, Alert Adaptation Parameters, Product Request Response or Alert Message, wfoApi sends the message as a notification to the Radar Server.
- **CS_LOST_CONN:** occurs when either side of the link disconnects. wfoApi sends a "connection down" message to the Radar Server and then exits. On exit, wfoApi cleans up after itself and disconnects.

- **CS_WRITE_COMPLETE:** occurs when the CS-API detects that a write to an ICP port has completed. On receipt of this event, wfoApi goes back into a wait state.

wfoApi interacts with the RadarServer using IPC class libraries. The classes include:

- **SyncCommsClient:** handles receipt of the RPS list, One-time request, and Alert request from the RadarServer. Sends out the request.
- **RpgRequest:** swaps the bytes of the RPS list and the one-time request, and issues cs_write to write the request to the port.
- **RpgAlertRequest:** swaps the bytes of the alert request, and issues cs_write to write the request to the port.

RadarServer

The Radar Server is responsible for routing messages, and needs to do this efficiently. The server was designed to do very little I/O, and uses the WFO IPC classes in asynchronous mode, in an effort to achieve maximum throughput. The server was modeled after the CommsRouter process (part of the Data Processing Subsystem), which has similar requirements.

The Radar Server was designed using object-oriented analysis and design techniques described in Object-Oriented Modeling and Design by Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen. The major classes that resulted from the analysis, and the event traces that show interaction between classes are discussed below.

Some of the major classes are:

- **RadarServerClient:** a subclass of the RadarServer class. Responsible for processing all IPC messages the Radar Server receives. Maintains a list of wfoApi processes that have registered.
- **RadarStatus:** maintains current state of a radar RPG. Radar ID, radar name, connection status, current operational mode, current volume coverage pattern, current RPS list, current One-Time Request list, current Alert Request list.
- **ProductRequestList:** provides functions for creating a list of products to request from the RPG. Sends the list to a specified target. Writes the list to disk in editable ASCII format, reads list from disk. Send to multiple radars.
- **CurrentRpsList:** subclass of ProductRequestList. List of products that have been requested from the RPG. Replaces the current list with a default list from disk, or with a list received from a workstation. Formats the list for sending to an RPG, and sends the formatted list to a specified wfoApi process.
- **OneTimeReqList:** subclass of ProductRequestList. List of one-time requests that have been sent to the RPG.
- **AlertRequest:** handles the receipt of an Alert request from the workstation.

The Radar Server takes action in response to receiving an IPC message from either a wfoApi process or a workstation process. The messages received from the wfoApi processes and the subsequent actions are listed here:

- Registration message received: RadarServerClient adds RadarStatus object to list of registered wfoApi processes.
- Connection up message received: CurrentRpsList reads current RPS list from disk, and sends list to the wfoApi process. RadarStatus sends "connection up" notification to workstations.
- Connection down message received: RadarStatus removes the OneTimeReqList and sends a "connection down" notification to the workstation.
- General Status Message (GSM) received: If the operational mode has changed, the CurrentRpsList reads a default RPS list for the new mode from disk, and sends this list to the wfoApi process. RadarStatus forwards a GSM notification to the workstations.
- Product notification message received: RadarServerClient forwards the notification to the data processing subsystem.
- Product Request Response (PRR) message received: OneTimeReqList removes the request from the list. RadarServerClient sends PRR notification to workstations.
- Alert Adaptation Parameters (AAP) message received: RadarServerClient sends AAP notification to workstations.

The messages received from the workstations and subsequent actions are listed here:

- RPS list message received: CurrentRpsList stores list on disk, formats the list for the RPG, and sends list to appropriate wfoApi processes.
- One-time request message received: OneTimeReqList adds request to the list, formats the request for the RPG, and sends the request to the appropriate wfoApi processes.
- Alert request message received: AlertReqMsg writes request to file (for retrieval later if the system goes down), formats the request for the RPG, and sends the request to the appropriate wfoApi processes.

6.3.1.4 Tables and Files Accessed

The radar ingest system accesses static information from various tables and files. This feature provides flexibility and expansibility. Some of the primary tables and files are listed below under the process heading.

syncComms

- fw1000_pvc(n).setup: This is a "command" file read by the syncComms server to configure X.25 on the freeway ICP for port n.

wfoApi

- cs_config(n): This configuration file is read by wfoApi to provide specific run-time parameters to the CS-API. These parameters are used by the CS-API to associate X.25 circuits with low-level session names. This information is used to access lower level configuration files.
- freeway.config: contains the maximum block size read by wfoApi. It is currently 1024 but could go as high as 8192 bytes.
- portInfo.txt: contains all the ports dedicated to associated radars. It contains such information as port number, board number, radar name, and radar ID.

- productId.txt: contains the product IDs and product types of all possible radar products. An example of a product type is "Graphic."

No static tables or files are associated with the RadarServer process.

6.3.2 Dial-out Interface

The dial-out interface provides the forecaster at the workstation with the ability to send out a request for a product on a one-time basis to a non-associated RPG. When the radar ingest system receives the request, it dials the appropriate radar over one of the available dial ports. Once the connection is established, the request is sent out. The connection remains in effect until all requests to that radar are honored for that RPG. The interface can also handle multiple one-time requests going out to the same radar as long as the connection is up. One caveat here is that there are adaptation parameters on the RPG which limit the number of outstanding requests to five (5).

There can be multiple connections open for multiple radars, but only one connection for any single radar.

6.3.2.1 Design

The dial-out (Class II) system is very similar to the Class I system. There are two major processes, the Dial Radar and the Dial Server. DialRadar makes and maintains connections to the RPG using the CS-API library routines provided by Simpack to manage the transmission and receipt of messages to and from the RPG. DialServer is patterned after the Radar Server. It provides an interface between DialRadar and the workstation, routing data requests from the workstation to DialRadar, and forwarding RPG status messages from the DialRadar process to the workstation.

The association between DialRadar and DialServer differs from that of wfoApi and Radar Server, in that DialRadar is a child of DialServer.

6.3.2.2 Data Flow

On receipt of a one-time request from the workstation, DialServer starts up DialRadar for the specified radar, only if a dial connection to that radar is not yet established. DialRadar dials the RPG, sends out a one-time request, and waits for a message from either DialServer or the RPG. The receipt of messages from the RPG is handled in the same way as in the Class I system. As each message is received, DialRadar assembles the packets into a complete message based on the message length from the header.

One of the first messages received from the RPG is the product list which contains all products available on the RPG. The RPG also sends back a General Status Message (GSM) and either the requested product or a Product Request Response message (PRR). DialRadar stores the messages into "raw" data files and sends a notification to the Radar Server. The same procedure is followed as in the Class I interface. A high-level data flow is given in Figure 6.2.

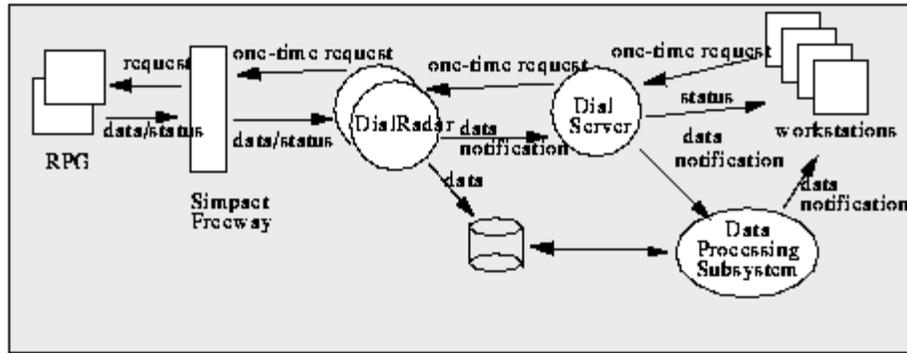


Figure6.2 Radar Process Architecture, Data Ingest System (Class II)

6.3.2.3 Process Decomposition

DialServer

DialServer is responsible for routing messages and starting up DialRadar. DialServer uses WFO-Advanced IPC classes in asynchronous mode in an effort to achieve multiple throughput. It is patterned after RadarServer and uses similar classes as defined below:

- DialServerClient: subclass of RadarReceiver class. It is responsible for processing all IPC messages the DialServer receives. It maintains a list of Dial Radar processes that have registered.
- DialRadarStatus: maintains current state of the RPG. The state information consists of radar ID, radar name, connection status, current one-time request list.
- One-timeRequestList: subclass of ProductRequestList. It maintains a list of the one-time requests that have been sent out to an RPG.
- Prr: decodes the Product Request Response message.

DialServer takes action in response to receiving an IPC message from either a DialRadar process or a workstation process. Messages received from DialRadar and the subsequent actions are listed here:

- Registration message received: DialServerClient adds DialRadarStatus object to the list of registered DialRadar processes.
- Connection Up message received: OneTimeRequestList extracts the repeat count from the request and adds the request to the one-time request list. It then sends the request to the Dial Radar process. DialRadarStatus sends a "connection up" message to the workstations.
- Connection Down message received: DialRadarStatus calls on OneTimeRequestList to remove all requests from the list, and sends a "connection down" message to the workstations. DialRadarStatus also removes DialRadarStatus object for DialRadar from the list of registered DialRadar processes.
- Product Request Response message received: OneTimeRequestList checks the PRR to see if the product will be available in the next volume scan. If not, then the product is removed from the one-time request list. If the list is empty, DialRadarStatus sends a "disconnect" message to the Dial Radar process. DialRadarStatus sends a PRR notification to the workstations.
- Product Notification message received: OneTimeRequestList decrements the repeat count for that request. If the repeat count is zero, it removes the request from the list. If the list is empty

then DialRadarStatus sends a "disconnect" message to DialRadar. If the repeat count is not zero it updates the repeat count. DialRadarStatus sends a "data" notification to the workstations.

- General Status Message or Product List message received: DialRadarStatus sends the appropriate notification to the workstations.

The messages received from the workstations and the subsequent actions taken are listed here:

- One-time request message received: DialServerClient checks whether the requested radar is in the list of registered DialRadar processes. If not, DialServerClient forks a child process, DialRadar, for that radar. Once the connection is established, DialRadarStatus calls on the OneTimeRequestList class to add the request to the one-time request list and to extract the repeat count. DialRadarStatus then sends the request to DialRadar.

DialRadar

DialRadar is responsible for establishing and maintaining a dial-out connection with the RPG, and for passing information between the RPG and the rest of the radar data ingest system. As in the case of wfoApi, DialRadar establishes a direct interface to the Freeway as discussed in Section 6.3.1.3. Unlike a dedicated line, the dial-out connection is temporary and remains in effect only as long as there are outstanding requests.

On startup, DialRadar does the following:

- sends a registration message to the Dial Server;
- gets the telephone number and password from the configuration file using the radar name passed to it by the Dial Server;
- gets an available port and tags that port as "busy;"
- runs the x25_manager as a child process to configure the link as HDLC;
- dials the RPG. On successful connection, DialRadar sends a "connection up" message to the Dial Server, and runs the X.25 manager as a child process to reconfigure the link as a PVC for data transfer. If connection is unsuccessful, DialRadar sends a "connection down" message to the Dial Server.

The messages received by DialRadar from the RPG and the Dial Server, and the subsequent actions, are listed below:

- One-time request received from the DialServer: DialRadar formats the one-time request, sends it out and waits for a message from either the RPG or the Dial Server.
- ProductList message, GSM, or PRR from the RPG: DialRadar stores the product in the "raw" data file and sends the message to DialServer.
- Product message from the RPG: DialRadar stores the product in the "raw" data file and sends a product notification to DialServer.
- Disconnect message from DialServer: DialRadar disconnects from the RPG, tags the port as "free" and sends a "connection down" message to DialServer.

All DialRadar actions are asynchronous. As an event occurs, the event is processed and DialRadar goes into a "wait" state. Some of the asynchronous events are listed below:

- **CS_ATTACH_SUCCESS:** The CS-API sends this event to the client, DialRadar, on completion of a successful `cs_attach`. `cs_attach` attaches the client to the specified communications server.
- **CS_BIND_SUCCESS:** DialRadar issued a `cs_bind` which completed successfully. `cs_bind` binds the client to the HDLC or X.25 protocol.
- **CS_CONNECT_SUCCESS.** DialRadar issued a `cs_connect` which completed successfully. `cs_connect` enables the client to establish an HDLC connection to the remote modem (DTE).
- **CS_WRITE_COMPLETE:** DialRadar issues `cs_writes` to send the password and telephone number to the modem, and to send out information to the RPG. On successful completion, the CS_API sends this event to the client process.

6.3.2.4 Tables and Files Accessed

DialRadar

- `fw1000_hdlc.setup(n)`: This is a "command" file read by the `x25_manager` to configure HDLC on the Freeway ICP for port `n`.
- `fw1000_pvc(n).setup`: This is a "command" file read by the `x25_manager` to configure X.25 on the Freeway ICP for port `n`.
- `cs_config(n)`: This configuration file is read by DialRadar to provide specific run-time parameters to the CS-API. These parameters are used by the CS-API to associate X.25 circuits with low-level session names. This information is used to access lower level configuration files.
- `freeway.config`: This file contains the maximum block size read by DialRadar. It is currently 1024 but could go as high as 8192 bytes.
- `dialPorts.txt`: contains information on all the dial ports available on all ICPs. It contains such information as port number and board number.
- `dialRadars.txt`: contains all the radars containing dial ports. The information consists of radar name, radar ID, telephone number, and password.
- `productId.txt`: contains the product IDs and product types of all possible radar products. An example of a product type is "Graphic."

6.4 LDAD Ingest

6.4.1 Introduction

Automation of field offices' interactions with local data observation systems, spotter networks, cooperative observers, and members of the local decision-making community is recognized as vital to the success of the NWS modernization. This automation is required (1) to acquire observational data to complement the basic federal weather observing systems and (2) to disseminate warning information to key decision makers in local and state communities.(1)

6.4.2 Concept

The proposed LDAD functionality for WFO-Advanced includes a suite of PC hardware outside the AWIPS network at the WFO. This hardware acquires and disseminates data to the general public while preserving the security efforts necessary for the AWIPS network.

LDAD has several intended users in the WFO network. The forecaster depends on these data for reliable forecasting and verification within the local scale. The local analysis extensively uses the local data for results. Application programs (such as the WHFS applications) use these data for further analysis.

In order to accommodate these users, FSL is investigating a generic database for storage of these data sets. Currently, the data are stored in flat "plot" files, unusable by any users but the WFO-Advanced D2D display. The concept is to have a generic netCDF file in which any local data point can be stored. This file would have enough self-describing information for identification by any end-user.

Currently, the Denver system includes two local data sets: a set of surface sensors maintained by the Colorado Department of Transportation (CDOT), and a set of sensors gathering hydrological information maintained by the Denver Urban Drainage and Flood Control District (UDFCD).

6.4.3 CDOT

The Colorado Department of Transportation (CDOT), with Surface Systems, Inc. (SSI), operates a network of approximately 38 Remote Sensing Units (RSUs). Each RSU collects data from 5 sensors ◆ 1 surface and four sub-surface ◆ that measure such phenomena as air temperature, precipitation, dew point, etc. The CDOT polls the RSUs every 15 minutes and transmits the data to FSL's Community Server.

6.4.3.1 Design

Every fifteen minutes, a process running on the WFO-Advanced data server copies the current CDOT file via FTP from the Community Server to WFO-Advanced, where the file is stored for later decoding. The decoded data are stored to disk in both flat file and netCDF formats. The flat file is later accessed by the workstation and translated on the fly for display on the workstation.

6.4.3.2 Data Flow

The CDOT data are copied to the Community Server from the CDOT every 15 minutes. From here the data are copied to WFO-Advanced and stored in \$FXA_DATA/async/cdot. A notification is sent to the Comms Router of data arrival. The Comms Router sends on a notification to the Data Controller which then sends on the data to the CdotDecoder. The decoder translates the data from ingest format into meteorologically meaningful values and stores the decoded data in both flat file and netCDF file formats.

6.4.3.3 Process Decomposition

- ftpCdot: started via a cron job which runs every fifteen minutes starting at 5 minutes after the hour. ftpCdot copies the data from the Community Server, stores the data to disk, and notifies the Comms Router.
- Comms Router: sends a notification to the Data controller.
- Data Controller: passes on the notification to the CdotDecoder.
- CdotDecoder: a child process started by the DataController. On receipt of a notification from the Data Controller, the decoder reads the data file from \$FXA_DATA/async/cdot and decodes the data. It reads its station information file and generates a report that is stored both in flat file and netCDF file formats. Both of these files are hourly files based on observation time.

6.4.3.4 Tables and Files Accessed

The decoder reads station information from file \$FXA_HOME/data/cdotStationInfo.txt. This file contains the station ID, latitude, longitude, and station description.

6.4.4 ALERT Decoder

The ALERT data are hydrometeorological observations from remote data collection platforms (DCPs). These DCPs are connected to a base station, at the Denver Urban Drainage and Flood Control District (UDFCD), via radio communications.

The ALERT data are copied to WFO-Advanced by an FTP cron job from the Community Server. This scheduled copy is synchronized with the receipt of data from the Denver UDFCD to the Community Server. The Community Server receives these data by a land line link (modem), and after a short delay the data are relayed to WFO-Advanced.

6.4.4.1 Design

The AlertDecoder is started up by a DataController process. When started, the AlertDecoder registers a pattern string and a callback function with the DataController. When the DataController receives a message containing the AlertDecoder pattern string, this message is passed to the callback function, instantiating the AlertDecoder process.

Upon successful completion and transfer of ALERT data files from the Community Server, the FTP process sends an ALERT pattern notification to the DataController. (See Section 7.6.2.2 for more information on data arrival notification messages.) This IPC notification causes the AlertDecoder to read each of the received data files, decode their content, then store the data. These decoded observations are also "serialized" encoded in a platform-independent manner and sent via an IPC message to the SHEF encoder process, where the message is "quantized" decoded from the "serialized" message and prepared for ingest into WHFS's hydrologic data base.

6.4.4.2 Data Flow

The ALERT data received from the Community Server are parceled to three files: meteorological non-precipitation weather observations, precipitation observations, and hydrological observations. These

files are stored in the \$FXA_DATA/async/alert directory while awaiting decoding. As each of these data files is processed, its data contents are stored in LDAD plot files (a display-ready format used by WFO-Advanced) and ALERT netCDF data file (containing a complete data record). When each file is successfully decoded it is unlinked. If there is a failure in the decoding process, the file is moved to the \$FXA_DATA/async/alert/badAlert directory to await evaluation of the decoder failure.

6.4.4.3 Process Decomposition

A cron job FTPs data periodically to WFO-Advanced from the Community Server and sends a data arrival notification message to a DataController. When the AlertDecoder was started by a DataController, the decoder registered a pattern string to have specific data sets delivered to it. When this data arrival message pattern string is received by the DataController, the message is forwarded to the decoder. Upon receipt of the forwarded message, the AlertDecoder reads the data files in \$FXA_DATA/async/alert and decodes them, storing their data in LDAD plotfiles and netCDF data files. The AlertDecoder also extracts the relevant hydrologically significant data from the decoded data and forwards this using an IPC data message to the SHEF encoder, where these data are SHEF encoded for ingest into the WHFS hydrologic database.

6.4.4.4 Tables and Files Accessed

The names of the un-decoded ALERT observations received from the Community Server are alert_wx.dat (the meteorological non-precipitation data observations), alert_r5.dat (the incremental precipitation observations), and alert_wl.dat (the hydrologic observations). All are stored in \$FXA_DATA/async/alert.

The AlertDecoder uses the AlertStationInfo.txt file to provide the necessary location (latitude, longitude, and elevation) data and station name details that are stored in the LDAD plotfile and netCDF data files along with the decoded observations.

The decoded data are stored in the \$FXA_DATA/point/localdata directory where the plotfiles are named by the date and hour of data that are contained in the file. Similarly-named netCDF files are stored in \$FXA_DATA/point/alert.

Footnotes

(1)

User Language Requirements Document (ULRD) for the Local Data Acquisition and Dissemination (LDAD) Function of AWIPS, May 1994, NWS Internal Document.

CHAPTER 7 Data Management

7.1 Overview

The Data Management system is composed of a number of software modules that are responsible for processing and storing data as they arrive from external sources, and for locating and retrieving data for clients such as the D2D workstation and LAPS. The Data Management modules contain C++ classes and C functions that decode data arriving in an encoded transmission format, store the decoded data on disk, locate and retrieve data from disk, and route data between processes. These modules also include netCDF CDL definitions that specify data storage formats.

7.1.1 System Design

7.1.1.1 Data Processing and Storage

The data processing and storage tasks have been divided among a number of processes: the routers, the data controllers, and the decoders. The router processes receive messages from the Data Acquisition system, and send them immediately to the data controllers. The controllers are responsible for sending the messages to the appropriate decoder processes, and also for queuing the messages until the decoders are ready for it. See Figure 7.1.

Some of the messages contain data (specifically text data), and some of them contain "notifications" of data that have been written to disk. The Data Acquisition processes write most of the incoming data to disk, and send messages to the routers consisting of the file name and a text string containing a unique product identifier. For the remainder of this document, the term "data message" will be used to refer to messages that contain either data or a data notification.

This design was adopted to ensure that none of the data arriving from external sources are lost. Some of the processing tasks (especially decoding) are time consuming, and processes performing these tasks are unable to keep up with incoming data rates. Neither the routers nor the controllers perform any decoding or other data processing, so they are able to keep up with the rate of incoming data. Also, the controllers cache the incoming data and notifications until the slower decoders are ready for them. Since most of the data already reside on disk, very little will be lost if any of the processes exit.

This design also makes it easy to spread the processing over several hosts if the system becomes too large to run on a single machine. The InterProcess Communication (IPC) software

that sends messages between processes can be used for both local and remote communication. Data can be received on one machine and routed to a decoder on another machine if needed.

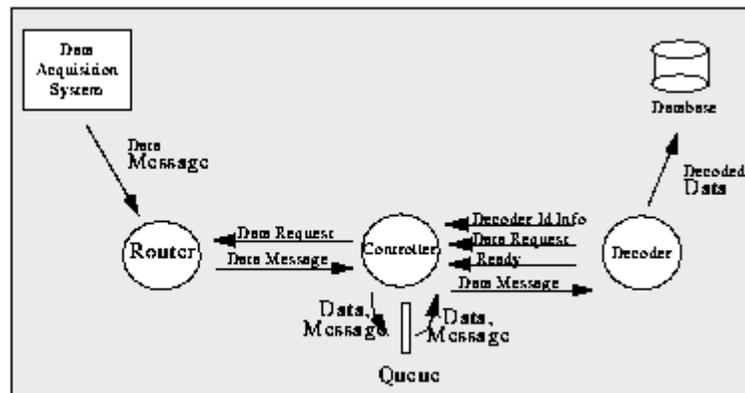


Figure 7.1 Data Processing and Storage System Design

7.1.2 Data Retrieval

7.2 Data Routing

The data routing software is responsible for moving data and data notifications among processes. This part of the system uses a modified client/server methodology: client processes send requests for data to server processes, and the servers send data or notifications in response. The client/server paradigm breaks down somewhat, though, because the servers send data continuously to the clients, without waiting for requests each time.

The data controllers are clients of the routers, and servers for the decoders. The decoders initiate the client/server communications among these processes by sending requests for data to the appropriate controllers, which in turn forward the requests to the routers. The incoming data messages flow through the system in reverse order: from routers to controllers to decoders.

The data request sent from a decoder to a controller, thence to a router, consists of a regular expression that identifies the data being requested and the IPC address of the requesting process. When the routers and controllers receive a data message, they match a unique data ID contained in the message with the patterns contained in the regular expressions of the data requests. If there is a match, the message is forwarded to the corresponding IPC address.

This design has made adding new data types very easy, because the data controllers and routers don't have to be modified. All the information they need to route the messages they

receive is contained in the data requests and data messages. The controllers and routers don't need to have any knowledge of existing data types.

7.2.1 Router Processes

7.2.1.1 Description

The router processes are responsible for transporting data and data notifications from the Data Acquisition processes to the data controller processes. The main function of the routers is message passing; they do not do any data processing. The routers provide a known location where the Data Acquisition processes can send data messages. No new routers are added to the system when additional data types are added.

There are two routers in the data processing system: the Grid Router, which handles all of the gridded data messages, and the Comms Router, which handles everything else. This design was adopted because grids account for a large percentage of the incoming data, and having a separate path for them lessens the impact on the rest of the system. Both routers are instances of the same executable.

The router processes are started automatically at system reboot, or can be started manually by the ingest restart program. The routers run continuously, exiting only when killed by the restart program.

7.2.1.2 Main objects

This section describes the main objects involved in routing data messages from the Data Acquisition processes to the data controllers. Some of the objects are created by the routers, and some are created by the processes sending messages to the routers.

- **CommsReceiver:** CommsReceiver objects are responsible for receiving messages sent to the router processes. Each router has one CommsReceiver object, which handles data request messages from the data controllers, and data messages from the Data Acquisition system. When the CommsReceiver object receives a message, it invokes a specific member function to take an appropriate action.

The CommsReceiver class is derived from the Receiver class, which is part of the IPC module. An object of the Receiver base class or its derived classes can register with the IPC objects to receive messages that travel through the IPC system. Objects of the CommsReceiver class are created automatically at run time, enabling the routers to receive messages from the IPC layers.

- **CommsDataMsg:** The CommsDataMsg class handles the data requests sent to the routers by the controllers. This class maintains a list of requests in a file static variable. When the CommsReceiver object receives a data request, it uses a static-CommsDataMsg member function to add the request to the list.

Objects of the CommsDataMsg class handle the data messages sent to the routers by the Data Acquisition processes. Each time the router receives a data message, the CommsReceiver creates a CommsDataMsg object which uses the static list of requests to decide where to send the message.

- CommsRoutingEntry: This class contains the information needed for routing data messages to a client: the data request pattern and the IPC address of the client. The CommsDataMsg request list is a list of CommsRoutingEntry objects, which are created by the CommsDataMsg member function that adds data requests to the list.
- DataSourceMsg: The Data Acquisition processes create DataSourceMsg objects to send data messages to the routers. The data members of the class contain the data or notification, and several fields of metadata.

7.3 Data Controller Processes

7.3.1 Description

The data controllers are responsible for creating the decoder processes (as child processes), restarting them if they exit, routing appropriate data messages to them, and queuing incoming data messages until the decoder is ready.

There are currently seven data controllers in the system, each of which creates one or more decoder processes:

- Satellite Controller: creates the Satdecoder process;
- Grib Controller: creates the GribDecoder process;
- Text Controllers: there are two of these. One creates the RaobDecoder, profilerDecoder and MetarDecoder processes; the other creates the shefEncoder, AlertDecoder, CdotDecoder and binLightningDecoder processes;
- Radar Controller: creates the RadarStorage process;
- TextDB Controller: there are two of these. One creates the CollDBDecoder, and the other creates the StdDBDecoder and RadarTextDecoder processes.

The reason for having so many controllers relates to their data queuing function. Each controller creates a queue for each decoder it manages. As data messages are stored in the queue, the size of the process grows. It is important for controller process size to remain under the maximum allowed by the operating system. Another reason has to do with the way messages are routed to the decoders. For example, the Coll and Std text decoders were originally children of the same controller, but we lost messages that got "grabbed" by the wrong decoder.

The controller processes are started automatically at system reboot, or can be started by the ingest restart program. The controllers run continuously, exiting only when they are killed by the restart program. All data controllers are instances of the same executable.

7.3.2 Interactions With Decoders

When a decoder first starts up, it sends a message containing its IPC address and process number to the parent controller. The decoder follows this with a data request message, and then with a third message indicating that it is ready to receive data messages. In response, the controller stores the decoder ID information and the data request, and sends a data message if it has one. The controller then waits to receive another ready message from the decoder before sending any more data messages. The controller queues any messages it receives until the decoder sends another ready message.

7.3.3 Main Objects

This section describes the main objects involved in routing data request messages from the decoders, and in routing the data messages back to the decoders that issued the requests.

- **DataCaptReceiver:** DataCaptReceiver objects are responsible for receiving messages the decoders send to the data controllers. Each controller creates one Data-CaptReceiver object. This object invokes specific member functions to handle the data request messages and ready messages received from the decoders.

The DataCaptReceiver class is derived from the Receiver class, which is part of the IPC module. An object of the Receiver base class or its derived classes can register with the IPC objects to receive messages that travel through the IPC system. Objects of the DataCaptReceiver class are created automatically at run time, enabling the data controllers to receive messages from the IPC layers.

- **CommsReceiver:** The data controllers use CommsReceiver objects for receiving data messages from the routers. The CommsReceiver objects are created automatically at run time. When a data controller sends a data request to a router, an object of the DataRoutingManager class (described below) registers a callback function with the CommsReceiver object. When the CommsReceiver receives a data message from a router, it invokes this callback function. See the description of the DataRoutingManager object for details.

Using callback functions in this situation allows more than one type of client to receive data messages using the CommsReceiver. The client can process the data messages any way it wants by registering its own handling function.

- **DataRoutingManager:** The DataRoutingManager class is an example of a "mediator" class as defined in Design Patterns by Gamma, Helm, Johnson, and Vlissides. This class provides an interface for using objects of the ChildProcessSet class to create and restart the decoder processes, and the DataRoutingEntry class to route data messages to the decoders.

The data controller main program creates a DataRoutingManager object when it first starts up, and invokes a member function to start the decoder processes. When the data controller receives a data request message from a decoder, the DataRoutingManager object creates a DataRoutingEntry object to contain the request information, and adds this to a list. The DataRoutingManager object then sends the request message to the appropriate router process, and also registers a callback function with the CommsReceiver object. When a data message arrives, the CommsReceiver object invokes the callback

function, which routes the data message to the appropriate decoder. This function searches the request list for decoders that have requested those data, and forwards the data message.

When a decoder exits, the `DataRoutingManager` removes the child process entry from the `ChildProcessSet`, and also removes the corresponding `DataRoutingEntry` object from the request list.

- `DataRoutingEntry`: This class is derived from the `CommsRoutingEntry` class, which contains information needed for routing data messages to a client: the data request pattern and the IPC address of the client. `DataRoutingEntry` objects each also contain an object of the `QueuePtr` class, where incoming data messages are stored until the appropriate client (decoder) is ready. The `DataRoutingManager` object creates one `DataRoutingEntry` object for each data request it receives from a decoder, and adds the object to its data request list.
- `Q_Data`: Objects of this class contain the data messages that are queued until the appropriate decoder is ready. Before sending incoming data messages to a decoder, the `DataRoutingEntry` object containing the decoder's request checks to see if the decoder is ready. If not, the `DataRoutingEntry` object creates an object of the `Q_Data` class and adds it to the queue.
- `CommsReqMsg`: The data controllers create objects of this class to send requests for data to the routers. The request contains a regular expression that defines the specific data to send.
- `DataReqMsg`: The decoders use this class to send requests for data to the controllers. The data request contains a regular expression that defines the specific data to send.
- `DataMsg`: The `DataMsg` class is responsible for sending data and notifications to the decoders. The `DataRoutingManager` creates objects of the `DataMsg` class when the data controller receives a data message from one of the router processes.
- `ReadyMsg`: The decoders use this class to notify their parent controllers that they are ready to receive another data message.

Message classes:

The message classes (`CommsReqMsg`, `DataReqMsg`, `DataMsg`, and `ReadyMsg`) provide the interface to the IPC layers for sending messages. The `CommsReqMsg` is derived from the `CommsMessage` class, which defines the types of messages the `CommsReceiver` will recognize. The remaining three message classes are derived from the `DataCaptMessage` class, which defines the types of messages that will be recognized by the `DataCaptReceiver`. The `CommsMessage` and `DataCaptMessage` classes are derived from the `Message` class, which provides public send functions that access the lower level IPC layers.

7.4 Decoding

7.4.1 METAR

The surface weather observations and reports (METAR) arrive in text format over the Satellite Broadcast Network (SBN). The METAR decoder used by WFO-Advanced is a revised version of the NWS Office of

Systems Operations (OSO) decoder. For detailed information on the specific format of a report please refer to the Federal Meteorological Handbook Number 1 document, \`Surface Weather Observations and Reports.\`

The directory structure for METAR data begins with /data/fxa/point/metar. The following subdirectories exist below that:

- Raw (/data/fxa/point/metar/Raw) is where the reports are written before decoding.
- Bad is where the reports that were not correctly decoded are moved.
- netcdf is where the netCDF format storage files are kept.
- plot is where the binary plot storage files are kept.

The raw data arrive as one singular report or as a collective report. The collective report contains data from several stations. These data are ingested and stored in the Raw directory. As each report is written to disk, a notification is sent to the Comms Router which then notifies the Text Controller, which then pings the METAR decoder that there is data to be processed. The decoder will process every file in the directory, not just the one for which it received a notification. If the report is successfully decoded and stored, then the file is deleted. If an error occurred, then the file is moved to the Bad directory. The decoder then moves on to the next file in the directory. When it is finished processing all the files in the directory, the decoder waits for the next notification to arrive.

Decoded METAR data are stored in netCDF and plot files. For specific details on which parameters are stored, refer to Section 7.5.4.2 of this document.

7.4.2 Satellite

The Satellite decoder decodes raw GOES satellite data from on five scales: east conus, west conus, northern hemisphere, super national and Conus C. On receipt of a notification from the satellite controller, the decoder reads the directory containing the raw files. If the file is complete, based on information from the SatSizes.txt file, it decodes the file line by line. If the number of pixels on a line is less than the maximum number (nx), the remainder of the line is zeroed out. The same applies to the number of lines in the data which is checked against the value, ny. This is done to eliminate white space on the image. When all lines of the file are decoded the data are stored in NetCDF.

Further processing is done on GOES satellite imagery to create derived satellite products. These products are generated based on keys which indicate whether a "clean" satellite image is to be generated, clipped or remapped. These derived satellite products are also stored in NetCDF.

7.4.3 Models

Several models covering different areas are ingested by WFO-Advanced over the Satellite Broadcast Network (SBN). Some local scale models are also available by other means. The SBN models include:

- AVN (Aviation forecast model)
- Eta (Eta forecast model)

- MesoEta (Mesoscale Eta forecast model)
- MRF (Medium Range Forecast model)
- NGM (Nested Grid Model)
- RUC (Rapid Update Cycle)

The local scale models include:

- LAPS (Local Analysis and Prediction System)
- MAPS (Mesoscale Analysis and Prediction System)

The SBN models cover the following grid domains

- 201 (Hemispheric - polar stereographic)
- 202 (National - CONUS - polar stereographic)
- 211 (Regional - CONUS - Lambert Conformal)
- 212 (Regional - CONUS - double resolution - Lambert Conformal)
- 213 (National - CONUS - double resolution - polar stereographic)
- 215 (contiguous United States - quadruple resolution - Lambert Conformal)

For more detailed information on grid points, grid corners, latitudes, longitudes, etc., refer to the NCEP GRIB document.

The SBN models currently used by WFO-Advanced and their grid domains:

- AVN (201, 202, 211, 213)
- ETA (211)
- MesoEta (212, 215)
- MRF (201, 202, 213)
- NGM (202, 211, 213)
- RUC (211)

7.4.3.1 GRIB

GRIB (GRIdded Binary) format is a bit-oriented method of compacting information about gridded data for faster transmission over high speed telecommunications lines. For a complete description of GRIB format, please refer to the NCEP document *The WMO Format for the Storage of Weather Product Information and the Exchange of Weather Product Messages in Gridded Binary Form*. A copy of this document can be obtained by anonymous FTP at [nic.fb4.noaa.gov](ftp://nic.fb4.noaa.gov).

7.4.3.2 Grids

The directory structure for grids begins with `/data/fxa/Grid`. For decoding purposes the hierarchy below this level is as follows:

Data Source (SBN or FSL)

- `/data/fxa/Grid/SBN`
- `/data/fxa/Grid/FSL`

The next level down is the same for all data sources:

Raw is where GRIB format grids are written to by the data source before decoding.

- /data/fxa/Grid/"DATA-SOURCE"/Raw

Bad is where grids that were not properly decoded are moved

- /data/fxa/Grid/"DATA-SOURCE"/Bad

CDL is where all the CDL files for that data source are kept

- /data/fxa/Grid/"DATA-SOURCE"/CDL

netCDF contains subdirectories of grids decoded and stored in netCDF format

- /data/fxa/Grid/"DATA-SOURCE"/netCDF

Many different grids are currently received over the SBN and stored on disk in GRIB format in the Raw directory. All the grids from the SBN models mentioned in above are written into this directory. As each grid is written to disk, a notification of its arrival is sent to the Grid Router, which then routes it to the Grid Controller, which pings the decoder to go look in the Raw directory for data. When the decoder goes to the Raw directory, it processes every file in the directory at that time, not just the one for which it was notified. Obviously, at some point there will be several notifications sent to the decoder after the files have already been decoded. These are just ignored. The advantage of decoding every file, instead of just the one notified, is really seen when something goes wrong with the data ingest. If the IPC goes down, rather than having to send many phony notifications of data arrival, just one ping will get the decoder going and processing the entire directory.

For each file, the decoder "de-grips" the data, identifies the grid, and then stores it in netCDF format. Assuming there are no problems with any of the processing of that grid, the decoder deletes the file from the Raw directory before moving on to the next file. If there are any problems, errors are logged and the file is moved to the Bad directory so that it will not continuously recreate the same error for the decoder.

7.4.4 RAOB

Upper air data (Radiosonde Observations, commonly referred to as 'RAOB') are currently received through the SBN and stored on disk as ASCII text in the /data/fxa/ispan/text/raob directory. The nominal observation time for the majority of RAOBs is 0000Z and 1200Z, although there are stations that report at other times. In addition, some stations, called PIBALs (for PIloted BALloon) report only wind data and include no temperature data in the report. The decoder is set up to search text files with the header pattern [.*U[GJKMS].*|.*U[FGIQ]CN.*]. So for example, a RAOB file with the character pattern of UGUF in the file header will be decoded. As data are received, the RaobDecoder is notified via the Text Controller to decode the message and store it as a metadata plotfile. After the text file is decoded and

stored, the decoder deletes the text file from the directory. See Section 7.5.4.1 for further information on storage formats.

For specific details on how to decipher RAOB data, see WMO Manual 386. RAOB data basically comprises various data formats consisting of mandatory level data and significant level data. The mandatory part of the RAOB dataset, prefaced with the four letter designation of 'TTAA,' contains the WMO station number, date and time of the RAOB, temperature (degrees Celsius), dewpoint depression (degrees Celsius), and wind direction and speed (knots), reported at the surface and at each of the 'mandatory' pressure levels of 1000, 925, 850, 700, 500, 400, 300, 250, 200, 150, and 100 hPa, along with each level's corresponding geopotential height (meters). Also reported are the tropopause data (denoted with the '88' designator) and the maximum wind in the sounding (denoted with either the '66' or '77' designator).

Significant level data are further divided into temperatures/dewpoint depressions ('TTBB') and winds ('PPBB') from the surface up to 100 hPa. Significant level data above 100 hPa are designated with 'TTCC,' 'TTDD,' 'PPCC,' and 'PPDD' notations as reported. Significant level temperatures and dewpoint depressions are reported at pressure levels. Data for each significant level are prefaced by a two digit number (00, 11, 22, 33, etc.) followed by that level's observed pressure.

Significant level winds are encoded differently from the significant temperatures. Each wind group is prefaced with a '9' designator followed by three height levels (thousands of feet). A group encoded as '90078' means there are reported winds at the surface, 7000 ft., and 8000 feet. Similarly, a group encoded as '92567' indicates winds observed at 25,000, 26,000, and 27,000 feet. Following each group of heights are winds reported as direction degrees and knots. For example, '90078 20005 32512 33512' would be decoded as surface winds 200 degrees at 5 knots, 7000 ft. level winds 325 degrees at 12 knots, and so on. A height group encoded as '909//' means only one wind report follows, in this case at 9000 feet.

Since the TTAA, TTBB, PPBB, TTCC, etc. all arrive at different times, it is the job of the decoder to handle the time difference between these products and merge the data into one, cohesive data report for each reporting station and observation time. First the data must get sorted through RaobDecoder.C, from the surface up, before they can be merged.

RaobDecoder.C::mergeRAOB_Info handles the merging of the RAOB data. Data are stored as a SeqOf<RAOB_Info> object, and decoded as a Raob_t struct (getRAOB_InfoRpts) as defined in the RAOBdecoder.H header file. As data become available for decoding, RaobDecoder reads the text file and decodes the message as either mandatory or significant, storing the data in the appropriate fields as a plotfile. As new data arrive, it determines if the data are new, contain differences from earlier reports, etc., and append or merge them with the existing plotfile data for that particular station and observation time.

7.4.5 Text

The text decoder system consists of two separate decoders, each of which handles a different format of text product. The CollDBDecoder decodes all collective text products and the StdDBDecoder decodes all

standard text products. Each decoder determines the format of the product by the product type identified in the data's WMO header. The decoders then use the TextDB class which implements the communication between the decoders and the text database to write the decoded message to the text database.

Each text product has three sections: the Communications Control Block (CCB), the WMO header, and the data section.

7.4.5.1 Communications Control Block (CCB)

The CCB section gives the distribution information for the product. The first item in the CCB is the length of the block. This is the only data content parsed from the CCB. The length of the block is used to skip the CCB section and the block is not stored with the rest of the decoded message.

7.4.5.2 WMO Header

The second section of a product is the WMO header which has the form T1T2A1A2ii CCCC YGGggg [BBB](cr)(cr)(lf)

T1T2A1A2ii The data designator.

- T1T2 - The data type and/or format.
- A1A2 - Geographical and/or time designators.
- ii - Number used to differentiate products which contain data in the same code and originate in the same center.

CCCC The origin. A four letter location indicator of the station originating or compiling the product.

YGGggg Date-time group.

- YY - Day of the month.
- GG - Hour in Coordinated Universal Time (UTC).
- gg - Minute.

BBB Indicator that the product is an addition, correction, amendment, or sequential part of a long product. Not present in the original product.

(cr)(cr)(lf) ASCII codes for carriage return (cr) and line feed (lf). The individual decoders use the data designator and the origin of the product to determine the product type and format.

7.4.5.3 Product Data

The third section, the data section, of the product can be either collective or standard. A collective product contains reports from several stations that are combined into one product. The reports from a collective product are stored in the database as separate messages and each report contains a station identifier for that report. The format for the collective product is [Station ID] [Reported data] [Record Separator: `(cr)(cr)(lf)` or `(cr)(cr)(lf)\036']`. A standard product is a report from a single station. The standard product may contain part or all of the AFOS ID for that product as the first part of the section.

7.4.5.4 AFOS Identifier

The AFOS ID is a nine character unique identifier that identifies the originating NWS office; the product type, i.e. METAR, TAF, Flash Flood Guidance, State Weather Roundup, etc.; and the reporting station. The AFOS ID has the form CCCNNNXXX, where

- CCC - The originating NWS office identifier;
- NNN - The product type identifier;
- XXX - The reporting station identifier.

This AFOS ID is used to store and retrieve text products from the text database.

7.4.5.5 TextDB Class

The TextDB class receives a decoded message from the text decoders and, using DMQ and the TextDB_Server program, writes the message to the text database. The database identifies each message by its AFOS ID, the version number, and the time the message was created.

7.4.5.6 Collective Text Decoder

The CollDBDecoder parses and stores a collective product to the database. A collective product is identified by using the data in the collective_table.dat data file. The decoder then breaks the data section up into separate messages for each reporting station by searching for a record separator. The decoder uses upair_table.dat to determine if the report is an upper air product and get the NNN portion of the AFOS ID. These collectives include station numbers, which are used to get the XXX by checking the data in station_table.dat, and the CCC is obtained using national_category_table.dat. For other products, the decoder uses the 3- or 4-character station identifier from the report and national_category_table.dat to create the product's AFOS ID. The decoder then writes the message and ID to the text database using the TextDB class.

7.4.5.7 Standard Text Decoder

The StdDBDecoder creates an AFOS ID for each non-collective format product and stores the message in the text database. Because some standard products contain either the entire nine character ID or the last six characters of the ID as the first portion of the data section, the decoder checks the product for the nine character ID and stores the decoded message if it is present. If not, the decoder tries to create the ID with the first six characters of the product and the CCC portion of the AFOS ID obtained from the afos_lookup_table.dat. If the CCC is not present in the table or the ID is not valid, the decoder then goes to the ispan_table.dat file to get the full nine character ID. Once the ID is created and before the product is written to the database, the decoder checks the NNN against the bit_table.dat file to see if the product type is designated as a national bit product and, if so, replaces the CCC portion of the ID with the designator for the local NWS office.

7.4.5.8 Data Tables

The different data tables that are used by the decoders are designed to allow the decoders to be updated with changing products and to be portable from NWS office to NWS office without having to re-compile the code. The updates and/or changes can be made to the tables and the decoders restarted

and the changes are implemented. Each decoder will read and store only the tables that it uses to decode messages, so not all decoders may need to be stopped to implement an update or change.

The `collective_table.dat` contains the data designator and corresponding partial AFOS identifier for that product type. Each entry in the data file is of the form `T1T2A1A2ii CCCnnnXXX`

- `T1T2A1A2ii` - The data designator for the collective product.
- `CCCnnnXXX` - The product identifier portion of the AFOS ID with generic place keepers for the rest of the ID, e.g., `CCCMTRXXX` for METAR products.

The `national_category_table.dat` contains the three or four-character station identifier and corresponding NWS forecast office identifier for each station. These two identifiers make up the last (XXX) and first (CCC) portions of the AFOS ID, respectively. Each entry of the data file is of the form `XXX CCC`, where

- `XXX` - The 3- or 4-character station ID
- `CCC` - The matching office (NWS, Environment Canada, or international) ID.

The `upair_table.dat` is in the same format as the `collective_table.dat` and contains the data designators and partial AFOS IDs for each upper air product.

The `station_table.dat` contains the station numbers and corresponding character station identifiers for each station. Each entry in the station table is of the form `NNNNN XXX`, where

- `NNNNN` - The five digit station number.
- `XXX` - The character station ID for that station.

The `afos_lookup_table.dat` contains the origin and corresponding NWS office identifier. Each entry in the data file is of the form `Oooo CCC`, where

- `Oooo` - The four character origin.
- `CCC` - The NWS office identifier for that location.

The `span_table.dat` contains the data designator and origin with the complete corresponding nine character AFOS ID. Each entry is of the form `T1T2A1A2iiCCCC CCCNNNXXX`

- `T1T2A1A2iiCCCC` - The combined data designator and origin for the product.
- `CCCNXX` - The complete AFOS ID for the product.

The `bit_table.dat` contains the `NNN` portions of the AFOS ID that are national bit products and the national bit designator which is "AAA." Each entry of the data file is of the form `NNN AAA`:

- `NNN` - The product indicator of the AFOS ID.
- `AAA` - A literal string that is an indicator for the decoder to replace the string with the `CCC` for the local NWS office.

7.4.6 BUFR

BUFR is a binary data compression encoding format. It is used to reduce the data transmission bandwidth for exchanging raw data observations. It was used to transmit data from a network of remote meteorological data collection platforms (Mesonet) to FSL, where the data were deBUFRed, before they were decoded and stored in the LDAD plot files and netCDF data files on WFO-Advanced. FSL's Mesonet has been decommissioned, so the BUFR decoder is not currently in use.

RAOB data (see Section 7.4.4) are received in BUFR as well as text, and we have been working on a RAOB BUFR decoder. That will be available in a future release.

The BUFR decoder was adapted from code provided by Facilities Division at FSL.

7.4.7 Redbook Graphics

The Redbook Graphics Decoder was adapted from software provided by the National Centers for Environmental Prediction (NCEP). The graphics products which are sent over the SBN are in the AWIPS Redbook format as documented in *Standard Formats for Weather Data Exchange Among Automated Weather Information*. The Redbook Graphics decoder is called directly from the depictable (RedbookDepict class) as opposed to being started by a DataController. These products are handled uniquely in that there is no accessor for them and they are not converted into any other format but instead are decoded "on the fly" by the depictable. The original code was written in Intergraph/VAX Fortran. The main changes made were changing the calls to plot lines, vectors, and text from the Intergraph MicroCSL library to our Fortran Graphics functions which reside in our depict directory. The Redbook Decoder gets called from the paint member function of the Redbook Depictable.

7.5 Storage/Formats

7.5.1 Radar

Narrowband WSR-88D radar products are stored in their receipt format in \$FXA_DATA/radar/<radarName>/<productID>/<elevation>/<resolution>/<level>/<yyyymmdd_hhmm>. The receipt format is the WSR-88D "archive IV" format.

A WSR-88D product message consists of a number of blocks, headed by the message header block, and followed by the product description, product symbology, and graphic alphanumeric and tabular alphanumeric blocks. Not all products contain all five blocks, but the blocks are always transmitted in the same order. The product symbology, graphic alphanumeric, and tabular alphanumeric blocks have block IDs of 1, 2, and 3, respectively.

The message header block contains the message code, date and time of message, length of message, an identification of the source and destination radars, and the number of blocks in the message.

The Product Description Block (PDB) contains information about the radar and the product itself. The radar information consists of the latitude, longitude, and elevation of the radar; volume scan; operational mode; and volume scan date and time. The product description consists of the product identification code, product generation date and time, various and sundry product dependent parameters, and scan elevation number. There is also other information such as the offsets to the other three blocks. If a block is missing, its offset is zero.

The Product Symbology Block (PSB) contains the actual data fields. The data are in the form of display packets, each packet having a unique identifier.

The Graphic Alphanumeric Block (GAB) contains information for displaying storm-related data as a graphic.

The Tabular Alphanumeric Block (TAB) is used by "paired" products. It consists of textual information pertaining to the actual product. The TAB has its own header and PDB.

A detailed description of this and other WSR-88D formats is given in the Interface Control Document (ICD) for RPG/Associated PUP.

7.5.2 Satellite

The satellite data ingested by WFO-Advanced is remapped GOES satellite images. The remapped GOES product format consists of a header followed by "n" records, followed by an "end-of-file" identifier. Each record is one scan line of the satellite image. The header, which is called the Product Definition Block, contains such information as the source ID, sector ID, number and size of logical records in the product, valid time, map projection indicator, etc. This information is described in the AWIPS/NOAAPORT ICD.

The GOES satellite data are decoded into netCDF by the satellite decoder and are stored in directories according to scale. These are the east and west CONUS, Northern Hemisphere, super National, and CONUS. The full directory path is \$FXA_DATA/sat/<scale>/<imageType>. The "imageTypes" are visible, infrared, and water vapor.

7.5.3 Grid

All WFO-Advanced grids are stored in netCDF format. For each model, covering a specific domain, there is a text format CDL (network Common Data form Language) file that defines all the dimensions, variables, and attributes for the binary netCDF file. Refer to the "NetCDF User's Guide" for detailed information on all the storage and retrieval functions, format guidelines, and other questions about netCDF.

The WFO-Advanced netCDF files for grids are all set up with the same format. Each file contains an entire model run. This means the analysis and all the forecast grids are included in one big netCDF file. Each record represents a forecast period (valid time). The first record is the analysis data and any records following represent data for a pre-defined forecast time in the CDL file. Obviously, the size of the file depends on the size of the grid and how many forecast periods

there are for that particular model. All records within one file are defined identically with regard to allocating space for each field at all defined levels. For each field, there is a corresponding level variable with a list of all defined levels. There is also a corresponding inventory variable that is dimensioned by the number of levels. The following example illustrates this:

Dimensions defined for the entire netCDF file:

```
x = 93;  
y = 65;  
n_valtimes = 9;  
charsPerLevel = 10;  
levels_4 = 4;
```

The variable geopotential height:

```
float gh(record, levels_4, y, x);
```

- for each forecast time (record) there are 4 levels of data containing $x*y$ number of points.

There is a level variable associated with each field:

```
char ghLevels(levels_4, charsPerLevel);
```

- for each level there is an associated level type and level value. In this example, there are 4 levels dimensioned by 10 characters that represent the level type and level value. This is shown below.
- `ghLevels='MB 1000 '`
- `'MB 850 '`
- `'MB 700 '`
- `'MB 500 '`

There is also an inventory variable associated with each field:

```
char ghInventory(n_valtimes, levels_4);
```

- for each record(`n_valtimes`), there are 4 levels that are flagged as containing data or not for gh (geopotential height).

The forecast times are defined as follows:

```
int valtimeMINUSreftime(n_valtimes);
```

- this variable is dimensioned by the number of forecast times (`n_valtimes`) that were defined for this grid. Valtime (valid time) is the forecast time (e.g. 0, 6, 12, 18-hour forecast). Reftime (reference time) is the time of the grid run. This variable `valtimeMINUSreftime` then represents the number of seconds beyond the reftime for each record. The values would look like this:

```
valtimeMINUSreftime=0,21600,43200,64800,86400,108000,129600,151200,172800;
```

- these 9 valtimes represent analysis(0), 6, 12, 18, 24, 30, 36, 42, and 48-hour forecast periods. The order of these times corresponds to how the netCDF file is dimensioned record-wise. The analysis is the first record and the 48-hour forecast is the 9th record.

7.5.4 Point Data

All data sets in this section are defined as "Point Data" sets. Point data values are individual observations recorded at a single point in space and time. The spatial location is recorded as the latitude and longitude where the data value was recorded. The time that an observation was made is also recorded to the nearest second. These data sets are decoded from binary or ASCII data sets and then stored in their corresponding plot and netCDF data files. Point data sets are also maintained for weather data received from the Colorado Department of Transportation (CDoT) weather sensors, and the NOAA wind profiler network. All these data sets are stored in the "/data/fxa/point" root directory which is further divided into subdirectories for each of the individual data sets in this section.

7.5.4.1 RAOB

As explained in Section 7.4.4, upper air data (RAOB) get decoded from their raw message text format and stored as a SeqOf<Raob_Info> object. These are the data which get used by RaobSoundingDepict.C. Raobdecoder::storeRaobData handles the actual storage of the decoded data in either netCDF or plot file formats, which are put in the /data/fxa/point/raob/ directory. Plot file data sets are stored in the plot directory, and netCDF data sets (currently not being created) are stored in the netcdf directory. The plot file naming format is YYYYMMDD_TIME (e.g., 19961022_1200 contains the 12Z data for Oct 22, 1996). Each plot file contains 12 hours worth of data, so the 0000Z plot file contains all the RAOB data from 0000Z through 1159Z, and the 1200Z file contains 1200Z through 2359Z data. Each RAOB report in the plot file contains the WMO station identifier, station name, latitude and longitude, station elevation, synoptic time of the observation, number of mandatory and significant temperature and wind levels, number of tropopause levels, number of max wind levels, number of pressure observation levels, and number of height levels, with the rest of the data grouped in the appropriate type levels (mandatory, significant temperatures, significant winds, etc.) as outlined in the above station header information.

7.5.4.2 METAR

METAR data are stored in binary plot files and netCDF files. The plot files are used for display purposes while the netCDF files are used for access to METAR data by outside users.

The plot files are hourly binary files located in /data/fxa/point/metar/plot. As each METAR report is decoded, the observation time inside the report is used to determine to which hourly file to write. These files actually contain data from 15 minutes before the hour to 45 minutes after the hour. (A report time of 0448 Z would go into the 5 Z hourly file.) Each report is appended to the file. The following information is contained in the plot file for each report:

- stationID (the 4 character station id)
- timeObs (the time of the observation)
- reportType (METAR or SPECI)
- skyCover (CLEAR, FEW, SCATTERED, BROKEN, or OVERCAST)
- skyLayerBase (the sky layer base in feet)

- visibility (visibility in statute miles)
- presWeather (present weather phenomena)
- seaLevelPress (sea level pressure in hectopascals)
- temperature (temperature in degrees F)
- dewpoint (dewpoint in degrees F)
- windDir (true direction from which wind is blowing in tens of degrees)
- windSpeed (wind speed in whole knots)
- windGust (wind gust in whole knots)
- precip1Hour (precipitation accumulation in the last hour)
- precip3Hour (precipitation accumulation in the last 3 hours)
- pressChangeChar (the pressure tendency change character)
- pressChange3Hour (the pressure change over the last 3 hours)

Each METAR record that is written out to a binary plot file is also written out to a netCDF data file. The METAR netCDF data files are located in the "/data/fxa/point/metar/netcdf" directory. Both the METAR plot and netCDF data files contain the same stations, but the netCDF data files contain a few different parameters. The following information is contained in the netCDF file for each METAR report:

- wmoid (WMO numeric station ID)
- stationName (alphanumeric station name)
- latitude (station latitude)
- longitude (station longitude)
- elevation (station elevation in m)
- timeObs (the time of the observation [seconds since 1-1-1970])
- timeNominal (METAR hour [seconds since 1-1-1970])
- reportType (report type [METAR or SPECI])
- autoStationType (automated station type)
- skyCover (sky cover [CLR, FEW, SCT, BKN, OVC])
- skyLayerBase (sky-cover layer base in m)
- visibility (visibility)
- presWeather (the present weather recorded using surface synoptic codes)
- seaLevelPress (sea level pressure in hectopascals (mb))
- temperature (temperature in kelvins)
- tempFromTenths (temperature from tenths of a degree Celsius)
- dewpoint (dewpoint temperature in kelvins)
- dpFromTenths (dewpoint temperature from tenths of a degree Celsius)
- windDir (wind direction in degrees)
- windSpeed (wind speed in ms-1)
- windGust (wind gust in ms-1)
- altimeter (altimeter setting in hectopascals)
- minTemp24Hour (24 hour minimum temperature in kelvins)
- maxTemp24Hour (24 hour maximum temperature in kelvins)
- precip1Hour (precipitation accumulation in the last hour)
- precip3Hour (precipitation accumulation in the last 3 hours)
- precip6Hour (precipitation accumulation in the last 6 hours)
- precip24Hour (precipitation accumulation in the last 24 hours)
- pressChangeChar (the pressure tendency change character)

- `pressChange3Hour` (the pressure change over the last 3 hours)
- `correction` (corrected METAR indicator)
- `rawMETAR` (the raw ASCII METAR message string)

7.5.4.3 Lightning

Lightning data are collected by the National Lightning Detection Network (NLDN) and received on the SBN network in an encoded binary format. This data set contains the location that a lightning strike was detected given by its latitude and longitude and the time that the flash occurred recorded to the nearest second. The lightning flash's signal strength is recorded in kiloamperes. Also included is the multiplicity and polarity (either positive or negative) of each flash. After the binary lightning data are decoded they are stored in plot and netCDF data files. The plot data files are stored in the `"/data/fxa/point/binLightning/plot"` directory. The netCDF data files are stored in the `"/data/fxa/point/binLightning/netcdf"` directory. These parameters are stored in the lightning plot and netCDF data files:

- `latitude` (latitude where the lightning strike occurred)
- `longitude` (longitude where the lightning strike occurred)
- `time` (time that the lightning strike occurred)
- `sigStr` (strike's normalized signal strength [kiloamperes] and polarity [+/-])
- `multiplicity` (multiplicity of the flash)

7.5.4.4 LDAD

At present, we use two LDAD sources: CDoT (weather stations located along highways, property of the Colorado Department of Transportation) and ALERT (river and weather gauges deployed by the Denver Urban Drainage and Flood Control District. These are decoded and stored as netCDF in `/data/fxa/point/cdot` and `/data/fxa/point/alert`, respectively, and collected for plotting in `/data/fxa/point/localdata`.

7.5.5 Redbook Graphics

Redbook graphics products are sent over the SBN and stored in the `$FXA_DATA/ispan/graph` directory, with file name `<WMO ID><YYYYMMDD_HHMMSS.mmm>` (e.g. `PYMA85KWBC.19961217_120605.929`; the `mmm` part is in milliseconds). They are decoded directly by the Redbook depictable paint member function. The products are not converted in any way before being stored.

7.6 Notification

The purpose of the Notification Server is to provide notification to its display clients that a new version of a Depictable is available.

In discussing the Notification Server, it is important to distinguish between data notifications and depictable notifications. Data acquisition processes send data notification messages to the

notification server. The notification server converts these data notifications into corresponding depictable notifications, and notifies its workstation clients that depictables are available. In converting from data to depictable notifications, the notification server acts as a bridge between the data acquisition and workstation display components of WFO-Advanced.

The notification server relies on the same depictable inventory software used by the workstation to convert from data to depictable notifications.

The notification server process runs on the data server host processor. Data acquisition messages notifying it of new data sets do not invoke any network traffic, although they do cause interprocess communications overhead. When a new depictable is available as a result of new data availability, network traffic is generated to send the resulting depictable notification message to registered clients. In general, the number of depictable notifications is smaller than the number of data notifications, so this strategy is deemed network-efficient. The calculation of whether or not new depictables are available when data notifications are available can be computationally intensive, so having this overhead in one notification server process on the data server saves additional CPU overhead on each workstation, as well.

Clients wanting to receive depictable notifications must register the depictables in which they are interested with the notification server.

7.6.1 Notification Server Classes

7.6.1.1 class NotificationServer

The NotificationServer class implements the notification server as an abstract state machine, maintaining internal state, and responding to stimuli implemented as public and protected member functions. The NotificationServer inherits from the DataMgmtReceiver class, and implements the following protected receive... member functions to process the receipt of specific types of messages.

receiveDepictRegistration(IPC_Target, DepictKey, bool notifyAlso)

This member registers a client (identified by IPC_Target) to be notified of new occurrences for a particular depictable (identified by DepictKey). If the bool flag notifyAlso is true, a notification for the most recently available depictable is generated and returned to the client immediately. This capability is used by the user interface to generate "green times" when starting up.

receiveDepictCancellation(IPC_Target, DepictKey)

This member undoes the registration of a client for a particular depictable.

receiveClientCancellation(IPC_Target)

This member undoes the registration of a client for any depictables for which it has previously registered.

receiveNotificationServerPing(IPC_Target, unsigned seqNum)

Originally a hook for checking that the notificationServer process is alive, this routine would just echo back the message it was sent by the client. The seqNum was originally just to allow the client to distinguish between multiple messages sent.

The functionality has now expanded to include a number of additional control functions, depending of the value of the seqNum specified.

- o seqNum <1000: These messages are just echoed back to the sender.
- o seqNum = 30001: The notificationServer process exits cleanly.
- o seqNum = 30002: Unregisters the sender for all Depictables.
- o seqNum = 30003: Forces a dump of the statistic counters.

receiveDataNotification(IPC_Target, DataAccessKey, DateTime)

This member is called on receipt of a data notification message. After a data acquisition client has stored a new version of a dataset, it sends a message to the notification server with the corresponding DataAccessKey and DateTime for the new dataset. The notification server adds an entry to a list of recently notified datasets. The IPC_Target is not used by the notification server, but is maintained for consistency with other receive member functions, and possible future use by logging, etc.

In addition to the above protected receive... members that are called as a result of an interprocess communications message being received, there are two public members that are intended to be called from within the main event loop in the notificationServer process.

bool done(void)

This member returns true when the NotificationServer object is finished (the abstract state machine has reached the "final" state). The process should then exit with success status.

void timerExpired(void)

This member is called repeatedly from within the main process event loop, to handle low-priority tasks. Message handling is considered high priority (to prevent the possibility of message queue overflow). In the process main event loop, if no message was received within a 0.1 second IPC time-out interval, this routine will be called to do some processing. Since some of this processing can be rather intensive (generating depictable inventories in particular), doing it only when the message queue is empty minimizes the risk of queue overflow.

7.6.1.2 The Notification List (class NotificationListEntry)

The NotificationListEntry is a class containing a pair of items: a DepictKey and a DateTime. This class represents a notification for a new version of a depictable. Objects of this class are added to a list (the notification list) as soon as the NotificationServer discovers that a new depictable is available. They are removed from the notification list after the notification message has been sent to all interested clients.

7.6.1.3 The Client List (class ClientList)

The ClientList tracks registered clients of the NotificationServer, and the depictables for which they have registered. The list is implemented for efficient access by DepictKey, returning a sequence of clients registered for that depictable. Members are also provided that correspond to the DepictableServer registration members, adding or removing a client from the list of clients registered for a DepictKey, and removing a client from all DepictKeys in the list.

7.6.1.4 The New Data List (Dict<DataAccessKey, SeqOf<DataTime> >)

The NotificationServer also maintains a list of (DataAccessKey, DateTime) pairs received from data notification messages sent by data acquisition clients. This list is used as an optimization, to remove from further consideration any DepictKeys that don't depend on a recently notified data access key.

7.6.2 Key Notification Server Algorithms

7.6.2.1 Message Receipt and Dispatching

The NotificationServer class inherits from DataMgmtReceiver, which in turn inherits from Receiver, one of our IPC abstract base classes. When a message is received by the Dispatcher (also an IPC abstract base class), it calls the receive() member of the registered Receiver. In our case, the DataMgmtReceiver implements the receive() member, decodes it, and in turn routes it to an appropriate protected member function. These members are overridden by the protected receive... members of the NotificationServer class, described above.

The main program of the notification server process (in notificationServer.C) instantiates a NotificationServer object, and registers it with the Dispatcher (via Dispatcher::registerReceiver). The main program also registers the process as the one well-known IPC target named NOTIFICATION_SERVER (via Connection::setMyTarget).

7.6.2.2 Data Notification Arrival

After a data acquisition process has stored a new version of a data set, it sends the notification server a data notification message. Via our normal IPC mechanisms, the IPC dispatcher routes the message ultimately to the NotificationServer class receiveDataNotification() member.

NotificationServer::receiveDataNotification adds the (DataAccessKey, DateTime) pair to the newDataList if it isn't already there. It also does some optimization for METAR plot file notifications. Since there are so many of them, it adds an additional minimum time interval between notifications.

7.6.2.3 Depictable Registration and Cancellation

When a client wants to register for a particular depictable, it sends the notification server a registration message, which results in the NotificationServer receiveDepictRegistration() method being called. Similarly, cancellation of registration requests results in calling either receiveDepictCancellation (for a single depictable) or receiveClientCancellation (for all depictables registered for a client). The IPC_Target

object in these calls is the IPC target address for the client process, and can be thought of as being the client.

`NotificationServer::receiveDepictRegistration` adds the `(IPC_Target, DepictKey)` pair to the `clientList`. If the client also requested an immediate notification, an entry is added to the notification list for the requested `DepictKey`, with a special `DateTime` value of 0 which will (later) cause a notification to be sent for that `DepictKey`.

`NotificationServer::receiveDepictCancellation()` merely removes the `(IPC_Target, DepictKey)` pair from the `clientList`.

`NotificationServer::receiveClientCancellation()` generates a list of all `DepictKeys` for which the client has registered (from the `clientList`). It then individually removes all corresponding `(IPC_Target, DepictKey)` pairs for this client from the `clientList`.

7.6.2.4 Timer Expiration and Low-Priority Processing

The immediate processing that results upon message receipt within the `NotificationServer` is always relatively minor. Usually, just an entry is added or removed from a list somewhere. Where, then, does the major processing work get done? It's initiated when the main program event loop (in `notificationServer.C`) times out, indicating that no IPC messages are available. When this happens, `NotificationServer::timerExpired()` is called to handle some small chunk of processing. The time-out interval is 0.1 second, so `timerExpired()` will get called up to 10 times per second.

The nature of "a small chunk of processing" in this context is not very exact, but the idea is to do a small enough piece of work that the incoming IPC message queue won't overflow with messages before we can get back to the main event loop.

`NotificationServer::timerExpired()` calls three further member functions to perform its processing: `showServerStats()`, `checkNotificationList()`, and `updateNotificationList()`.

`NotificationServer::showServerStats()` maintains an internal interval timer. Once every interval (currently once per hour) this function dumps a set of counters to the log file. These counters are updated at various places within the `NotificationServer`, and are intended to help track usage and performance of the `NotificationServer`.

`updateNotificationList()` and `checkNotificationList()` perform a two-pass screening and generation of `DepictableNotifications`. Calculating what `Depictables` are available is based upon a data notification, because the natural `depictable` dependencies flow the other way. That is, a particular `depictable` depends upon one or more data sets, sometimes in a rather complicated way, with recursive dependencies, or alternative data set dependencies. Rather than try to invert this logic, the `NotificationServer` relies on the `DepictableInventory` mechanism to generate lists of available `depictables`, which are then correlated with `DataTimes` of data notification messages, to determine if one of the `depictable` times was generated by the data notification. If so, a `depictable` notification message is generated.

The basic approach is a polling loop, in which every 10 seconds a DepictableInventory is done for every candidate depictable. If the time of an available depictable matches a corresponding prerequisite dataset whose data notification was received in the interval, a new depictable notification is generated.

For example, consider a wind depictable that depends on the U and V wind velocity data sets. If a U data set is available before the V, there will be no depictable available that matches the time of the U, so no depictable notification will be generated. But when the V data set notifies, with the U also available, there will be a Depictable with a time that matches the V data notification, so the corresponding depictable notification will be generated.

NotificationServer::updateNotificationList() is called by timerExpired() when the notificationList is empty. It performs a first pass at notification, adding candidate depictables to the notificationList, by waiting until the polling interval (10 seconds) has passed, then comparing the clientList (all the registered DepictKeys) with the newDataList (all the DataAccessKeys that have been notified in the polling interval). If a prerequisite DataAccessKey for a DepictKey has been notified in the polling interval, the DepictKey passes this first screening, and an entry is added to the notificationList, with the DepictKey and the corresponding DateTime of the data notification. This first pass screening happens at most every 10 seconds, and can result in many entries being added to the notificationList for further processing by checkNotificationList(). Since updateNotificationList is called only when the notificationList is empty, it may not get called every 10 seconds, if multiple calls to checkNotificationList() are taking longer than that to process entries added during the previous polling interval. NotificationServer::checkNotificationList() is called by timerExpired() when the notificationList is not empty. Its purpose is to process one or more entries on the notificationList. It gets a (DepictKey, DateTime) pair from the notificationList, does a Depictable inventory for that Depictable, then checks that the DateTime is available within the inventory, and if so, sends a Depictable notification message to all clients of that Depictable. The Depictable inventory is protected by an interrupt handler using the setjmp()/longjmp() mechanism.

7.7 Inventory

7.7.1 Concepts of time in WFO-Advanced

We define valid time as the most essential time characteristic of any data. It is that point in time when a given piece of data correctly represents or is expected to represent some atmospheric state. This term has usually been applied to model data. Applied liberally, this definition means that for observational data, the valid time is the same as the observation time.

We define initial time as applying to data with a forecast component most often model data. This is the time most representative of the data that went into initializing the model. It is sometimes also referred to as the analysis time or model run time.

We define reference time as a unifying concept between data that have a forecast component and data that do not. For data that have a forecast component, the reference time is the same as the initial time. For data with no forecast component, the reference time is the same as the observation time.

We define forecast time as the length of time between the reference time and the valid time. Thus, observational data have a forecast time of zero.

With these definitions in place we can now talk about the two main concepts of time that exist in WFO-Advanced. The first is **AbsTime**, which refers to a single point in time. An AbsTime can be used to define a value for either the reference time or the valid time.

The second concept of time that exists in WFO-Advanced is the **DataTime**, which is a complete description of all the temporal characteristics which are required to access and assign a viewing order to individual items in a data set. A DataTime contains a reference time, a forecast time, a valid period, an access time, and a data key. The utility of the reference time and forecast time are obvious, and the valid time can be obtained by adding the two. The valid period is used in the case where data are valid over some finite length of time rather than a single point in time. The access time is used in the case where a dataset is stamped for access by something other than its reference time; its receipt time, for example. The data key is used in the case where data items are identified as being fundamentally different types of data when for meteorological purposes, they really differ only by temporal characteristics.

7.7.2 Inventory Types

There are two main types of inventories in WFO-Advanced: data inventories and depictable inventories.

A data inventory is usually a list of AbsTimes for which some specific data set (Denver RAOB, 200mb plot, NGM 500mb vorticity, for example) is available, although for grids it is a list of DataTimes.

A depictable inventory is a list of DataTimes for a given instance of a depictable. To do a depictable inventory, one first looks up all of the data keys upon which the depictable depends and does a data inventory on each of them. The AbsTimes are converted into DataTimes, if required, and then the multiple data inventories are merged into a single depictable inventory. With some inventories, this merging is done as an intersection, as with the u and v components of wind, and for others this merging is done as a union, as with the various gate spacings of reflectivity data available for a single tilt from a WSR-88D.

Within the context of depictable inventories, there are three main ways that data inventories are performed. The first approach is to base the inventory on file time stamps in a given

directory. Examples of this are radar, satellite, all plan view plots other than LDAD, and Redbook graphics. The second is to use an accessor class to look into each of the available data files to determine the inventory. Examples of this are RAOB soundings, profiler time-heights, gridded data, and LDAD. The final approach is to use Informix to do an inventory on data stored therein, which currently applies to text and hydro data.

There are also some exceptions to these approaches to doing inventory. Sometime, the valid time of data must be approximated from its receipt time this happens with some text data and Redbook graphics. For lightning, it is assumed that the lightning data are continuously available for the period covered by any data file. For grids, data inventories are recursive; in other words they potentially comprise other data inventories.

7.8 Retrieval

This section will cover the data retrieval strategies in depictables and applications.

7.8.1 Data retrieval in depictables

There exist C++ APIs that were written in house for all data sources that are read into depictables. These APIs generally provide data in as close to a raw format as possible; decoding has often been done but no processing has been done that would commit the data to a particular visual representation or frame of reference, beyond what is intrinsic to the data source itself. Except for grids, the data are delivered to depictables pretty much as stored on the disk. Extensions, being in C++, use these same APIs for their data access, if applicable.

As an example, METAR data are delivered to depictables as a list of records which contain station IDs, temperatures, dewpoints, winds, weather codes, etc., with no information at all concerning where to place text or wind barbs, or what symbols to use to encode the weather. It is up to the depictable to do all of that processing, and to look up in static tables where to plot each station. For lightning, the depictable receives a list of stroke locations in earth coordinates, times, and field strengths. For RAOBs, the depictable receives a structure containing the decoded mandatory and significant temperatures and winds, and it is up to the depictable to do the interleaving. The depictable gets satellite data as just a grid of bytes, and it receives raw RPG message files for radar.

For grids, the scenario is somewhat different. There is an API similar to the ones previously discussed called GridAccessor, which just retrieves raw grids straight out of the netCDF file. However, depictables actually make use of an intermediate layer called the GridSliceAccessor, which calculates virtual data. Virtual data includes things like doing hydromet calculations, doing finite differencing, packaging up components to make vector data, or packaging up a stack of grids to make a three dimensional representation of some parameter. The

GridSliceAccessor does no remapping of data, however. It is up to the depictable to slice a cube to make a cross section, or to remap gridded data to the output scale, if needed.

7.8.2 Data retrieval in applications

For applications, most data retrieval will be using netCDF directly, and some will be through APIs. All APIs for application data access will be callable from C, C++, or Fortran, although a C++ application could use the same APIs that are used in depictables. Current plans are to have application APIs for grid, text, hydro, and radar. The grid API will be a wrapper around the GridSliceAccessor. Text and hydro will use an Informix server that talks to the database through an IPC mechanism. The form of the radar API has not yet been determined.

7.9 Purging

There are two purgers currently operating within the WFO-Advanced system: scour and fxa-data.purge.

7.9.1 Scour

Scour deletes data files that have been written to disk either by the LDM, or by the decoders when they encounter data that can't be decoded. Scour is a shell script that was written by Unidata and comes bundled with the LDM and netCDF. This purger operates by checking the date the file was written, and deleting any files that are too old. There is a configuration file (scour.conf) where the user specifies the directories that should be purged, how many days worth of files to keep, and file specification information. The user can direct scour to purge all files in a directory, or only specified files. Wild card characters are acceptable. It is not possible to keep less than one day's worth of files using scour.

Here is an example of an entry in scour.conf:

```
/data/fxa/ispan/sao 1 sao.*
```

This entry directs scour to purge files in the /data/fxa/ispan/sao directory, to purge files that are older than one day, and to purge only files that match sao.*. Any other files will be left untouched.

Scour runs once every hour, as a cron job. The scour script resides in /usr/local/ldm/bin, and the configuration file is in /usr/local/ldm/etc.

7.9.2 Fxa-data.purge

The second purger, "fxa-data.purge," deletes product files written by the decoders (these are also the files used by the workstation). This purger is a temporary part of the FX Advanced software, and is expected to be replaced later as we continue to develop our purging design. Fxa-data.purge maintains a specified number of files in the data directories. This purger operates by counting the number of files in a directory, and if there are too many files present, the purger deletes the oldest files. The user specifies

the directories to purge and the number of files to keep in each directory. `Fxa-data.purge` provides a method of keeping files for less than one day, since this is currently too much data for our disks. This purger runs twice per hour, and is executed as a cron job.

7.9.2.1 Description

`Fxa-data.purge` is a shell script that consists of three parts:

- a function that performs a global purge. It purges data files in the lowest level subdirectories of a specified top level directory;
- a function that purges data files in a specific directory. This function does not access any subdirectories;
- a list of commands that call the two purging functions.

The global purge function is used for data directories that can be purged so that all the subdirectories contain the same number of files. An example is the "sat" directories, which contain satellite data. We receive three satellite data sectors via the SBN (Northern Hemisphere, superNational, and westCONUS), and create a fourth, CONUS C, locally; there are five channels for each scale. We receive approximately the same amount of data for each scale, so we can purge all of the satellite subdirectories identically. This function takes two arguments: the name of the top level directory, and the number of files to keep in the subdirectories.

Some data directories, however, contain data that are not as consistent as the satellite data, and these directories are purged separately. An example is the "grid" directory, which contains gridded data from different models. Some of the models execute every twelve hours, and some every three hours. Different numbers of files are needed in order to keep data for the same time period. The second function mentioned above is used for this type of purging. This function also takes two arguments: the directory to purge, and the number of data files to keep in the directory.

In the section containing the list of commands, the user specifies which purging function to use, which directories should be purged, and how many files to keep in each directory. Users of the purger must be familiar with the FX Advanced data directory structure.

`Fxa-data.purge` uses the UNIX `ls` command to list the files in a directory. Since the file names are actually dates, `ls` lists the files in ascending order, with the oldest files first. This allows the purger to easily find and delete the oldest files - it just deletes the first files in the list. This purger will not work in directories where the file names are not ordered like this.

7.9.2.2 Purging criteria

There are two criteria to consider when deciding how many files to keep on disk for a particular type of data:

- size of the data files
- workstation loading requirements

In a system with limited disk space, the size of the data files can be important in deciding how many files to keep.

Workstation requirements are also a driving factor in determining how many files of each type of data to keep on disk. The FX Advanced workstation can load 32 frames, but sometimes just keeping 32 files isn't enough. For example, it seems logical to keep 32 files for each type of satellite data. This amount of data would cover a time period of 8 hours, if the images are arriving every 15 minutes. However, when a forecaster loads model analysis and forecast data, the time span covered will generally be longer, possibly several days. If the forecaster wants to overlay model and satellite data, there won't be enough satellite data. So it is important to coordinate with the forecasters when deciding on the amount of data to keep.

7.9.3 Future Plans

At present, disk space is limited, so WFO-Advanced is keeping a limited amount of data. In the future, it is envisioned that the system will have enough space for around three days of data, so the careful control of fxa-data.purge won't be needed. The future purging system may make use of soft links to point the data ingest and workstation systems to the appropriate disk for current data. This system will also provide what is needed to begin data archiving, and also accommodate loading and displaying case studies. Details for the future of purging on FX Advanced have not been discussed, however, so these ideas have not yet been evaluated or adopted.

CHAPTER 8 User Interface

8.1 WFO-Advanced UI Philosophy

The user interface on PROFS and FSL workstations has been a key consideration of the overall workstation design from the beginning, receiving a great deal of time and attention. Our approach has been to design, build, and then modify (sometimes extensively) prototypes based on user feedback. The feedback is usually generated in a pseudo real-time forecast exercise which approaches actual operational use. Over the years, many designs have been tried. The interface has evolved based on this experience and in keeping with advances in computer technology. For example, early PROFS workstations utilized dedicated menu screens with a variety of product selection methods including command line, light pen, and touch screen. These generally worked satisfactorily when there were only a few products available as was the case with the early workstations. As the number of available products increased, however, it became necessary to use more precise and efficient product selection methods.

By the mid 1980s the mouse had become the industry device of choice for user input and selection. We quickly adopted it as well. Its precise point and click capabilities allowed us to include many more menu choices (buttons) on the screen. However, even with the mouse, our user interface continued on a separate dedicated screen for some time. This configuration created serious ambiguity problems when we began using one selection device to drive two display screens. It also had the disadvantage of requiring an additional display screen per workstation. Eventually, the separate menu screen was abandoned when windowing technology became available.

With the emergence of the X window system, it became feasible for menus to share the data-display screen. Again, we quickly began to take advantage of this advance in technology. However, the ability to have multiple display windows on a single screen immediately presented us with a fundamental user interface issue: What is the best way to control multiple windows on an operational meteorological workstation? It is typical using X to have multiple windows open simultaneously, each with its own set of menus and controls. Initially, we experimented with three UI approaches: 1) multiple windows each with their own menu, 2) multiple windows loading from a single menu, and 3) a "fixed panes" approach with menus loading to one main window. We tested prototypes of the first 2 of these interfaces in early versions of WFO-Advanced and found that they both presented many difficult operational problems. With multiple windows open, the forecaster faced difficult window management and

input ambiguity issues. These problems were exacerbated in the "heat of battle" when they could least be afforded.

We believe that an operational forecast workstation must differ substantially from the standard X philosophy. FSL's approach assumes that a forecaster workstation is not simply a collection of related applications, but should be a tightly integrated system of tools and capabilities in which frequent interaction with displayed data is the norm. If there is little data integration or forecaster interaction with the data, then UI options 1 or 2 (above) would probably be preferable. The forecaster could simply load a particular data set, say an IR image loop, into a window and passively let it run for long periods. We currently see some workstations used this way. However, the forecasting environment as envisioned for the AWIPS era, and more particularly the short-term warning environment, demands a time-efficient and unambiguous user interface.

This experience and user feedback from the first two FX-Advanced prototypes drove us to the latter of the above three options, the fixed panes approach with menus loading to one main window. Although less flexible, this layout provides a more structured environment for operational use. In addition, and to support the need for time efficiency, the single menu/single load window solution best accommodates tear-away menus which allow users to bypass the normal menu hierarchy and quickly load desired products.

The selection of the fixed 5-pane display area involved careful trade-offs. We traded some flexibility for simplicity and efficiency. This approach virtually eliminates window management tasks and searching through window stacks for a desired display or menu, yet it allows multiple datasets to be visible at the same time with the ability to quickly swap between the main window and the smaller monitor windows when needed. It may not be obvious to the casual observer that this trade-off is the optimum, but the experience in real-time forecasting experiments and in actual meteorological operations validates the benefits of this approach over the others that have been tried.

8.1.1 Layout

The interface for WFO-Advanced can be configured in several ways. The preferred configuration is a mouse, keyboard, and keypad for each graphic display screen. This allows a two headed system to be used by two users when circumstances call for it. This configuration includes two X servers, one for each screen, providing a somewhat more responsive system. This requires some additional memory in the workstation.

Other configurations are also possible. The keypads can be removed, for instance. The impact is that experienced users are slowed somewhat in their interaction with displayed data. While all of the functions can be exercised via buttons, pop-ups, and menus using the mouse on the display screen, this requires significant additional mouse travel.

Another option is a single mouse/keyboard for two screens. This has been run experimentally at FSL. In this case, a single X server is used for the two display screens. Although forecasters

were trained to use this and most felt comfortable, it has not been tested operationally at Denver. The loss of flexibility wherein only one user can operate the two graphic displays raised sufficient concern that it has not been implemented operationally. The ability for another user to operate one of the displays has been considered of such importance to the smooth operation of the office that the single user interface was not set up in Denver.

8.2 User Interface Library

8.2.1 Overview

The User Interface class library was designed to be used to construct Graphical User Interface (GUI) components and presentations for all FX-Advanced components that require a user interface.

Essentially, it is the description of building blocks to be used to create a GUI. The design does not dictate how a GUI will look, but rather provides the flexibility to use a set of components with which a GUI that is appropriate for an application can be constructed and customized. The components' look and feel are restricted to be similar to Motif widgets.

The library design has three major objectives:

1. *Hide the implementation.* Provide an interface that encapsulates and hides the underlying implementation. This will allow us (albeit with significant effort) to base the library on a different underlying product (e.g., Motif API instead of OI) without requiring any recoding of library clients.
2. *Uniform look-and-feel.* Provide a uniform look-and-feel for FX-Advanced components and applications that is automatically and centrally controlled and enforced. For instance, we could change the visual appearance of PushButtons in one place and have that change apply to all PushButtons in our system. Clients may need to re-link but client code will be unaffected. This goes well beyond the level of enforcement of look-and-feel provided by the Motif style guide.
3. *Simplified API.* Create a simplified programmer's interface to user interface elements. We will accomplish this by
 - ◆ creating a smaller API with fewer options,
 - ◆ providing automated layout assistance,
 - ◆ tailoring the available user interface elements for our needs, and
 - ◆ providing an easy-to-use mechanism for library extensions.

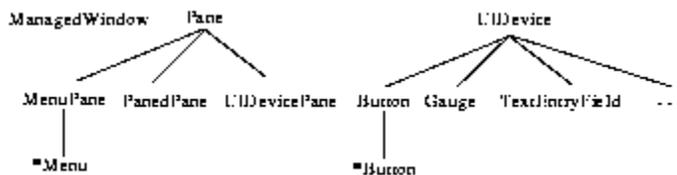
The library provides:

1. Basic elements that we will use in our user interface:
 - ◆ buttons (PushButtons, ToggleButtons, and MenuButtons);
 - ◆ menus for containing buttons;
 - ◆ value entry fields for user input of numeric values and character strings;
 - ◆ sliders for user selection of numeric values;
 - ◆ PictureAreas for display of image and graphic data;

- ◆ TextAreas for display and edit of text data; and
 - ◆ ManagedWindows, that is, top-level windows with window-manager specific borders and decorations and under the control of the workstation window manager (e.g., Motif or HP VUE).
2. Panes, used to specify the visual layout organizations ("geometry" in Motif parlance) of other elements.
 - ◆ They provide automated layout assistance via an attribute that allows the client to specify left to right or top to bottom orientation for the contents, and
 - ◆ they may be nested.
 3. Infrastructure for constructing reusable complex elements (such as a time range selection dialog) that are developed as extensions to the library.

The major classes and inheritance relations are shown in Figure8.1.

Figure8.1 Overview of major classes and inheritance relations



The object hierarchy of UI lib classes in an application starts with a ManagedWindow which contains a top level PanedPane. This PanedPane can contain additional PanedPanes which can be additionally nested. Any of these PanedPanes can then contain a MenuPane and/or a UIDevicePane. MenuPanes contain menus which in turn contain buttons. UIDevicePanes contain UIDevices including buttons.

The implementation of the library was done with a product called Object Interface (OI), developed by ParcPlace and now maintained by OpenWare. OI running on Unix machines utilizes Xlib for its implementation. It is capable of presenting both a Motif and OpenLook look and feel interface. For the purposes of this design, the Motif style will be used.

Applications built using the library will also be able to use the OI library and the X library for development. Use of X is strongly discouraged since it makes the applications more difficult to maintain and may cause future portability problems

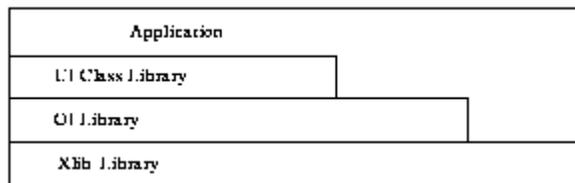


Figure 8.2 Application Access to Libraries

8.2.2 Client Classes and Callbacks

The overall callback scheme for applications using the UI Class library follows the client/server model. An instantiated object from the UI Class library acts as the server, the application object is the client, and

the client inherits the callback protocol from an abstract class defined by the server. For many of the UI classes, there is a corresponding client class. This class defines the message protocol to be used for callback. Member functions of the client class are defined to be pure virtual. It is the client's responsibility to define the implementation of these inherited functions. The UI object is handed a pointer to the client object at creation time.

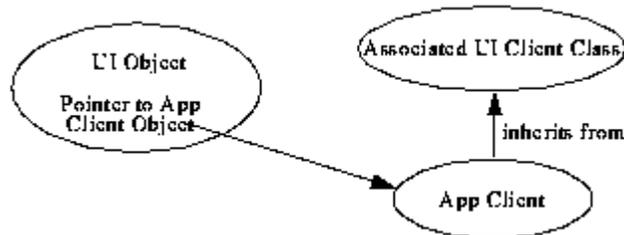


Figure 8.3 Client classes

8.2.3 Caveats - Tcl/Tk

This library is soon to become obsolete. We are planning to replace it with the Tcl command language and its Tk tool kit.

8.3 Software Design

The user interface process, fxa, is the main process of D2D. It has a number of responsibilities:

- It constructs the user interface and displays it on-screen.
- It starts and monitors the five IGC processes (each one serving one of the display areas).
- It monitors the user's actions on menu entries, buttons, and other aspects of the interface, such as a product load, and turns them into requests sent to the IGCs, such as a load request.
- It receives IPC messages from the active IGC (the IGC serving the large display area) and updates aspects of the user interface, such as the frame count display.
- It starts and monitors applications (see Section 11.1, "Applications").
- It updates the latest inventory times on every menu entry representing a product.

Note: the information in this section will soon be obsolete as we migrate the entire fxa process into a combination of Tcl script and C++ code.

8.3.1 FXA Process Design

The architecture of the fxa process is based around these two designs:

- Singletons
Singletons are objects that are guaranteed to have only a single instance and with a global point of access. The various singletons maintain state information for their specific domains of the process (such as the map menu, the applications, etc.). They receive messages from each other and from the IPC interface, and as a result of the user's interaction with the user interface.

- Object-oriented callbacks
In most user interface libraries, a programmer passes a function to the library to be called when the user uses a widget. We use instead an object-oriented callback: we pass an object to the library; the object is of a class derived from an abstract base class specified by the library. Sometimes, these objects are singletons.

8.3.2 Classes/Objects

This section describes each of the classes/objects in the fxa process.

8.3.2.1 Tiled Display Window

The singleton of class TiledDisplayWindow is what gets everything started. Constructed inside of the fxa process' main function, it constructs the five display areas, menu bar, tool bar, and status bar.

The constructor uses the user interface library to create a top-level window containing four small display areas and starts an IGC process in each using the UI workspace manager (see below).

It then adds to the window the menu bar. The menu bar contains the fixed menus File and Scale. The singleton ScaleMenuMgr is responsible for building the scale menu. The remaining menus except the Map menu are specified by the dataMenus.txt file. This file tells what pull-down menus, cascade menus, product buttons, application buttons, and title buttons appear in the menu and where. The format of this file is straightforward; comments in the file give details. The constructor adds the map menu, created by the singleton MapMenuMgr, and adds the clock, created by the singleton Clock.

Callbacks for the widgets built by various singletons are the singletons themselves. Callback for application buttons is the AppMgr singleton. Callbacks for the product buttons are objects of class BB2_ProductButtonClient.

Next, the constructor builds the tool bar. The tool bar (also known as the control bar in the code) creates the clear button, frame step buttons, and the controls menu. The TiledDisplayWindow is itself the callback client of these buttons. On the controls menu, three singletons are the callback clients: GraphicsControlsMgr, ImageControlsMgr, and AnimationMgr (the looping controls). It then adds the load mode button, created by the LoadModeMgr singleton. It ends with the WarnGen button; the TiledDisplayWindow is itself the callback client.

Next, the constructor builds the large display area and starts an IGC to serve it.

Finally, the constructor builds the status bar. It contains two display areas: the left for general status, and the right for radar. Two AnnouncementDisplayer objects are created, one to display general messages, the other for radar messages.

8.3.2.2 UI Workspace Manager

The singleton UI_WSM_Impl, known as the "workspace manager," maintains information about all IGC processes. It maintains a mapping between ID numbers (called UI WSM IDs) and the IGC objects that

represent running IGC processes. It also tracks which IGC is the active one. The active IGC is the one serving the large display area.

The TiledDisplayWindow singleton calls methods in the UI_WSM_Impl singleton to create IGCs by calling createWSMtuplet. This method takes a size type (large or small), a Cartesian pixel extent, and the window in which the IGC should display itself. It then creates the underlay window for that IGC as a child of the window passed in and starts the IGC process, passing it the size type.

When the user wants to swap a small IGC with a large one, the small IGC wishing to do the swap sends an IPC message which is received by the workspace manager's swapWithLargeDisplay method. This method initiates the swap: it tells the small IGC and the large, active one to suspend themselves. After responses from both IGCs have been received by the suspendAcknowledged method of this singleton, the singleton workspace manager tells each IGC to transform itself, passing the new window IDs they'll use. The singleton then saves the ID of the new active IGC.

All IGC requests (to load data, for example) are made to the active IGC. By asking the UI_WSM_Impl singleton for a pointer to the active IGC object, user interface callback objects can get the IGC to perform. For example, the BB2_ProductButtonClient callback is invoked when the user selects a product to load. The callback object determines the depict key to load. It then asks the UI_WSM_Impl for the active IGC object, and calls the load method of that object, passing in the depict key.

8.3.2.3 Product Button Client

Objects of class BB2_ProductButtonClient are the callback clients for product buttons, which are menu buttons that forecasters use to load products. The buttonSelected method of this class is invoked when a product button is selected. The class maps from the product button key (defined by class DM_ProductButtonInfo) to a depict key that depends on the current scale. (It queries the ScaleMenuMgr for the current scale.)

After determining the scale and the depict key, it tells the IGC to load the product via the LoadModeMgr (described below).

8.3.2.4 Product Button

Objects of class BB2_ProductButton (derived from PushButton, which can appear in a menu) are product buttons. They contain a product button key which they use to determine their label and also which depict key to load depending on the scale. The depict key is also used to determine which product time ("green time") to display next to the label.

As forecasters change scale, the set of available products changes. Product buttons must enable or disable themselves to reflect this information and also to change the product time displayed.

They do so by registering with the ScaleMenuMgr, which notifies every single product button what the new scale is.

8.3.2.5 Scale Menu Manager

The singleton of class ScaleMenuMgr manages the scale menu. The ScaleMenuMgr attaches a scale menu to the user interface. It builds the items in the scale menu by consulting DM_ScaleInfo.

Using an object-oriented callback strategy, callback clients can be registered with the ScaleMenuMgr. Whenever the forecaster changes the scale, the ScaleMenuMgr tells every client what the new scale is. Product buttons (described above) register so they can enable or disable themselves and display a new product time. Applications (represented by objects of class App) register to know when to send a scale change update to a running application.

Also, the ScaleMenuMgr sends the new scale to the active IGC whenever it changes. (The IGC will clear its display if there are no products loaded.)

8.3.2.6 Load Mode Manager

The load mode manager singleton is responsible for tracking the current load mode. It builds a load mode option menu in the tool bar, and remembers the forecaster's selection.

Like the scale menu manager, the LoadModeMgr keeps a set of callback clients to be notified if the load mode changes. The application interface uses this feature to send the load mode update to running applications.

The most important responsibility of the LoadModeMgr is to send load requests to the active IGC process. The parameters in the load request depend on the current load mode, so the LoadModeMgr is uniquely suited to calling IGC::load.

For certain load modes, the forecaster must enter a forecast time. The LoadModeMgr displays a dialog box and sends the request after the selection. For the Inventory load mode, the forecaster must enter both the forecast time and an inventory time. The LoadModeMgr arranges for the inventory time dialog box to appear after the forecaster makes a selection from the forecast time dialog.

Finally, the LoadModeMgr tracks if image combination is on, and sets the appropriate parameter in the load request.

8.3.2.7 Control Classes

Three singletons maintain the three different control dialog boxes:

- Looping controls: AnimationMgr
The AnimationMgr singleton creates and tracks the looping (animation) controls. As forecasters select forward and backward loop speeds and first/last frame dwell times, this singleton sends the corresponding requests to the active IGC.

- Graphics controls: GraphicsControlsMgr
The GraphicsControlsMgr singleton creates and tracks the graphics controls: the magnification and density option menus. As selections are made, it passes them to the active IGC.
- Image controls: ImageControlsMgr
The ImageControlsMgr singleton uses the ImageControlsDialog singleton to display the image controls dialog. User selections are passed to the active IGC.

The ImageControlsMgr also tracks the state of the image colors editor, which enables the forecaster to edit color tables. The image colors editor runs as a separate process which operates on the same color table used to render images in the active IGC window.

Although not appearing on the Controls menu, the FrameMenuMgr singleton builds and tracks the frame count menu and the current frame count display. When the user selects a frame count, the request is passed to the active IGC. When the IGC determines the actual number of frames available, it sends it to this singleton to update the display.

8.3.2.8 Clock Classes

Two classes maintain the display of the D2D clock and the notion of time used for the loading of products:

- Clock
The singleton of class Clock constructs the clock display widget and registers with the EventDispatcher for periodic update. Also, if the forecaster sets the D2D time with the time-setting application, the timeSettingChanged method is called (by the singleton described in the next bullet) to display the selected time in yellow instead of the current time in white.
- RTclockMgr
This singleton maintains the time. It has methods to return the time for display by the Clock singleton, and an inventory time used for loading of products. In its normal mode of operation, it always returns the current time of day for display, and the maximum possible future inventory time for loading.

When the forecaster uses the time setting application, this singleton will instead return the selected time for both display and inventory for loading.

8.3.2.9 Map Menu Manager

The singleton MapMenuMgr object creates the map menu. It uses DM_ScaleInfo to determine what maps are available and displays them on the menu. It enables and disables various maps depending on the currently selected scale. The IGC process tells the MapMenuMgr via an IPC message what scale to use.

Selecting a map from the menu results in an IGC load request with the selected map's depict key.

8.3.2.10 Announcement Classes

The announcer system is responsible for receiving and displaying status messages in the status bar at the bottom of the D2D window. It comprises the following classes:

- **AnnouncementDisplayer**
Objects of this class display announcements. The constructor creates a text area to hold the current announcement text and maintains a history of previous announcements for display in a separate dialog box. It also displays important announcements in a red dialog box. It registers with the `EventDispatcher` to know when to expire old announcements from the display.
- **AnnouncementFileHandler**
The announcement file handler watches an announcement file for new announcements. It uses the `File Access Controller (FAC)` to get automatic notification when the file changes. When it does, objects of this class read the file and notify their corresponding `AnnouncementDisplayer` objects to display the new text.
- **Announcer**
Singletons of this class reside in each process that wishes to make announcements appear in the announcer windows. The `announce` method accepts the type of the announcement, its importance, and its text. The singleton locks and saves the announcement in the correct file. When it unlocks the file, the `FAC` notifies the `AnnouncementFileHandler` objects that the file's been updated; they then read the file and tell the `AnnouncementDisplayers`.

8.3.2.11 Procedure/Bundle Classes

The D2D procedure feature lets the forecaster collect bundles, which encapsulate the IGC display state, into sets called procedures. Bundles and procedures are stored in files and can be named by the forecaster.

Bundles are created by the IGC process and are passed to the fxa process for ownership. They live in the D2D history list until they're copied to a procedure. When the procedure is saved, the bundle files are collected into a single directory named after the procedure. When selected by a forecaster, either from the history list or from a procedure, a bundle is loaded into the active IGC display. The fxa process treats bundle files as opaque data. It only manages them; it never looks inside.

Because both procedures and the history list display a list of bundles, the class `Bundle-List` is used to manage the list. This class creates a user interface list widget and populates it with text strings representing the bundle names. An object-oriented callback mechanism tells the callback client of the `BundleList` when bundles in the list are selected.

The singleton class `BundleHistory` creates a `BundleList` to display the bundles in the history list. It gets called by the IGC process to "take a bundle" whenever the process completes one. By taking a bundle, the `BundleHistory` takes ownership of the file and displays the bundle name in its contained `BundleList`. Selecting a bundle on the list and pressing `Load` asks the IGC to load it. Pressing `Copy Out` copies the bundle to every open procedure.

The class ProcedureDialog represents a procedure. It too contains a BundleList plus a number of buttons that operate on the contained bundles. Loading works as for the history list, as does copying out. Pressing Rename opens the bundle rename dialog, contained in class RenameBundleDialog. The other buttons are straightforward.

ProcedureDialogs are tracked by the ProcedureMgr singleton, which is the callback client for the New... and Open... buttons on the Procedures cascade menu.

8.3.2.12 UI_KeypadClient

This class, derived from KeypadEventClient, handles events generated by the D2D keypad (also known as the touchpad). An object of this class is created in the fxa main function and is registered with the EventDispatcher. Whenever the EventDispatcher sees output on the keypad's file descriptor, it passes it to the UI_KeypadClient object.

The UI_KeypadClient passes the request to the appropriate singleton for processing.

8.3.3 Interprocess Communication

The user interface process, fxa, has one main IPC receiver defined in the ipcUProc directory. It's the class IGC_uiProcReceiver. The IGC processes use the IGC_uiProcMsg class to send messages to the fxa process through a proxy interface. The workspace manager (see Section 8.3.2.2) is the proxy.

The IGC processes link with the class UI_WorkspaceManager. Methods in this class send IPC messages to the fxa process; these are handled by the UI_WSM_Impl (in addition to its function of tracking running IGCs and handling swapping). The fxa-side object passes the requests to the various singletons for handling.

Also, the fxa process links with other receivers that are part of other software interfaces (for example, the data management receiver for inventory notifications).

8.3.4 Starting FXA and the Main Function

To start the D2D display, one runs the start-d2d script. This script initializes the LOG_FILE environment variable, loads X resources needed by fxa and the various applications and extensions, checks and optionally stops/starts the DMQ IPC system, kills any existing run and orphaned processes from a previous run, and executes fxa. The script also displays the D2D splash page and gives an idea of the initialization progress.

The script monitors the standard output of fxa: as soon as it sees the text "Ready" it knows the fxa has initialized to the point that it's entered the Event Dispatcher loop. Because the IGCs usually haven't yet initialized at this point, the script waits a bit longer before exiting (and thereby removing the splash page).

The file uiProc/fxa.C contains fxa's main function. It registers a signal client with the signal catcher to handle shutdown (SIGTERM, SIGINTR, etc.) and child died (SIG-CHLD) signals. It

parses the command line with the user interface library class `UixConnection`. It sets the `DISPLAY` environment variable for future child processes, creates the `EventDispatcher` and the `TiledDisplayWindow`, and enters the dispatch loop.

CHAPTER 9 Interactive Graphics Capability (IGC)

The Interactive Graphics Capability (IGC) is a process that displays meteorological data (satellite images, station plots, radar, gridded model graphics, soundings, etc...) inside an X window.

An IGC process provides some user interface, but the majority of its interface comes from the UI process which is described in Chapter8.

The actual conversion from data to its pictorial form is done by Depictable objects which are described in detail in Chapter10.

This chapter will first discuss the software and hardware issues that influenced the design of the IGC. Then it will describe the IGC process environment, including what the major objects are, what they do, and how they interact with each other. Finally, this chapter will provide step by step descriptions of how the IGC implements its major features such as loading and displaying a product, zooming, looping, sampling, etc.

9.1 IGC Design Drivers

All workstations developed at FSL prior to the WFO-Advanced were carefully crafted around highly specialized display hardware. The display hardware not only performed many powerful display functions, such as zooming and panning, but also had a unique architecture that supported animation from local display memory and high speed disks. Because of this specialized hardware many display functions were easy to implement, fast in performance, and required very little of the host's processing resources.

FSL's previous workstations used a single large window to display all meteorological data: contours, images, and plots, whether they were generated in advance or interactively by means of an application. However, there was a desire on the part of a number of meteorologists to have an additional display window for the purpose of monitoring weather in a particular area or to display the output of an application, such as a skew-T or a time-series plot. A proposed solution was to create an additional small window along one side of the screen strictly for applications. This would allow the forecaster to look at applications without losing the display in the large window. For closer inspection, the user could transfer the contents of the application window to the large window. At the same time, the contents of the large display would be transferred to the small application window so it would still be visible. This approach, suggested

initially for the UNIX PC workstation became easy to implement on a UNIX workstation with the X window system.

The first prototype of this approach was implemented on a Sun Sparc 2 and consisted of one large square window and four small windows to the left side of the large window. For simplicity, a single process controlled and managed the information in all the windows. Since the Sparc2 had only an eight bit (256 colors) lookup table, the images and graphics in the five windows had to share a common lookup table. Before an image was loaded into a window the value of each image pixel was converted to correspond to the appropriate color in the lookup table. As long as all the images and graphics used similar colors this approach provided reasonably good results. However, if one of the images was displayed in the gray scale then the number of available colors and gray scale values became unacceptable. The experience with this prototype resulted in two significant recommendations: 1) increase the number of available colors by using a 24-bit color system, and 2) reduce the complexity of managing five windows by creating a separate process (IGC) for each window.

A prototype was tested on a Sun Sparc 2 GTX workstation with 24 bits of color. The five windows now shared over 16 million colors, more than adequate for all workstation displays. Each pixel of an image was converted and stored as 24 bits of data plus eight bits of zeros to fill a 32-bit computer word. The drawback to this approach was the four-fold increase in computer memory requirements. For an animation sequence of 32 1000x1000-pixel images, the memory requirement rose from 32 to 128 megabytes. This larger size resulted in slower loading and animation rates because considerably more bytes had to be transferred into and out of memory. An interesting side effect on the GTX was the sequential changing of display colors as the red, green and blue values were transferred from memory to the screen.

The introduction of Hewlett Packard workstations with the CRX24 display controller provided additional options for implementing multiple display windows. In addition to its true 24-bit color, the CRX24 supports four separate 8-bit color tables: one table for the image underlay, a second private table for overlays, a third for overlays needed by the X window manager (and also the default), and a fourth for transparent displays. Instead of deciding on either 8 bits or 24 bits of color, it is now possible to consider a hybrid approach using 8-bit and 24-bit colors for the display windows. The suggested approach uses 8-bit colors in the large window and 24-bit colors in the smaller windows. This satisfies the requirement for a full range of 256 colors in all the display windows. Using this approach, the large window requires approximately 32 megabytes and each small window approximately 5 megabytes of memory for a 32-frame animation sequence, for a total of 52 megabytes for all five windows. If additional savings in memory is desired, the maximum animation length in the small windows can be reduced to a smaller number. The total memory requirement for this hybrid approach, even when considering that the host workstation has to drive two monitors, is considered reasonable. Aside from these memory considerations, this hybrid approach also provides acceptable load and animation performance since the images are relatively small in size.

Although this approach adequately addresses the memory and some basic performance concerns, it also results in increased software complexity since the same data must be loaded differently depending on whether they are loaded to an 8-bit or 24-bit window. The display for each small window is prepared in a 24-bit (i.e. 32-bit computer word) pixmap in the host memory. When a satellite image with a graphic overlay is desired, the 8-bit satellite pixels are converted and mapped to a 24-bit pixmap and the vector graphic is then merged with the image in the pixmap. To remove the graphics from the displayed image, the satellite image is reloaded into the pixmap. To avoid the load time from the database, a second pixmap exists for each frame that contains only the image. When interactively moving a graphics line in the small window this second pixmap refreshes the image each time the line is moved. Because the windows are small (200 by 200 pixels), the performance in these small windows is good.

To load the same image and graphics combination described above into the large window requires a different technique. Images are loaded into an 8-bit underlay window and vector graphics are added by using the transparent overlay window. Each vector graphic is prepared as a bit map and combined with other graphics in an 8-bit pixmap. A technique known as "stipple fill" is employed to draw additional graphics into the pixmap. Although not practically useful, as many as 256 graphics can be drawn into the pixmap and displayed in the overlay window.

The complexity of the IGC is increased further by a requirement to subdivide the display window into four separate panels and load each panel with a set of products. This is useful if the user wants to look at radar reflectivity at four different elevation tilts or view different satellite channels. As long as the same color table is needed for all four panels, the large display window can adequately display all products using the 8-bit color architecture. One can assure that the 8-bit color architecture suffices by carefully defining the products that can be loaded into the four panels in advance. However, if the user is given the capability to modify the pre-defined four-panel display by loading new data into a panel then the possibility of exceeding the number of available colors is high. To assure that sufficient colors are always available for all panels, the IGC needs to be able to concurrently support the 24-bit color mode. A special algorithm is implemented that assigns either 8 bits or 24 bits of color to a panel based on the data displayed in the panel. In order to retain good performance, the algorithm tries to assign 8-bit color tables to as many panels as possible and re-examines the assignments each time a new product is loaded.

9.2 Process Environment

9.2.1 Child Process

The common invocation of the IGC is as a child process, invoked by the user interface process (UI process). The parent process is responsible for defining the area in which an IGC will display. For the D2D workspace, the UI will create five X windows and position them inside the UI's application window. All of these windows will be created using the underlay planes of the graphics hardware, and will be capable

of displaying images. Four small windows are created with a 24-bit visual, while the large window is created with an 8-bit visual. With this scheme, five images with different color requirements can be displayed simultaneously.

9.2.1.1 Invocation Arguments

The UI process invokes an IGC process for each of the five display areas, passing the area's X window ID as a process argument. The IGC is allowed to draw into that window, solicit events for that window, and create additional child windows to be displayed on top of that window. However, the UI process is the owner of that window, and is the only process that can destroy it. The UI process can also tell an IGC to use a different window for its layout during a swap operation which is described in detail in Section 9.6.5, "Swapping".

The UI process assigns an ID to each IGC which is also passed as a program argument. Currently the ID is an integer in the range 0..4. The ID is used by the UI process to identify which IGC process is sending an IPC message to the UI.

The UI also informs the IGC whether it is running in a small or large display area through a process argument. An IGC will exhibit the following differences depending on whether it is running in a large or small display area:

- Small IGC's provide a user interface to initiate a swap. Large IGC's do not.
- Small IGC's do not display legends, or provide an interface to operate on legends, since the display area is not big enough. Instead, they display the time of the frame being displayed.
- Small IGC have a smaller maximum frame count than large IGC's which is specified in the configuration file, "config/fxa.config."

The last argument of interest is the IPC target of the parent process. An IGC needs to send messages to the UI process in order to inform the parent of its state, so that the UI can keep its UI widgets in sync with what is being displayed.

9.2.1.2 User Interface

As a child process, the IGC is dependent upon the UI process for its user interface. Most of the UI process widgets affect only the IGC in the large display area. Since the IGC is in a different process, commands to initiate IGC actions are in the form of IPC messages.

However, an IGC running as a child process does provide some of its own user interface through the use of the mouse. An IGC can receive these X events directly. These events are enough to manage context-sensitive pop-up menus and some dialog boxes, such as a dialog to choose an overlay graphic's color.

9.2.2 Stand-Alone Process

The IGC can be run by itself without having to run the UI process. The advantages of this approach is that the start-up time for a single IGC is much less than start-up time for five IGC processes and a UI process. Also, it is slightly easier to connect a debugger to a stand-alone process. However, without the

UI process, the menu bar, control bar, and dialog boxes (including the volume browser) are replaced by a far-less elegant, harder to use keyboard interface. Thus, running as a stand-alone process is useful for developers but not practical for end users.

9.2.2.1 Invocation Arguments

An IGC requires an X window that defines its display area. The UI normally has the responsibility of creating that window. However, in stand-alone mode, the IGC has to create its own window, which will be a window with window manager decorations whose parent is the root window. If the IGC process is invoked with the "trueColor" argument, then this window will be a 24-bit window. If the IGC_Process is invoked with no arguments, then the window will be an eight-bit window.

9.2.2.2 User Interface

Some of the functionality that the UI process provides is implemented in a keyboard interface. For example, to load a product, the user would type in "[<A product key>]". The complete key bindings are described in the file, igc/KeyLegend.Doc.

The user interface provided by the mouse that is available to the IGC as a child process is also available as a stand alone process.

9.2.3 Thread of Execution

Like many display applications, an IGC process performs initialization, waits for and processes events, destroys its objects, and then terminates.

9.2.3.1 Initialization

An IGC does the following during process initialization:

- Initializes OI, which will also initialize Xlib and obtain a connection to the X server. OI is passed the process argument list since it may be interested in some of the arguments such as -display <X Server hostname>.
- Parses the process arguments, as described in Section 9.2.1.1 and Section 9.2.2.1.
- If the IGC is a stand-alone process, creates an X window on which an IGC display layout will be built.
- Constructs an IGC_Impl object which will construct many of the other main IGC objects. Its constructor is passed many of the parsed process arguments, plus the handle to the X server.
- Constructs an EventDispatcher object. Its constructor is also passed the handle to the X server.
- Constructs a UI_WorkspaceManager object. If the IGC is a child process, then this object is passed the IPC target of the parent process, which is one of the process arguments.

9.2.3.2 Event Processing

After initialization, an IGC enters an event dispatch loop which is managed by the EventDispatcher object. Execution will remain inside this loop until the EventDispatcher receives an event indicating that the process should be terminated. If the IGC is a child process, this event will be an IPC message from the UI process. If the IGC is a stand-alone process, this event will be an <Alt Q> entered from the keyboard.

9.2.3.3 Termination

After exiting the event dispatch loop, the IGC will destruct the `UI_WorkspaceManager`, `IGC_Impl`, and `EventDispatcher` objects. The `IGC_Impl` object will then destruct most of the important IGC objects. The `EventDispatcher` needs to be destructed last, since other IGC objects need to refer to it, upon their destruction.

Destructing objects is important, since the IGC may have started some child extension processes which need to be terminated. Also, the IGC may have registered with the `NotificationServer` process and will need to un-register itself.

9.3 Overview of the Main Objects

This description of the objects that provide the IGC functionality will have the following format:

- *Process Context*: Identifies the number of these objects that exist at one time in an IGC process, and the object or module responsible for creating and destroying these objects. Also describes the important arguments used to construct these objects.
- *Inheritance*: Identifies the inheritance classes and why the inheritance relationship exists. If this object does not use inheritance, this section is omitted.
- *Modes*: Describes an IGC object's different modes of operation. If this object has only one mode of operation, then this section is also omitted.
- *Responsibilities*: Lists the main tasks for which this object is responsible.

9.3.1 EventDispatcher

9.3.1.1 Process Context

Only one `EventDispatcher` object exists in an IGC process. The process's main module is responsible for the construction and deletion of this object. A pointer to the X server connection is needed to construct this object since the `EventDispatcher` will be making Xlib calls directly.

9.3.1.2 Responsibilities

Listens for the following kinds of events by polling, and dispatches these kinds of events to multiple clients. Polling is inefficient! It would be preferable to use the UNIX `select` system call, but DMQ doesn't support file descriptors.

- IPC messages from the UI Process, Notification Server, and extension processes.
- OI/X events generated from the pop-up menus, and the color chooser dialog.
- X events such as button clicks, cursor movement, exposure, and key presses that occur inside a window managed by a `DisplayPanel` object. These events will be dispatched to `DisplayPanel` objects.
- Timer expirations which are used for looping and distinguishing a button click from a button drag. This object supports a timer granularity in milliseconds.

- Idle events that are dispatched whenever the IGC is not busy processing its other events. The FrameSeq object is a client of these kinds of events so that it can prepare non-current frames for display.

9.3.2 IGC_Impl

9.3.2.1 Process Context

Only one IGC_Impl object exists in an IGC process. The process's main module is responsible for the construction and deletion of this object. The following is needed to construct this object.

- A pointer to the X server connection which will be passed to the DisplayMgr object.
- The ID of the X window on which the IGC should build its layout. This will also be passed to the DisplayMgr object.
- Whether this IGC is to run in a large or small display area.

9.3.2.2 Inheritance

IGC_Impl inherits from the IGC_ABC class. This object is the implementation portion of a "Proxy" design pattern. The proxy portion of this pattern is the IGC object that lives inside the UI process. The IGC_ABC class ensures that the public interface between the proxy and the implementation remains consistent.

9.3.2.3 Responsibilities

- Creates and destroys the main IGC objects.
- Receives IPC requests from the UI process and passes them on to the other IGC objects that do the real work. This object sort of resembles the "Facade" design pattern.
- Caches and provides access to whether the IGC is running in a large or small display area.

9.3.3 UI_WorkspaceManager

9.3.3.1 Process Context

Only one UI_WorkspaceManager object exists in an IGC process. The process's main module is responsible for the construction and deletion of this object. The following is needed to construct this object.

- The IPC target of the parent UI process. If a null IPC target is passed in, then the IGC is running in stand-alone mode.
- The ID of this IGC assigned by the parent process.

9.3.3.2 Responsibilities

UI_WorkspaceManager's primary responsibility is to be a proxy to the UI process. In other words, any IGC object that needs to send information to the UI process will do so by calling a public method of this object. This object is qualified to send messages to the UI since it knows the UI's IPC address and the ID of this IGC.

9.3.4 LoopingMgr

9.3.4.1 Process Context

Only one LoopingMgr object exists in an IGC process. The IGC_Impl is responsible for the construction and deletion of this object.

9.3.4.2 Inheritance

LoopingMgr inherits from the TimerEventClient class so that it can receive a callback from the EventDispatcher when a timer expires.

9.3.4.3 Responsibilities

- Provides methods for querying and setting the parameters that control looping. See "Stepping and looping".
- Implements looping by managing a timer. Upon timer expiration, it tells the FrameSeq object which frame to display. See "Looping walk-throughs".

9.3.5 DisplayMgr

9.3.5.1 Process Context

Only one DisplayMgr object exists in an IGC process. The IGC_Impl is responsible for the construction and deletion of this object. The following is needed to construct this object.

- A pointer to the X server connection which is needed by this object, and other IGC objects who will be making Xlib calls.
- The ID of the X window on which the IGC will build its layout. This object will use this window, but will not destroy it since it did not create it.

9.3.5.2 Modes

Most often, an IGC will run with a single display panel. But sometimes it is useful to divide the display into four equal portions, such as for displaying four different tilts of radar.

The DisplayMgr implements a single panel layout by constructing a single DisplayPanel object, passing the main layout window as an argument. When switching to a four-panel layout, that DisplayPanel object will be destructed.

The implementation of a four-panel layout is slightly more complicated. This object will create four sub-windows on top of the main layout window. These windows will not be transparent, so the main layout window will not be visible. These four windows will equally subdivide the area of the parent window, and will have the same depth (8 or 24 bits) as the parent window. This object will then create four DisplayPanel objects, each receiving one of those four sub-windows as a constructor argument. When switching to a single-panel layout, the four sub-windows and the four DisplayPanel objects will be destructed.

9.3.5.3 Responsibilities

- Provides methods for querying and changing the display layout (single- or four-panel).
- Provides a method for altering the display mode of a DisplayPanel object (8- or 24-bit mode).
- Manages cursor resources. The IGC supports the following kinds of cursors:
 - ◆ Pie Cursor used when preparing frames in the background.
 - ◆ Busy Cursor used when the IGC is not responsive to user input.
 - ◆ Interactive drawing cursors.
 - ◆ Linked cursors which are used only during four panel layout.
- Manages font resources for legends and sampling. Font sizes are dependent on the IGC magnification and the width and height of the display panel(s).
- Maintains the image display parameters (fade/dim).
- Allocates the color set used for graphics/legends. The initial values of this set are read in from the file, dataMgmt/fga.config, but this set can be changed dynamically since the user can specify colors of a graphic overlay. All display panels will be using the same color set, and will be using the same X colormap for their graphic window. The DisplayMgr keeps track of which colors in the set have been used to display an overlay.

9.3.6 DisplayPanel

9.3.6.1 Process Context

Either one or four of these DisplayPanel objects exist at one time during the life of the IGC process, depending on the mode of the DisplayMgr (see Section 9.3.5.2). The DisplayMgr object is responsible for the construction and deletion of these objects. The ID of the X window that this panel object will use as the image window is required for construction.

9.3.6.2 Inheritance

Inherits from the DisplayAreaEventClient class so that it can receive a callback from the EventDispatcher whenever an Xlib event occurs inside the panel windows.

Inherits from the TimerEventClient class so that it can receive a callback from the EventDispatcher when a timer expires. This is used to distinguish button clicks from button drags.

Inherits from the DisplayPanelActionsClient class so that it can receive a callback from the PopUpActionMgr whenever the user selects an option from a pop-up that pertains to this display panel.

9.3.6.3 Modes:

This object can run using either a pseudo-color (8-bit) display algorithm or a true color (24-bit) display algorithm.

In 8-bit mode, this object will use two X windows mapped to the same screen space. The bottom window is the same window that was passed in at construction time. This window will be used to display images using a private X colormap, utilizing all 256 colors. This window is

created with the visual that uses the underlay (image) planes of the H-CRX24 hardware. The top window will be created by this object, and will be a child window of the bottom window. In order to see the bottom window, the top window will have a transparent background. This window will be used to display graphic products (non-images) and legends. This window is created with the visual that uses the overlay planes and the transparency feature of the H-CRX24 hardware.

Here are some of the advantages of using 8-bit mode with a two window architecture:

- Image and graphic colors don't have to be shared between a single 256-color set. This makes image combination a lot less complicated.
- Graphics can be drawn independently without having to refresh the image.

However, there are these disadvantages:

- Creates a dependence on the graphics hardware to provide transparency and overlay planes.
- In order to achieve fast looping, two sets of pixmaps must be allocated (each set containing a pixmap for each frame). Substantial pixmap allocation can cause memory resource problems for the X server.

In 24-bit mode, this object will use the window passed in at construction time as the only panel window. Thus, all graphics and legends will be displayed directly on top of the image in the same window.

Here are the advantages of the 24-bit mode.

- Many different images with different color requirements can be displayed simultaneously in different display panels that are using true color.
- Image combo looks more realistic since an 8/8-bit combo is used instead of a 4/4-bit combo.

And there are also disadvantages:

- 24-bit pixmaps require four times as much X server memory as 8-bit pixmaps.
- Image combos take much longer to fade or dim than image combos displayed in 8-bit display panels.

9.3.6.4 Responsibilities

- Displays depictable output, sample text, and legends.
- Provides a mechanism for selecting a legend by clicking on it.
- Handles timer and display area events from the event dispatcher.
- Maintains sets of pixmaps (each set containing a pixmap for every active frame), so that graphics and images can be displayed quickly for fast looping speeds (up to 20 frames/second).

9.3.7 ViewMgr

9.3.7.1 Process Context

Only one ViewMgr object exists in an IGC process. The IGC_Impl is responsible for the construction and deletion of this object.

9.3.7.2 Responsibilities

- Manages the scale and map projection used by all loaded products.
- Manages the current view consisting of:
 - ◆ Zoom factor
 - ◆ Display center
 - ◆ Magnification
 - ◆ Default relative picture density. Each overlay has the ability to over-ride this value.
- Manages the coordinate conversion between screen coordinates and lat/lon which is used for sampling and by the interactive depictables.

9.3.8 FrameSeq

9.3.8.1 Process Context

Only one FrameSeq object exists in an IGC process. The IGC_Impl is responsible for the construction and deletion of this object.

9.3.8.2 Inheritance

Inherits from IdleStateClient class so that it can receive a callback from the EventDispatcher whenever there is no other events to process.

Inherits from OverlayActionsClient class so that it can receive a callback from the PopUpActionMgr whenever the user selects an option from a pop-up that pertains to a particular overlay.

9.3.8.3 Responsibilities

- Loads and unloads all products (images, graphics, background maps, and combo images). It implements the load by organizing depictable objects with matching times into frames by managing Frame, DepictSeq and DepictTuple objects.
- Initiates the display and printing of the current frame.
- Prepares non-current frames for display.
- Maintains the following data structures:
 - ◆ *Tuple table*: A hash table (Dict) that contains indices to DepictTuple objects. The index is built from the DepictSeq object pointer, and the data time of the tuple's depictable. In the case of combo image tuples, the index contains two data times.
 - ◆ *Frame list*: A list (SeqOf) of frame objects.
 - ◆ *Load order list*: A list of DepictSeq objects. This list maintains the order in which products were loaded and will be displayed.
- Keeps extension processes up to date with what is being displayed.

- Stores and retrieves a complete display context from a file.
- Provides a mechanism for changing the number of displayed frames and the frame that is currently displayed (current frame).

9.3.9 Frame

9.3.9.1 Process Context

During the entire life of the IGC process, the number of Frame objects that exist is equal to the maximum frame count (which is currently 32, for a large IGC). The FrameSeq object is responsible for the construction and deletion of these objects.

9.3.9.2 Responsibilities

- Maintains a list of DepictTuple objects that all have matching times in common. This object is not responsible for the creation and deletion of tuples, it just refers to them. There is no limit on the number of tuples/depictables that a frame can contain.
- Provides access to the list of DepictTuple objects for purposes of display, legend selection, sampling, and printing.
- Provides a mechanism for adding and removing tuples from the frame. The order of the tuple list is important since it is also the order in which depictables will be displayed and printed. Thus, map background tuples/depictables are first on the list, so all other graphics will be drawn on top. A frame can contain multiple image tuples/depictables as long as the tuples are being displayed in different display panels.
- Maintains a nominal frame time, which all depictables contained in this frame should match.
- Keeps track of which tuple is currently being prepared in the background, and whether any tuples need to be prepared in the background.

9.3.10 GraphicDepictTuple

9.3.10.1 Process Context

The number of GraphicDepictTuple objects that exist in an IGC_Process depends on the number of graphic (non-image) products that are loaded. The number of tuples for each graphic product is equal to the number of unique times from the time-matched inventory. The FrameSeq object is responsible for the construction and deletion of these objects. The following is needed to construct these objects:

- A list of data times that will be passed to the depictable. Most depictables require only one time, but there are some that require a list of times, such as Time-Height Cross Section depictables.
- A pointer (only for reference) to a GraphicDepictSeq object.

9.3.10.2 Inheritance

GraphicDepictTuple inherits from the DepictTuple class because graphics and images have a number of characteristics in common, and a few that differ. Also, it is convenient for the Frame and FrameSeq objects to think of just DepictTuple objects, rather than GraphicDepictTuple or ImageDepictTuple objects.

9.3.10.3 Responsibilities

- Creates, destroys, and maintains an X pixmap, which has the same width and height as the display panels. The depth of this pixmap is 1 bit.
- Creates, destroys, and maintains a GraphicDepictable object. This object uses the DepictableServer to actually create the object, since only the server knows which concrete class to use.
- Creates, destroys, and maintains a TextString object containing the legend which is the product name plus the data time of the depictable.
- Maintains a struct containing the IGC zoom and density parameters of the last time this object's depictable was rendered.
- Maintains a pointer and provides access to a GraphicDepictSeq object. This object contains information about this graphic overlay.
- Maintains a count of the number of Frame objects that point to this object (called the usage count).

9.3.11 ImageDepictTuple

9.3.11.1 Process Context

The number of ImageDepictTuple objects that exist in an IGC_Process depends on the number of image products that are loaded. The number of tuples for each image product is equal to the number of unique times from the time-matched inventory. The FrameSeq object is responsible for the construction and deletion of these objects. The following is needed to construct these objects:

- Two lists of data times that will be passed to the two depictables. If this is for a single image, then only one list of times is really necessary. Most depictables require only one time, but there are some that require a list of times, such as Time-Height Cross Section depictables.
- A pointer (for reference only) to a ImageDepictSeq object.

9.3.11.2 Inheritance

ImageDepictTuple inherits from the DepictTuple class because graphics and images have a number of characteristics in common, and a few that differ. Also, it is convenient for the Frame and FrameSeq objects to think of just DepictTuple objects, rather than GraphicDepictTuple or ImageDepictTuple objects.

9.3.11.3 Responsibilities

- Creates, destroys, and maintains an X pixmap, which has the same width and height as the display panels. The depth of this pixmap (either 8 or 24 bits) depends on the mode of the DisplayPanel object that will display this image.
- Creates, destroys, and maintains one or two ImageDepictable objects depending on whether this is a single or a combo image. This object uses the DepictableServer to actually create the object, since it knows which concrete class to use.
- Creates, destroys, and maintains a TextString object containing the legend which is the product name plus the data time of the depictable. In the case of a combo image, there will be two product names and data times.

- Maintains a struct containing the IGC zoom and density parameters of the last time this object's depictable was rendered.
- Maintains a pointer and provides access to an ImageDepictSeq object. This object contains information about this image overlay.
- Maintains a count of the number of Frame objects that point to this object (called the usage count).

9.3.12 GraphicDepictSeq

9.3.12.1 Process Context

The number of GraphicDepictSeq objects that exist in an IGC_Process is the number of graphic (non-image) products that are loaded. The FrameSeq object is responsible for the construction and deletion of these objects. The following is needed to construct these objects:

- Load information such as the load mode, forecast time, and load time.
- A Set object containing display panel identifiers indicating which display panels will be used to display depictables that are associated with this DepictSeq.
- The DepictKey for this graphic product.
- A flag indicating whether this product is visible (displayed) or not.
- A value indicating what the relative picture density for this overlay should be.

9.3.12.2 Inheritance

Inherits from the DepictSeq class because graphics and images have a number of characteristics in common, and a few that differ. Also, it is convenient for the FrameSeq objects to think of just DepictSeq objects, rather than GraphicDepictSeq or ImageDepictSeq objects.

9.3.12.3 Responsibilities

- Provides interfaces for setting and querying various attributes about this graphic overlay. Some of these attributes are:
 - ◆ Load Attributes (time matching mode, forecast time, inventory time);
 - ◆ Relative picture density;
 - ◆ Whether the overlay is displayed or toggled off (visibility);
 - ◆ The set of display panel identifiers into which depictables of this sequence will be drawn;
 - ◆ Information about the product obtained from dataMgmt's depictable information file (depictInfo.txt);
 - ◆ Whether this product is slot loaded;
 - ◆ Whether this product is an interactive overlay, and whether it is editable or not;
 - ◆ The color of the overlay and the legend, (obtainable in both RGB format, and as a color table index or pixel value).
- Determines which pop-up actions make sense for this graphic overlay.

9.3.13 ImageDepictSeq

9.3.13.1 Process Context

The number of ImageDepictSeq objects that exist in an IGC_Process is the number of image products that are loaded. The FrameSeq object is responsible for the construction and deletion of these objects. The following is needed to construct these objects:

- Load information such as the load mode, forecast time, and load time. If the image is a combo, then two sets of load info are passed in.
- A Set object containing display panel identifiers indicating which display panels will be used to display depictables that are associated with this DepictSeq.
- The DepictKey for this graphic product. If the image is a combo, then two keys are passed in.
- A flag indicating whether this product is visible (displayed) or not.
- A value indicating what the relative picture density for this overlay should be.

9.3.13.2 Inheritance

Inherits from the DepictSeq class because graphics and images have a number of characteristics in common, and a few that differ. Also, it is convenient for the FrameSeq objects to think of just DepictSeq objects, rather than GraphicDepictSeq or ImageDepictSeq objects.

9.3.13.3 Responsibilities

- Provides interfaces for setting and querying various attributes about this image overlay. Some of these attributes are:
 - ◆ Load Attributes (time matching mode, forecast time, inventory time).
 - ◆ Relative picture density. Raster, gridded, and radial images don't use relative picture density, but deep graphic images, such as the perspective profiler image, do.
 - ◆ Whether the overlay is displayed or toggled off (visibility).
 - ◆ The set of display panel identifiers into which depictables of this sequence will be drawn.
 - ◆ Information about the product(s) obtained from dataMgmt's depictable information file (depictInfo.txt).
 - ◆ Whether this product(s) is slot loaded.
 - ◆ The color of the legend, (obtainable in both RGB format, and as a color table index or pixel value). By default, images use white as their legend color, but that color can be altered by the user.
- Determines which pop-up actions make sense for this image overlay.
- Creates, destroys, and manages one or two ColorTable objects, depending on whether this is a single or combo image. Provides methods for querying and setting these objects. These objects are capable of writing to an X colormap, and also implement the color table combination algorithms discussed in Section 9.5.4.3, "Rendering an 8-bit image combo".

9.4 Walk-through of Loading a Product

9.4.1 Load of a product to an empty display

Initial Situation: No time varying products are loaded, but static products (maps /topo image, etc.) may be loaded and displayed.

1. User selects product from the user interface of the fxa process. An IPC message is sent containing a depict key, load mode, inventory time, forecast time, and a combine flag. The IGC_Impl object receives this message and then calls FrameSeq::load.
2. Checks for an implicit clear. Product may require a different map projection than the one being used. If so, then all the existing products are unloaded.
3. If combine flag is not set, and the new product is an image, unloads any images that are being displayed in the panel in which the new image will be displayed.
4. Obtains an inventory from dataMgmt.
5. Calls the routine that time-matches primary sequences, passing the preferred frame count.
6. Assigns each time on the time match list to the corresponding frame's nominal frame time.
7. Copies the pointers to the static (map background) tuples in the first frame to the rest of the frames, and increases the use count of these static tuples.
8. Creates a new DepictSeq object, and adds it to the beginning of the load list. Passes to the DepictSeq constructor the load information that was received from the UI process.
9. For each time in the time match list, checks to see if there is a DepictTuple object with the same time and sequence. If not, creates one, and adds it to the tuple table. Then adds a pointer to that tuple to the corresponding frame, and increases the tuple's use count.
10. Analyzes the image color table requirements for each display panel. If the IGC layout is single panel, then this step is skipped. If all the panels have the same color table requirement, then nothing further needs to be done. If there is more than one requirement, the requirement with the most panels, (or in the case of a tie, the combo requirement) will be used to determine the panels that can be displayed in 8-bit pseudo color. All the other panels will be converted to 24-bit true color.
11. Redraws the current frame, and marks all other frames as needing redrawing.

9.4.2 Load of a product to a non-empty display

Initial Situation: One or more time varying products have already been loaded.

1. Follows steps 1 - 4 described in Section 9.4.1.
2. Calls the routine that time matches overlay sequences, passing the nominal frame times.
3. Tries to find a DepictSeq object with the same load attributes. If not, create one, and add to the load list before all the static products but after all the existing time-varying products.
4. Follows steps 9 - 11 described in Section 9.4.1.

9.4.3 Image combo load

Same steps as the single image, except that an existing image is not unloaded. Rather, its DepictTuple and DepictSeq objects are expanded to include the new image as the second image.

9.4.4 Multi-load

A Multi-load is a way of loading multiple products with one load command. This mechanism is used for families, radar combos, and four panel configurations.

When a multi-load command arrives from the UI, the IGC uses that (single) key to look up the following information in a dataMgmt table:

- which products should be loaded;
- into which panels the products should go;
- whether the products should initially be displayed or toggled off.

The IGC then loads each product, but defers the display and rendering until all the products are processed.

9.4.5 Product update (auto-update)

- When a DepictSeq object for a time varying product is created, it sends a message to the NotificationServer process, asking for updates. When this object is destructed, it un-registers with the NotificationServer.
- When a new time for a loaded product arrives, the NotificationServer sends a message to the IGC_process, containing the depict key and that new time.

The next four steps are also executed whenever the user changes the preferred frame count.

- The FrameSeq object removes all DepictTuple objects from the frames, but does not delete those objects from the tuple table, or the DepictSeq objects from the load order list.
- For every product in the list, obtain a new inventory, redo the time matching, and place tuples back in the frame object according to the time matching. Some new tuples may need to be created, and some tuples will be not be re-added to the frames.
- The frame sequence then deletes all unused DepictTuple and DepictSeq objects.
- The current frame is displayed, and all other frames are marked as needing redrawing.

9.5 Walk-through of Displaying a Product

9.5.1 Displaying the current frame

The current frame needs to be displayed whenever any of the following occurs:

- Loading or unloading a product.
- Arrival of an auto-update.
- Adjustment of the preferred frame count, but only if the current frame changes.
- Toggling a product's visibility
- Changing the display attributes of one or more products. Display attributes include:
 - ◆ magnification
 - ◆ picture density
 - ◆ color
 - ◆ texture (line style or line width)
- Modification of an interactive overlay by an extension process.

- Clearing the display.
- Change in image fade or dim for an image displayed in a 24-bit panel. For 8-bit images, display is required only if the legends change, since fade and dim are done for 8-bit by modifying the color table.

To display the current frame, the following steps are taken:

1. Every tuple in the current frame is examined. Image tuples are processed first followed by graphic tuples. This is because if the display panel is 24 bits, the image needs to be drawn first, so graphics will be drawn on top of it. For 8-bit panels, the ordering is not important, since two different windows are used.
2. For each tuple that is visible (the tuple's DepictSeq object maintains the visibility flag), the following is done:
 - ◆ Renders the tuple, if necessary. The ViewMgr's zoom and density parameters are compared with the tuple's zoom and density parameters the last time it was rendered. If they are the same, then this tuple does not need to be rendered. The details of rendering a tuple are covered in Section9.5.3 and Section9.5.4.
 - ◆ Copies the tuple's drawable to every affected display panel. (The DepictSeq object also maintains the panel information).

For an image drawable, the DisplayPanel object will take that drawable directly and assign it as the image window's background pixmap. The panel object will also receive a pointer to the ColorTable object which is instructed to load the image's 256 colors into the image window's private colormap. Loading the color table is not necessary for 24-bit panels.

For a graphic drawable, the 1-bit pixmap (bitmap) is copied into the DisplayPanel object's backing graphics pixmap corresponding to the current frame. This copy is done with a stipple fill in order to preserve what was drawn by previous tuples. The color of the tuple is applied when the tuple's pixmap is copied into the panel's pixmap.

3. Once all the tuples have been examined, every DisplayPanel object will do the following:
 - ◆ Obtain a list of tuple objects from the current frame. It will display the legends of each tuple, with the first legend displayed in the lower right corner of the display panel and the others stacked on top. If the tuple is visible, then the legend is displayed in the color of the tuple, otherwise the legend is displayed in gray.
 - ◆ Assign the graphics backing pixmap corresponding to the current frame to the graphics window's background pixmap.

9.5.2 Preparing frames in the background

As mentioned earlier, one goal of the IGC is to support fast looping speeds, up to 20 frames per second. In order to achieve this goal, each frame's picture needs to be rendered into a pixmap (or two pixmaps in the case of 8-bit display panels). The advantage of using a pixmap is that it can be displayed into a window rather quickly, and it can be filled off screen, behind the user's back, while the IGC process is waiting for events. If the user wants to step or loop to a frame that hasn't been rendered to a pixmap

yet, then the user will notice a delay. The length of the delay depends on how many products are in the frame, and how long it takes for each product's depictable to produce its picture. So, the trade-off is to perform as much rendering as possible while the workstation is idle, but still respond promptly to the user's actions.

Here is how those pixmaps are prepared in the background.

1. When the EventDispatcher doesn't have any X, IPC, or timer events to process, it calls a method in the FrameSeq object. This method counts the total number of frames that need to be redrawn, then figures out which pie cursor the DisplayMgr object should display based on the ratio between frames needing redrawing and the total number of frames. If no frames need to be processed, then the default cursor is displayed.
2. One of those Frame objects is then instructed to prepare a single tuple as described in Section 9.5.1 step 2. However, instead of copying the tuple's bitmap or pixmap to the X window, it is copied to a pixmap corresponding to the frame that is being processed.
3. If that frame has processed its last tuple, then the legends for that frame are drawn into the pixmap corresponding to that frame, and the frame is marked as no longer needing redrawing.

9.5.3 Rendering a graphic tuple

Graphic rendering involves passing a drawable (which can be thought of as a blank canvas) and having the depictable (the artist) paint a graphic representation of the data into the canvas. Since the drawable is only one bit deep, the depictable does not have to deal with color. The following steps are taken by the GraphicDepictTuple object:

1. Tells the DisplayMgr object to change the cursor to its busy representation (wrist-watch).
2. Clears the drawable by setting all the pixels to 0.
3. Instructs the depictable to paint into the drawable by passing the depictable the following information:
 - ◆ data needed to use Xlib, such as the pointer to the display connection, the drawable, and the graphics context;
 - ◆ the extent (width and height) of the drawable;
 - ◆ zoom, picture density, and line texture data obtained from the ViewMgr object.;
 - ◆ scale and map projection data obtained from the ViewMgr object.
4. Restores the cursor.

9.5.4 Rendering an image tuple

Image rendering is similar to graphic rendering, except that the canvas is a byte (8-bit) buffer instead of an X drawable. Each ImageDepictable object (there might be two in the case of image combo) will fill its byte buffer with each pixel representing a color index between 0 and 255. The display panel will expand and/or combine the buffers, and then instruct Xlib to copy the buffer into the tuple's drawable. The details differ depending on whether it is a single or combo image, and whether the display panel is 24 or 8 bits deep.

9.5.4.1 Rendering a single 8-bit image

The following steps are performed by an ImageDepictTuple object:

1. Tells the DisplayMgr object to change the cursor to its busy representation (wrist-watch).
2. Allocates a buffer which is an array of bytes (char) whose size is the width * height of the tuple's drawable.
3. Instructs the ImageDepictable object to paint into the byte buffer by passing the following information:
 - ◆ a pointer to the buffer;
 - ◆ the extent (width and height) of the drawable;
 - ◆ zoom, picture density, and line texture data;
 - ◆ scale and map projection data.
4. Passes the byte buffer to Xlib, to copy it into the tuple's 8-bit pixmap.
5. De-allocates the memory allocated in step 2.
6. Restores the cursor.

9.5.4.2 Rendering a single 24-bit image

The following steps are performed by an ImageDepictTuple object:

1. Executes the first three steps described in "Rendering a single 8-bit image."
2. Allocates a second byte buffer, which uses 4 bytes for every pixel. Thus, the size of this buffer is 4 times that of the buffer used as the depictable's canvas.
3. For each byte of the buffer filled by the Image Depictable object, the tuple looks up the red, green, and blue intensity values using the image color table and pixel index represented by that byte. The three color components are then multiplied by the image brightness value, a float between 0 and 1 that is maintained by the DisplayMgr object. Four bytes are copied into the second byte buffer: a blank (0) byte, and one byte for each of the red, green, and blue values.
4. Passes the second buffer to Xlib, so it can be copied into the tuple's 24-bit pixmap.
5. De-allocates both byte buffers.
6. Restores the cursor.

9.5.4.3 Rendering an 8-bit image combo

The following steps are performed by an ImageDepictTuple object:

1. Tells the DisplayMgr object to change the cursor to its busy representation (wrist-watch).
2. Allocates a buffer which is an array of bytes (char) whose size is the width * height of the tuple's drawable.
3. Creates a ComboDepictable object, by passing pointers to the two ImageDepictable objects owned by this tuple. This step needs to be done only when rendering for the first time or whenever one of the image depictable objects is replaced.
4. Instructs the ComboDepictable object to paint into the byte buffer by passing the depictable the following information:
 - ◆ a pointer to the byte buffer;
 - ◆ the extent (width and height) of the drawable;
 - ◆ zoom, picture density, and line texture data;
 - ◆ scale and map projection data.

It is important to understand how the combo depictable fills its byte buffer in order to fully understand the image combination algorithm. For each pixel or byte, the combo depictable asks each of its two depictables for the color value (an 8-bit number between 0 and 255) of that pixel. The two pixel values are quantized into one value using the algorithm depicted in

Figure 9.1. That combined pixel value is then written at the appropriate position in the byte buffer.

5. Passes the byte buffer to Xlib, so it can copy the buffer into the tuple's 8-bit pixmap.
6. De-allocates the memory allocated in step 2
7. Restores the cursor.

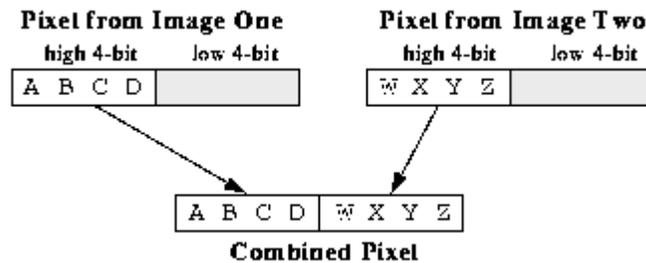


Figure 9.1 Combining two 8-bit pixels into one.

9.5.4.4 Rendering a combo 24-bit image

The following steps are performed by an ImageDepictTuple object:

1. Tells the DisplayMgr object to change the cursor to its busy representation (wrist-watch).
2. Allocates three byte buffers which are arrays of bytes (char). Two of these byte buffers will be used as a drawing canvas for the tuple's two image depictables. Their sizes are the (width * height) of the tuple's drawable. The third byte buffer, four times the size of the others, will be used to send the combined data to X.
3. Instructs both Image Depictable objects to paint into their byte buffers by passing the depictables the following information:
 - ◆ a pointer to the byte buffer;
 - ◆ the extent (width and height) of the drawable;
 - ◆ zoom, picture density, and line texture data;
 - ◆ scale and map projection data;
 - ◆ a flag indicating whether the color bar should be displayed left justified. One depictable will have this set to true, while it will be false (right side) for the other.
4. For each byte of the buffers filled by the depictables, the tuple looks up the red, green, and blue intensity values using the image color table and pixel index represented by that byte. The two colors are then combined into a new color using a linear interpolation based on the image fade value, a float between 0 and 1 that is maintained by the DisplayMgr object. The three components of the new color are then multiplied by the image brightness value. Four bytes are copied into the third byte buffer: a blank (0) byte and one byte for each of the red, green, and blue values.
5. Passes this third buffer to Xlib, to be copied into the tuple's 24-bit pixmap.
6. De-allocates the buffers allocated in step 2.
7. Restores the cursor.

9.5.5 Dimming and fading images

Dimming refers to adjusting the brightness of a single or combined image. Fading is relevant only to combined images and refers to the adjusting of the contribution of each image. A fade value of 0.5 implies that both images are being displayed equally. The user can change these values using the Image Controls Dialog which is managed by the fxa (UI) process.

The new fade and combo values are sent by the fxa, received by the IGC_Impl object and handled by the DisplayMgr object.

If the display panels are 24 bits deep, then the images in those panels will need to be re-rendered, following the recipe outlined in Section 9.5.4.4. If the display panels are 8 bits deep, then the combined color table needs to be recalculated as described in the following section. A display can have a mixture of 8- and 24-bit display panels.

9.5.5.1 Calculating a combined color table

A color table object includes a method that combines itself with another color table object and then loads the resulting color table into the X colormap of a display panel's image window. This is accomplished by the following algorithm:

1. Two subsets of 16 colors are computed from each color table. The color table is subdivided equally into 16 buckets. The most colorful color from each bucket is added to the subset. Three factors are considered in determining whether a color is more colorful:
 - ◆ the color's brightness (sum of all three red, green, and blue components);
 - ◆ the amount it differs from gray, white, or black;
 - ◆ the difference between this color and the color from the previous bucket.
2. Each entry in the combined color table is a linear interpolation of one subset color with a color from the other subset. The interpolation is calculated using the image fade value, a float between 0 and 1 that is maintained by the DisplayMgr object. The scheme for building this color table is depicted in Figure 9.2.
3. Applies the image brightness level to every entry in the combined color table by multiplying the brightness with each color's red, green, and blue component.

Figure 9.2 Arrangement of a combined color table (entries 16 - 239 omitted)

0:	SubsectA [0] with SubsectB [0]	240:	SubsectA [15] with SubsectB [0]	
1:	SubsectA [0] with SubsectB [1]	241:	SubsectA [15] with SubsectB [1]	
2:	SubsectA [0] with SubsectB [2]	242:	SubsectA [15] with SubsectB [2]	
3:	SubsectA [0] with SubsectB [3]	243:	SubsectA [15] with SubsectB [3]	
4:	SubsectA [0] with SubsectB [4]	244:	SubsectA [15] with SubsectB [4]	
5:	SubsectA [0] with SubsectB [5]	245:	SubsectA [15] with SubsectB [5]	
6:	SubsectA [0] with SubsectB [6]	246:	SubsectA [15] with SubsectB [6]	
7:	SubsectA [0] with SubsectB [7]	247:	SubsectA [15] with SubsectB [7]	
8:	SubsectA [0] with SubsectB [8]	• • •	248:	SubsectA [15] with SubsectB [8]
9:	SubsectA [0] with SubsectB [9]		249:	SubsectA [15] with SubsectB [9]
10:	SubsectA [0] with SubsectB [10]		250:	SubsectA [15] with SubsectB [10]
11:	SubsectA [0] with SubsectB [11]		251:	SubsectA [15] with SubsectB [11]
12:	SubsectA [0] with SubsectB [12]		252:	SubsectA [15] with SubsectB [12]
13:	SubsectA [0] with SubsectB [13]		253:	SubsectA [15] with SubsectB [13]
14:	SubsectA [0] with SubsectB [14]		254:	SubsectA [15] with SubsectB [14]
15:	SubsectA [0] with SubsectB [15]		255:	SubsectA [15] with SubsectB [15]

9.6 Walk-throughs of Other IGC Operations

9.6.1 Zooming and roaming

Zooming is when the user specifies both a new display center and a zoom factor, while roaming is when the user specifies only a new display center. Although the user interfaces are quite different, the implementations of zooming and roaming are essentially the same.

9.6.1.1 Zooming user interface

The simplest user interface for zooming is positioning the cursor in the display panel where the display center should be, and clicking button 1 to zoom out or button 2 to zoom in. In four-panel mode, even though only one of the panels receives the button click event, all four panels will respond to the zoom. Here's what happens behind the scenes:

1. The EventDispatcher object receives the Xlib event, and determines to which display panel object to send the event, by examining the window where the event occurred.
2. The DisplayPanel object then tells the ViewMgr whether a zoom out or zoom in was selected, and also what the new display center should be.
3. The ViewMgr then executes the zoom operation using the procedure described in Section 9.6.1.3.

A more useful interface is through the context sensitive pop-up menus. The user can pop up a cascading menu that presents a choice of zoom factors specified in terms of the width of the display in kilometers. The new display center is where the user pops up the menu. The advantage of this interface is that the user can go directly to the desired zoom factor. With the button click interface, it may take several display refreshes (which can be time consuming) before the desired zoom factor is obtained.

9.6.1.2 Roaming user interface

Roaming is initiated by the user by holding down button 2 at a certain position inside one of the display panel. As the user moves the cursor while holding the button, the picture (both the image and graphics/legends) follows the cursor, only inside the selected display panel. As the picture moves off the screen, the area where the picture used to be is drawn in black (see Figure9.3). Once the user releases the button, all the display panels are redrawn with the new display center. At this point, the roaming panel will have its black areas filled in with the new picture.

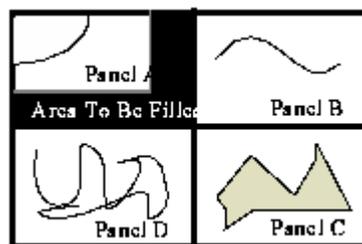


FIGURE 9.3 Example of roaming Panel A

Here's what happens behind the scenes:

1. The DisplayPanel object who has the cursor gets the button press event from the EventDispatcher object.
2. If the button is button 2, the display panel initiates a timer to expire in 0.5 seconds. If the button is still held down when the timer expires, then roaming begins, and the roaming start position is recorded.
3. The DisplayPanel object receives motion notify events as the user moves the cursor.
4. The window coordinate of where the upper left corner of the panel's pixmaps will now be placed is computed according to the new cursor position. The pixmaps (image and graphic) are recopied into the window at that new coordinate.
5. Two rectangles, which are the difference between the new position and the old position of the pixmaps, are computed. Those two rectangles are drawn in the panel windows using the background color.
6. Steps 3-5 are continued until the DisplayPanel object receives the button press event from the Event Dispatcher. The new display center, which is the (roaming end position - roaming start position) + the center of the display panel, is computed. The new display center is then passed to the ViewMgr object which then executes the roam operation using the procedure described in the next section.

9.6.1.3 Zooming and roaming implementation

1. Tell the DisplayMgr object to change the cursor to its busy representation (wrist-watch).
2. Convert the display center to 1:1 coordinate space by dividing it by the current zoom factor and adding it to the display origin (upper left corner position in 1:1 coordinate space).
3. The ViewMgr maintains a list of zoom factors which are floating point values that are initialized from the fxa.config file. The ViewMgr also keeps track of the current zoom factor. If the user is zooming in, then select the next largest zoom factor. If the user is zooming out, select the next smallest zoom factor. If the user is roaming, then use the current zoom factor.
4. Convert the panel's extent (width and height) in 1:1 coordinate space using the new zoom factor. If the display center is outside of the panel's extent, then move it inside.
5. Calculate the display origin in 1:1 space, and save that coordinate, the display center, and the new zoom factor as the current view.
6. Redraw the current frame, and all other frames marked as needing redrawing. Since the current view has changed, all tuples should be re-rendered, which means that each depictable will need to re-fill its tuple's drawable. See Section 9.5.1, "Displaying the current frame".
7. Restore the cursor.

9.6.2 Toggling an overlay's visibility

The user is able to control whether a product is to be displayed, without having to unload or reload it. Basically, the IGC supports this by providing a mechanism for the user to select an overlay by clicking on its legend. The visibility flag inside the DepictSeq object associated with that overlay is then adjusted, and the display is refreshed. Here is a more detailed walk-through of this procedure:

1. The user clicks on one of the displayed legends in the lower right corner of one of the display panels with mouse button 1.

2. The EventDispatcher object receives the Xlib button release event, and determines to which DisplayPanel object to send the event, by examining the window where the event occurred.
3. The DisplayPanel object then checks to see if the legends are being displayed. If not, then this event is interpreted as a zoom out and is handled as described in Section 9.6.1.
4. Obtains a list of the tuple objects that are loaded into the current frame. If only legends for background maps are being displayed, then all non-map tuples are removed from the list. Likewise, if product legends are being displayed, then all map tuples are removed from the list. Also, if a tuple is not being displayed in the display panel where the click took place, it is removed from the list.
5. Uses the Y coordinate of the button release event to determine which legend is underneath the cursor. If the Y position is above the top-most legend, then this event is interpreted as a zoom out.
6. Uses the X coordinate of the button release event to determine if the button click is to the left of the first character of the legend selected in the previous step. If it is to the left, then this event is interpreted as a zoom out. However, if it is to the right of the first character, then we found the legend under the button click. That legend has a DepictTuple object associated with it.
7. The DepictSeq object, obtained from the selected tuple object, toggles its internal visibility flag.
8. Redraws the current frame, and all other frames are marked as needing redrawing. If the selected overlay is turned off, then all tuples that have a pointer to the DepictSeq object associated with that overlay will be skipped during the copying of a tuple's drawable to a display panel. However, the legends of those tuples will still be displayed, but drawn in a gray color. For more details, see Section 9.5.1.

9.6.3 Sampling

Sampling is an IGC feature that displays text describing the meteorological values of the displayed products at the point where the cursor is. Thus, every time the cursor is moved while sampling is active, new values are displayed. Not every product currently supports sampling, although METAR plots and most images do. The text appears as close as possible to the cursor, and each product that supports sampling has a single line of text, drawn in the color of the product's legend. Here is how that feature is supported behind the scenes:

1. The DisplayPanel object who has the cursor gets the button press event from the EventDispatcher object.
2. If the button is button 1, the DisplayPanel object initiates a timer to expire in 0.5 seconds. If the button is still held down when the timer expires, then sampling is made active.
3. When sampling becomes active, the FrameSeq object is instructed to disable background processing. The cursor is not as responsive while tuples are being rendered in the background.
4. While sampling is active, the DisplayPanel object will do the following every time it receives a motion notify event from the X server (which is generated every time the user moves the cursor).
 - ◆ Tells the ViewMgr object to convert the display coordinate from the motion notify event to a lat/lon coordinate.
 - ◆ For every visible tuple in the current frame, instructs that tuple to call its depictable's "interrogate" method, passing the lat/lon coordinate. If that tuple is for a combo image, it may have several depictable objects to call. That method will return a TextString object. If that object does not contain a null string, then registers that string and the overlay color with every DisplayPanel object that is displaying this tuple.

- ◆ Instructs every DisplayPanel object to erase the previous sample text. This is done by copying the area to be erased from the backing graphics pixmap to the panel's graphics window.
 - ◆ Instructs every DisplayPanel object to display all the registered sample text directly to the panel's graphics window. The upper left corner of the sample text is computed so that it appears directly below, and slightly the right of the cursor. However, if this position does not allow the entire sample text to be displayed in the panel, that position is adjusted to the left or above.
5. When the user releases the mouse button, the DisplayPanel object with the cursor will receive the button release event from the EventDispatcher. If sampling is active, then it is made inactive, by setting an internal flag in the affected DisplayPanel object. Any sampling that was drawn to the panel's window is erased, and background processing is turned back on.

9.6.4 Stepping and looping

Stepping refers to specifying a new current frame and displaying that frame. If the frame does not need to be redrawn, then displaying that frame just involves having each display panel object copy the backing pixmaps corresponding to the new current frame into the panel windows. Stepping implies that the stimulus for changing the current frame comes from a user request. The IGC supports the following step commands:

- *Step to first frame*: The current frame is set to the earliest non-empty frame.
- *Step to last frame*: The current frame is set to the latest non-empty frame.
- *Step forward*: The new current frame is set to the next non-empty frame later than the old current frame. If the old current frame is the latest frame, then the current frame is set to the earliest frame.
- *Step backward*: The new current frame is set to the next non-empty frame earlier than the old current frame. If the old current frame is the earliest frame, then the current frame is set to the latest frame.

A non-empty frame is defined as a frame that satisfies all of the following conditions:

- Contains at least one visible, time-varying DepictTuple object.
- Has not been marked by the user as a frame to skip.

Looping can be thought of as stepping, except the stimulus for changing the current frame comes from a timer expiration. In order for the user to effectively control looping, the IGC supports the following looping parameters.

- *State*: Whether looping is on or off.
- *Direction*: Whether the loop will move forward, move backward, or cycle (move forward to the latest frame, move backward to the earliest frame, then repeat).
- *Forward dwell*: The amount of time that each frame is displayed while looping forward.
- *Backward dwell*: The amount of time that each frame is displayed while looping backward.
- *First frame dwell*: The amount of time to spend displaying the earliest frame.
- *Last frame dwell*: The amount of time to spend displaying the latest frame.

9.6.4.1 Stepping walk-through

Here's what happens when the user selects one of the step control buttons of the control panel managed by the UI (fxa) process.

1. The fxa sends an IPC message to an IGC process. The IGC_Impl object receives this message which contains a step command.
2. If looping is on, then the LoopingMgr is told to turn looping off.
3. The new current frame is calculated according to the step command described in the previous section.
4. The new current frame is displayed. If the frame needs to be redrawn then the procedure described in Section 9.5.1, "Displaying the current frame" is followed. If the frame does not need to be redrawn, then each DisplayPanel object copies the backing pixmaps corresponding to the new current frame into its panel windows.

9.6.4.2 Looping walk-throughs

Here's what happens when the user enables looping by clicking on the loop state button of the control panel managed by the UI (fxa) process.

1. The UI process sends an IPC message to an IGC process. The IGC_Impl object receives this message which contains a flag indicating that looping is to be turned on. This request is then passed on to the LoopingMgr object.
2. The FrameSeq object is instructed to disable background processing since this is not really needed if looping is active.
3. The EventDispatcher is told to activate a timer, and to call a method in the LoopingMgr object when the timer expires.
4. When the timer expires, the following is done:
 - ◆ Determines the step command which is calculated from the looping direction and the current frame and passes the command to the FrameSeq object which executes steps 3 and 4 of the Stepping procedure (Section 9.6.4.1).
 - ◆ Determines how long to display this current frame by considering the looping direction and the dwell times. The EventDispatcher object is then told to expire the next timer using this computed duration.

Here's what happens when the user disables looping by clicking off the loop state button of the control panel managed by the UI (fxa) process.

1. The UI process sends an IPC message to an IGC process. The IGC_Impl object receives this message which contains a flag indicating that looping is to be turned off. This request is then passed on to the LoopingMgr object.
2. The FrameSeq object is instructed to re-activate background processing.
3. The EventDispatcher is told to deactivate the timer.

9.6.5 Swapping

Swapping is when the user selects one of the small IGC processes to display its loaded products in the large display area. That IGC process will now become the large IGC, responsive to the user interactions from the menu bar, control bar, volume browser, and other applications initiated by the UI process.

Meanwhile, the IGC process that was displaying its products in the large display area will now use the display area that the small IGC was using.

Here is a walk-through of what happens to the IGC Process that is selected to be swapped into the large display.

1. The user clicks mouse button 3 inside one of the display panels.
2. The EventDispatcher object receives the Xlib button release event, and determines to which DisplayPanel object to send the event, by examining the window where the event occurred.
3. The DisplayPanel object instructs the UI_WorkspaceManager object to send a message to the UI process requesting a swap.
4. The IGC_Impl object will then receive an IPC message from the UI process indicating a new size type which the IGC_Impl object saves and the ID of a new X window to use for the IGC layout. This is sent to the DisplayMgr object.
5. The DisplayMgr object destroys all of its DisplayPanel objects, and any X windows that were created for the four panel layout.
6. If the panel layout is single panel, then a DisplayPanel object is created using the X window that was sent by the UI process.
7. If the panel layout is four panel, then four new X windows are created. The X window passed in by the UI is the parent window. The origin and extent of these windows are computed so that they evenly subdivide the display area of the parent window. Since these windows will be displayed on top of the parent window, the parent window will not be visible, and will not be used by any DisplayPanel object. The depth (8 or 24 bits) of these new four windows will be the same as the parent window. The DisplayMgr will then create four DisplayPanel objects, each constructed with one of the four windows just created.
8. Each created DisplayPanel object will determine whether its in 8- or 24-bit mode by querying the depth of the window that was passed in to the constructor. It will then create an additional window for graphics, if in 8-bit mode, and also create a backing pixmap for each active frame.
9. Every DepictTuple object in the FrameSeq is forced to re-render by resetting its internal zoom and density information.
10. The current frame is redrawn, and all other frames are marked as needing redrawing.

Here is what happens to the IGC_Process that will be swapped out of the large display and into one of the small displays.

1. Receives a message from the UI process which asks the IGC_Impl to send a message back to the UI process acknowledging the receipt of the first message. This lets the UI process know that this IGC is not busy drawing or handling a user's request and is ready to swap.
2. Steps 4 - 10 of the above procedure are then followed.

CHAPTER 10 Depictables

10.1 What is a depictable?

A depictable is a software object that contains data and a method for turning those data into a viewable display (referred to herein as 'rendering'). The basic defining characteristics of a depictable are its depictable key and DateTime. A depictable key is just a number that points to a table entry describing a particular combination of depictable type, data sets to use, and other items that characterize a particular displayable product. A DateTime object contains information about both valid time and forecast time. A depictable receives data in a relatively unprocessed format, and is responsible for any remapping and graphics generation.

10.2 The depictable class hierarchy

All depictable objects are derived from the Depictable base class. The Depictable class itself contains no hydro-met data, only the metadata that identifies the basic characteristics of the particular instance of the depictable.

Derived from the Depictable class are the two major subclasses, ImageDepictable and GraphicDepictable; all of the concrete depictable types are derived from one of these two. Neither of these classes carries any data at all; they serve just to functionally separate images from graphics. The major difference in the public interface between the two is that ImageDepictable has routines that support doing image combination and drawing color bars.

There are currently just over 50 different concrete depictable types available. Some depictable types have an additional sub-hierarchy beyond just inheriting GraphicDepictable or ImageDepictable. This can result from two or more depictable types sharing a data source, a rendering strategy, or a frame of reference. In a case where two depictables share one or more of these characteristics, but one is an image and one a graphic, they currently need to be in separate sub-hierarchies.

10.3 Support software

This section describes seven different categories of support software that depictables use to do their job of creating viewable displays. Some of these represent class hierarchies and others just groups of related routines.

10.3.1 The Painter class

The Painter class provides an abstract interface for drawing graphics in depictables. The Painter class also contains methods that generate text and symbols using scalable stroke fonts. It does no mapping; it deals only in raw device coordinates. The Painter class is an abstract base class which currently has three concrete derived classes: one that draws to an X drawable (bitmap or pixmap), one that draws to an image (array of char), and one that creates PostScript output. The concrete Painter that writes to X is what is used to create mono-color graphics. The concrete Painter that writes to an image is used for multi-color graphics and for creating color bars in images. The concrete Painter that creates PostScript output is used for hard copy. When the 'paint' method is called an object referred to as a 'GraphicsTarget' is passed to the depictable and the depictable creates the correct type of painter based on it. The Depictable base class then holds a pointer to this object and uses it to do all of its graphics.

10.3.2 The Depictor class

A depictor (not depictable) is a software object that describes a physical frame of reference. The Depictor class is an abstract base class which has three concrete derived classes: the MapDepictor, the ThermoDepictor, and the XsectDepictor. The MapDepictor class describes a geographic frame of reference (such as a polar stereographic projection); the ThermoDepictor class describes a thermodynamic frame of reference (such as a skew-T diagram); the XsectDepictor describes either a time-height or space-height cross section frame of reference. A Depictor object is generally constructed from a file: *.sup files are used for the MapDepictor, *.thermo files are used for the ThermoDepictor, and *.xsect files are used for the XsectDepictor. A Depictor object carries with it methods for mapping back and forth from a physical coordinate system (lat-lon, pressure-temperature, or distance-height) to an arbitrary cartesian (x-y) coordinate system. A Depictor object also carries with it a domain in arbitrary cartesian space, so it can completely describe a physical frame of reference, such as the Northern Hemisphere scale. A depictor can also be used to describe the frame of reference of raw data sets, such as gridded or image data. Individual depictable keys also have depictors associated with them. By default they are associated with the depictor of the currently selected scale, but an entry in the depictable information table might associate them with, for example, a skew-T diagram. Any time the depictor associated with the depictable being loaded is not equivalent to the depictor associated with the depictable in the display, an implicit clear is generated. This is how the WFO-Advanced knows not to overlay soundings on top of satellite data.

10.3.3 The Xform class

The Xform class is a software object that handles mapping from one frame of reference to another. In this case, a frame of reference could be index coordinates in a grid, latitude and longitude, pressure and temperature, or pixel coordinates in an X window. In the most general case, an Xform is constructed with and data domain, a data depictor, a display depictor, and a display domain. An example of a data domain would be a range of grid indices (e.g. 1 to 95 by 1 to 65 for the AWIPS 211 grid). The data depictor in this case would be one describing the geographic area covered by the grid, and the display

depictor would be one describing the output scale. The display domain would be one describing the range of pixel coordinates in the display (e.g. 0 to 919 by 0 to 919 for the big IGC). In this example, the Xform created would provide a mapping from a grid index to a pixel on the display.

The Xform object also allows the user to define an offset origin in data coordinates. There are then methods that allow the user to map points relative to that offset, including the ability to automatically account for any local rotation of the coordinate system rotation. These features are especially useful for plotting station model plots and vector data.

10.3.4 The XformFunctions module

Most of the time, the code that creates Xform objects that depictables use is not actually in the code for depictables. Rather, it is in one of a set of free functions that reside in the module XformFunctions.C. Whatever Xform object is provided by these routines also has the zoom state incorporated into it.

10.3.5 Remapping tables

There are two separate remapping tables classes, ImageTable and GridTable, both of which map data from one grid to another. ImageTable works with grids of bytes and simply copies values from one grid to the other, whereas GridTable works with grids of floats and does bilinear interpolation. GridTable also has the ability to map vector data in a way that handles component rotation from one grid-relative coordinate system to another. In order to operate, both classes require that the size, area, and projection of both the input and output grids be fixed and known. When a remapping table is requested, an attempt will be made to find the table in memory first. If the table is not found in memory, an attempt will be made to read it off of the disk.

10.3.6 Fortran Graphics module

There are many Fortran-based graphics creation routines that we have borrowed from FSL's previous meteorological workstation effort (DARE-II). The FortranGraphics module allows these Fortran routines to make use of the Xform and Painter classes to do graphics. All of the routines in FortranGraphics are free functions. In order to use the FortranGraphics module, a depictable must first give FortranGraphics a Painter and an Xform using the FGinitialize routine. The FortranGraphics module then provides an interface to Fortran routines that has all of the capabilities of the Painter. All coordinates are automatically passed through the Xform that was provided, and the user can specify that coordinates be treated as absolute or offset.

10.3.7 Current usage of Fortran in depictables

There are some general principles followed for interfaces between Fortran and C or C++ in depictables. All C or C++ routines that are intended to be callable from Fortran are global in scope, have all lower case names, are declared as extern "C", and have all variables passed as pointers. It is straightforward to build C strings in Fortran, but difficult to do the reverse. Thus all strings passed to C or C++ from Fortran are declared as simple character arrays, and it is up to the Fortran caller to build an array of bytes with a null terminator. Conversely, if a Fortran routine needs a string passed in from C or C++, it should be declared as a byte array, and it is up to the Fortran routine to convert it to a Fortran string internally.

Fortran and C or C++ should never try to exchange C style multi-dimensional arrays (pointers to pointers).

Fortran modules borrowed from DARE-II include the contouring routine, the streamline routine, and the routines that draw fields of wind barbs and arrows from gridded data. Also borrowed from DARE-II are some thermodynamics routines, sounding diagnostics routines, and some horizontal finite differencing routines. We also have adapted a Fortran Redbook decoder from NCEP.

10.4 How depictables work

10.4.1 How depictables create viewable displays

When a depictable is constructed, it generally only gathers metadata. This metadata might include, among other things, information about which specific data sets it needs, or style information. Part of gathering information about data sets usually includes obtaining a depictor (not depictable) object that describes the geographic frame of reference of the data. Examples of style information would be a contouring interval or how to label a color bar.

When a depictable object renders itself, the method used in the depictable is called `paint`. One of the arguments to `paint` is a graphics target. From the graphics target, the depictable obtains a Painter object appropriate to whether it is an image or graphic depictable. Other arguments to `paint` include the size of the display, a depictor that describes the geographic frame of reference of the display scale, and the zoom state.

The hydro-met data needed for creating a display are read in the first time a depictable renders itself. These data are then held within the depictable until the object is destroyed. If the user zooms or roams and the same depictable object needs to render itself again, the depictable does not need to go out and reread the data. Occasionally, the act of acquiring the data held within the depictable might also involve performing some coordinate transformations on the data.

Once a depictable has its data, the process of creating a viewable display begins by creating an Xform from the display and data depictors, the display size, and the zoom state. The location in its native coordinates of each pertinent piece of data is passed through the Xform in order to get its location in display coordinates, and then those display coordinates are passed to a Painter method to plot graphical information representing that data at the proper location on the display. This description is obviously highly simplified, but it is, in essence, what happens.

10.4.2 A specific example

As an aid to understanding how depictables create viewable displays, it is helpful to have a specific example. The depictable that will be used for this example is the depictable that creates a plan view raster image display of satellite data.

The actual C++ class name of this depictable is the `SatPVImageDepict`. It is directly subclassed from the `PVImageDepict`, which is subclassed from `ImageDepictable`. As far as creating a viewable image goes, the `SatPVImageDepict` only provides a method for retrieving the data; the work of changing this data into an image all happens in the base class `PVImageDepict`. `SatPVImageDepict` currently has one sibling class that makes plan view images out of raster data from the Nexrad radar.

During construction, `SatPVImageDepict` gathers a substantial amount of metadata, but no satellite data. It stores internally the depictable key and `DataTime` on whose behalf it was constructed. Based on its depictable key, it retrieves depictable information and style information. The most important item of depictable information for this depictable is the list of data keys that refer to the data sets this depictable needs to obtain. The style information determines how the color bar is labeled and how data are scaled and formatted for cursor sampling. Using the data key, the depictable also looks up the name of a file that allows it to construct a depictor (not depictable) that describes the projection and area of the satellite data it is to retrieve.

Most commonly, this depictable has only one associated data key. The first time it renders itself, the depictable passes this data key and the time to a satellite accessor to retrieve an array of bytes that represents the satellite data. The accessor also returns the dimensions of this array, and these dimensions along with the data depictor provide a complete description of the geographic frame of reference of this data.

If the depictor of the display scale and the depictor of the data are determined to have different scales, an image remapping table is used to remap the data to a new geographic frame of reference which has the same projection as the display scale and roughly preserves the areal coverage and resolution of the original. This remapped grid then becomes that which is cached in the depictable and the original data from the accessor are discarded.

Once the display and data are determined to have the same projection, the depictor and dimensions of the display scale, the zoom information, and the depictor and dimensions of the data are passed to one of the modules in `XformFunctions` called `getPVImageXform` to get an `Xform` object that maps from data pixels to display pixels. For image depictables, another item that can be obtained from the graphics target is a pointer to the character buffer in which the IGC expects to receive the output image. This `Xform` is used to perform a linear mapping from the data to the output character buffer. Finally, using the style information and the `Painter`, the depictable creates the appropriate color bar.

10.5 Managing depictables

10.5.1 Context of depictables

In order to understand how to manage depictables, it is helpful to understand the context in which they operate. The following is a generalized description of what happens when data are displayed on the WFO advanced.

When the user selects a product button from a data menu, the correct depictable key to load is looked up in the product button info based on the current scale. The user interface process sends this depictable key to the IGC (display process) with a load command. The IGC does an inventory based on this depictable key, and if there are currently no items in the inventory for this depictable key, load processing stops. Otherwise, using the depictable information table, the correct depictor for this depictable key is obtained and compared to what is currently in the display; if they differ, an implicit clear is generated. Next, the inventory that was previously obtained on the depictable key is passed to the time matching routines, which determine which frame gets data for which time. The IGC requests from the DepictableServer a Depictable object for each frame that needs one. A Depictable object is created based on a depictable key and a DateTime. Finally, each depictable object is rendered to its frame.

10.5.2 Adding new depictables

The section will discuss several items with which one needs to be concerned when adding a new depictable to WFO-Advanced. Not all of these tasks need to be performed every time a depictable is added. The discussion here will be at a higher level of detail than has been true up to now, with more references to specific files in the WFO-Advanced software tree. All file name are assumed to begin with `#{FXA_HOME}/`.

10.5.2.1 Accessor for a new data type

Accessors in general have three methods that need to be available for depictables: constructor, inventory, and access. A very common paradigm is to construct an accessor object based on a data access key, and then the inventory method has no input arguments, and the access method has only a DateTime as an input argument. Some accessors, such as the METAR accessor, don't deal with data access keys at all, since they always deal with only one data set. Often it is possible to make a new depictable type that is only a different way to render an existing data set, in which case a new accessor is not needed.

10.5.2.2 Formally define a new depictable type

The file `src/dataMgmt/DM_DepictableInfo.H` contains an enumeration called `DEPICT_TYPE` which is the formal list of all depictable types which can be instantiated. A new item in this enumeration is always required when a new depictable type is created. New items in this enumeration should always be added to the end of the enumeration; changing any of the existing numbering can break things.

10.5.2.3 Code for the new depictable type

The source code for all depictables lives in the directory `src/depict`. All concrete depictables are derived from either the `GraphicDepictable` or `ImageDepictable` class, which are both derived from `Depictable`. Some depictable types have an additional sub-hierarchy beyond just inheriting `GraphicDepictable` or `ImageDepictable`. This can result from two or more depictable types sharing a data source, a rendering strategy, or a frame of reference. In a case where two depictables share one or more of these characteristics, but one is an image and one is a graphic, they currently need to be in separate sub-hierarchies.

10.5.2.4 Enable instantiation of the new depictable type

The source code file `src/dataMgmt/DepictableServerDepicts.C` contains the routine `DepictableServer::newDepictable`. This routine returns a depictable object of the desired type as a pointer to the base class `Depictable`. The depictable information table is used to determine the value of `DEPICT_TYPE` for a given depictable key; a huge case statement has an entry for each possible displayable depictable type.

10.5.2.5 Enable inventories for the new depictable type

The source code file `src/dataMgmt/DepictableInventory.C` contains the routine `DepictableInventory::allTimes`. This routine returns a list of `DataTimes` based on a depictable key. Just as with `DepictableServer::newDepictable`, the depictable information table is used to determine the value of `DEPICT_TYPE`, which then goes to a case statement. In this routine, it is often possible for different depictable types to use the same branch of the case statement. This happens most often when two depictables types are just different ways of displaying the same data set.

10.5.2.6 New data key entries

The file `src/dataMgmt/dataInfo.manual` is the base data for the data access information for all data types except for gridded data. When changes are made to this file, one needs to rerun the data target in this directory, which will cause gridded data keys to be generated and be put together with the keys in `dataInfo.manual` to create `dataInfo.txt`, the file that the workstation reads at start-up to get its data access key information. `dataInfo.manual`'s header documentation describes what each column in the table means. Some columns can have different meanings for different data types. It is assumed that inside a depictable object, the depictable knows what kind of depictable it is and what kind of data it needs, and thus which methods in the `DM_DataAccessInfo` class (defined in file `src/dataMgmt/DM_DataAccessInfo.H`) are valid for that data type. It is possible that a new depictable type will require new kinds of data access info; this will require modifying the code for the `DM_DataAccessInfo` class and adding additional header documentation to `dataInfo.manual`. In a case where a new depictable type is just a different way of displaying an existing data set, it is possible that no modification needs to be made to this file.

10.5.2.7 New depictable key entries

The file `src/dataMgmt/depictInfo.manual` is the base data for the depictable information for all depictables except those for gridded data. When changes are made to this file, one needs to rerun the data target in this directory, which will cause gridded data keys to be generated and be put together

with the keys in depictInfo.manual to create depictInfo.txt, which the workstation reads at start-up to get its depictable key information. depictInfo.manual's header documentation describes what each column in the table means. While different types of depictables might leave certain columns unused that others do not, in general each column in the depictable information retains the same meaning for all depictable types. Adding a new type of depictable will always result in at least one new line being added to this file.

10.5.2.8 New product buttons

The file src/dataMgmt/productButtonInfo.txt is a table where product buttons are made known to the workstation. Header documentation in this file describes what each column in the table means. For each product button, this file contains a list of which depictable keys get loaded for each scale. Adding a new type of depictable will generally result in at least one new product button being created. In general, there is a one-to-one relationship between product buttons and depictable keys, but it is possible for the same product button to load different depictable keys at different scales and for two or more different product buttons to load the same depictable key.

10.5.2.9 Place new product buttons on the menu

The file src/uiProc/dataMenus.txt determines the layout of the data selection menus. Header documentation describes how to make changes to the file. Any product button needs to be entered in the dataMenus.txt file in at least one place for it to be used to load data. There is in general a one-to-one relationship between product buttons and entries in the dataMenus.txt file, but it is possible for the same product button to occur in two or more places in the data menu.

CHAPTER 11 Applications and Extensions

This section describes D2D's applications and extensions. Applications are loosely integrated external programs that communicate with D2D using their standard input and output channels. Extensions are tightly integrated objects made from a framework that can interact with the forecaster directly on the D2D display areas. Both provide a mechanism to add customized, special features and functionality not in the base D2D system.

11.1 Applications

Applications run under the D2D application interface, a shell-like subsystem that runs programs external to D2D. Both new and existing programs can become D2D applications through this interface. They may send requests to the interface, and receive notifications from the interface. Requests cause D2D to perform actions. Notifications let D2D tell applications about changes in its state.

11.1.1 What is the Application Interface?

In the simplest sense, the D2D application interface provides a way to run programs external to the D2D software. For example, either a Web browser or a word processor could be an application run by the interface. Any program that runs with the HP/UX variant of the Unix operating system can be considered an application, which can bring up its own user interface by creating top-level windows on the X display.

11.1.1.1 Interaction with D2D

In addition to being started by the D2D system, applications can also interact with D2D. Applications communicate with D2D using their *standard input and output channels* provided by the Unix operating system. Before executing an application, the application interface creates two pipes: one to connect the application's standard output and the other to connect the standard input. These pipes are connected to D2D. Data on the output pipe is read by the application parser, which looks for requests. Notifications for an application are sent on the input pipe.

Because reading from the standard input and writing to the standard output is trivial, the interface makes it easy to develop nearly any kind of application - or to adapt existing programs to the interface.

11.1.1.2 Launching Applications

Forecasters using the D2D software launch applications by selecting them from the D2D menus. Site administrators can add and remove applications from the menus by editing configuration files, without needing to re-compile D2D software.

11.1.1.3 Application Characteristics

One of the D2D configuration files (appInfo.txt) describes all the characteristics of the available applications, including

- An *on-screen label* appearing on the D2D menu, that the forecaster selects to launch a particular application.
- A *filename* containing the application's executable code.
- *Arguments* passed to the application on its command line.
- A choice *whether to prestart* the application: D2D will start these applications when it starts; otherwise, applications are started when the forecaster selects them.
- A choice *whether to automatically restart* the application if it terminates.
- A choice *whether to allow multiple copies* of the application to be running at one time, per D2D display.

The D2D application interface enforces the characteristics specified in the configuration file, which is read each time the D2D software starts.

11.1.2 Implementation of the Application Interface

Just like the Unix shells, the D2D application interface executes programs, which we call applications. Instead of typing the names of applications to run, users select them from the D2D menus. The applications always run asynchronously (to enable the core D2D software to continue running) and their standard input and output are always connected to the application interface, using Unix pipes.

D2D software constantly monitors all of the pipes connected to each of the running applications. Whenever an application writes text to its standard output, the D2D application interface reads that text and attempts to recognize any of the standard requests using a parser built with Lex. Since Lex automates parser construction, adding new requests to the application interface is simple. The application interface also queues notification text pending for an application and sends the text along the pipe connected to the application's standard input. The application then reads its standard input to receive the notification.

The singleton object of class AppMgr is the application manager, which maintains objects of class App, which represent running applications. The constructor of class App starts an application; the destructor stops it. Each App contains an AppParser, which parses the requests that an application writes to its standard output. When notifications need to be sent to an application's standard input, the D2D code calls methods in class App.

11.1.3 Application Requests

Applications send requests to D2D by writing text on their standard output. The application interface ignores any output produced by an application that it cannot recognize.

11.1.3.1 The Load Request

An application may load depictable data onto the D2D display by issuing a load request. Each kind of hydrometeorological product that can be drawn and animated on the D2D screen is called a *depictable*, which is indexed by a *depict key*. A table provided with D2D lists all depict keys, which are positive

integers. For example, the depict key 101 is an infrared satellite image; depict key 2148499544 is the 310K relative humidity plot from the Rapid Update Cycle (RUC) model.

Writing the text @@-Load followed by a space-separated list of depict keys to the application's standard output will load each of the depictables represented by those depict keys. Certain depict keys are treated specially. If an application loads one of these special depict keys, then the application itself provides the depictable data in an external file in the netCDF format. The D2D software will read and render the data in the file on the display.

11.1.3.2 The Export Request

Applications that need to know what depictables a forecaster is viewing can use the export request to get a copy of the data used to draw those depictables. By writing @@-Export followed by an integer representing the type of the data to export, the application interface will create files in the /tmp directory representing any loaded data. The interface then sends a list of the files it created back to the application on its standard input. If the list is empty, then the forecaster was not viewing any data of the type the application requested.

11.1.3.3 Miscellaneous Requests

There are a number of additional requests an application can make; many of these were added to the application interface to facilitate development of certain features of D2D. However, application developers are also welcome to use these requests.

- @@-Clear
Clears the D2D main display area.
- @@-UserID userID
Sets the current user ID to userID.
- @@-ReqUserID
Requests that the application interface send a notification telling what the current user ID is.
- @@-Print dest mvi density copies scale landscape color manual invert size
Prints the main display area to dest, which is a file or a command if the first character is '|', using the given mvi (magnification) and density, and scaling the output by scale percent. Prints the requested number of copies, in landscape mode if true, in color if true, inverting black and white if true, and on the given paper size.
- @@-LoadMode mode
Sets the current load mode to mode.

11.1.3.4 Supporting New Requests

To add support for new requests, one edits the file appParser.l, which contains the Lex specification for each request. The Lex code should return a token specifying what request it got, and should set the yylval variable to the text of the arguments of the request, if any. The tokens are defined in appTokens.h. The yylval variable is the Lex variable to hold the l-value of a parsed line.

Next, handling for the new token must be added in the AppParser::parse() method defined in AppParser.C. The yylval variable becomes the app_lval variable in the file AppParser.C, and it's

in this variable that any arguments to the request are parsed. The `AppParser::parse()` method becomes the owner of the memory in `app_lval`, so it must delete it when it's no longer needed.

11.1.4 Application Notifications

As stated earlier, not only can applications send requests on their standard output, but they can also receive notifications on their standard input. By reading their input, applications can be apprised of user actions and the state of D2D's main display. These are the notifications an application can receive:

- `set_scale scale`
When the forecaster changes the display scale in D2D, all running applications are notified by receiving the text `set_scale`, followed by an integer representing the new scale, on their standard input.
- `load_mode mode`
When the forecaster changes the load mode in D2D, all running applications are notified by receiving the text `load_mode`, followed by an integer representing the new mode, on their standard input.
- `visible`
The application configuration file specifies that only one copy of certain applications may be running on a D2D system at a time. If the forecaster selects such an application from the D2D menu while one is running, a new copy of that application is not started. Instead, the text `visible` is received on the application's standard input. This is called a request for visibility. Applications are expected to de-iconify, or raise their windows (if applicable) upon receiving this notification.
- `USER: userID`
Sent in response to the application that sends the `@@-ReqUserID` request, this notification tells what `userID` is currently registered with D2D.
- `exported files ...`
Sent in response to the application that sends the `@@-Export` request, this notification tells what files were exported. If no data were exported, the list of files is empty.

To expand the notifications D2D sends to an application, new methods need to be added to class `App` to accept the data. The method should render the data as a printable text string and call `App::send()`, which buffers the data until an application can receive it.

11.2 Extensions

The extension framework is an object-oriented interface that enables developers to extend the function of the D2D system. Extensions are objects that reside in separate processes that provide interactive elements in the D2D display areas with programmer-provided logic. Such extensions could enable D2D users to display special kinds of data, interactively locate information in geographic databases, compute storm directions, and more.

We made the extension framework easy to use by simplifying the compiling, linking, and integration of extensions. This was achieved using the following concurrent strategies:

- The framework provides default behavior for extensions that performs the correct actions in most cases. Local extension developers augment or replace only as much behavior as necessary.
- The framework library provides its own main routine.
- The D2D software reads configuration information stored in plain text files; part of this information shows where extensions should appear in the D2D menus. An extension can be added by merely updating these files - there is no need to re-compile any of the core D2D software.

In addition, more than one extension may reside in the same process space (for example, to share code). Extension developers also have access to a user interface library, where they can display dialog boxes containing Motif-style widgets as separate, floating windows.

11.2.1 Extension Framework Classes

To create an extension, local extension developers make use of a number of C++ classes in the extension framework. Each of these classes represents a key extension concept. This section describes these concepts and their representative classes.

11.2.1.1 Interactive Depictables

Forecasters load products, such as satellite images or contour pressure plots, into the main D2D display. Products are represented by objects called depictables. An interactive depictable is a kind of depictable used by extensions.

Interactive depictables are half-objects shared between the core D2D software which draws them and the extension which manipulates their contents. They are indexed either by frame number or by frame time, since each frame in a displayed loop has a time index. Interactive depictables contain editable elements (described in the next section), and are the canvas onto which an extension draws elements, and on which the forecaster interacts with the elements.

The class `InteractiveDepictableExt` defined in the file `InteractiveDepictableExt.H` represents the extension half of the half-object. Local extension developers use the method `newElement` to add or change an editable element in the depictable. There are also methods to query and remove elements, and methods to query the frame and time index of the depictable.

11.2.1.2 Editable Elements

Editable elements are on-screen objects visible to the forecaster, which can be moved, reshaped, or deleted. They appear on the D2D display as points, lines, polygons, and polylines (unclosed polygons), and may have text annotation. An extension may prevent editing of any of its elements; if an extension allows editing of some elements, the vertices of the elements appear with markers that the forecaster uses as editing handles. A family of classes related through inheritance represents editable elements. The file `EditableElements.H` contains the class declarations for the elements.

Editable elements may be used to depict any kind of data: isobars may be a series of polylines, storm centroids may be points, warning areas may be polygons, and so forth. Extension developers place editable elements by specifying latitude and longitude, and the D2D software automatically draws them in the correct place in relation to the map background.

11.2.1.3 The Extension

Class Extension represents an abstract extension. To make a local extension, a developer derives a new class from class Extension, and overrides the virtual methods as necessary. This class is in the file Extension.H.

11.2.2 Using the Extension Class

To emphasize that developers must derive from class Extension, the constructor is protected. As a result, only derived classes may create derived extension objects. For example, a simple - yet complete - extension which does nothing follows:

```
class SimpleExtension : public Extension {
public:
SimpleExtension( omitted ) : Extension ( omitted ) {}
};
```

11.2.2.1 The Active State

Several methods of class Extension deal with the active state of an extension. Only one extension may be active at a time; forecasters who load more than one extension choose one to be active by using a pop-up menu (or the middle mouse button) on the main D2D display. Extensions can take action when they become active by overriding the virtual method activeStateChanged. For example, the WarnGen extension (which assists the forecaster in generating warnings) displays its dialog box when it becomes active, and hides the box when it becomes inactive.

11.2.2.2 Editable Element Notifications

The methods Extension::edited*() deal with changes to editable elements. As stated earlier, extensions draw on the screen by creating editable elements in their interactive depictables. Whenever the forecaster changes an editable element, the extension is notified through these methods. For example, if the forecaster moves a line created by the extension, the method editedLine is called with a pointer to the EditableLine object that the extension created. If the forecaster deletes a vertex from a polygon, editedPolygon is called. If a forecaster creates a new editable element, the boolean parameter to these methods is true; if it is an element that the extension created earlier, the parameter is false.

By overriding the definitions of these functions, the extension can provide any kind of behavior necessary. As an example, the WarnGen extension uses points in a frame sequence to mark the location of a storm. If the forecaster moves a point in one frame, the editedPoint function computes a new storm track and adjusts the locations of points in other frames accordingly, based on computed velocity and frame times.

Extension::deletedElement() is a method that is called when the forecaster deletes an editable element. The default behavior is to take no action; extension developers may override the behavior with any action they desire.

11.2.2.3 Depictable Sequence Notifications

The extension framework represents a sequence of on-screen frames by a sequence of interactive depictables. As time passes, the frame sequence changes. For example, the forecaster may change the length of the frame sequence, or new data may arrive that cause the time indices of the frames to shift (auto-update). In order to convey these events to an extension, the method handleIntDepictSeqChanged is called; it notifies an extension whenever the sequence of interactive depictables changes. It is called with three sequences of pointers to interactive depictables, which are either

- *about to be deleted*. If the forecaster shortens the sequence, the earlier depictables are passed in here. For the auto-update case, it is always the first depictable previously at the beginning of the frame sequence.
- *old*. These are depictables that may have changed their frame index, but were otherwise already known to the extension.
- *new*. These are depictables that just got created. For the auto-update case, it is the latest, new depictable.

The reason for all this complexity is that the extension may have to perform some sophisticated processing in light of the new time indices and/or new depictables. As an example, consider again a storm tracking extension using points to locate a storm in each frame. The new depictables will be empty, so the extension will have to compute the new storm location in those frames and create the points in the right places.

11.2.3 A Sample Extension

To demonstrate how one might use the framework, shown below is an example of a small extension that creates points at different locations in every frame, with an annotation that shows that latitude/longitude of the point. The forecaster can move the point in any frame, and the annotation will change to show the new latitude and longitude.

```
class SampleExtension : public Extension {
public:
SampleExtension( omitted ) : Extension ( omitted ) {}
virtual void handleIntDepictSeqChanged(
SeqOfPtr< InteractiveDepictExt* >,
SeqOfPtr< InteractiveDepictExt* >,
SeqOfPtr< InteractiveDepictExt* > newDepicts
) {
char buf[20]; // Space to hold annotation
for (int i = 0; i < newDepicts.length(); ++i) {
// Put a point in each frame.
LatLonCoord coord(40 + (i * 3), -100 - (i * 3));
sprintf(buf, "Point at %f,%f", coord.lat,
```

```
coord.lon); // Make the annotation
// Create the point and add it to depictable
newDepicts[i]->newElement(new
EditablePoint(coord,
Marker(Marker::X_ICON), buf));
}
}
virtual void editedPoint(EditablePoint* pt, bool) {
char buf[20]; // Space for new annotation
// Get new location
LatLonCoord newCoord(pt->location());
sprintf(buf, "Point at %f,%f", newCoord.lat,
newCoord.lon); // Make new annotation
pt->setAnnotation(buf); // Update annotation
interactiveDepictables()[activeIntDepictIndex()]
->newElement(pt); // Update point in frame
}
};
```

CHAPTER 12 Monitoring and Control

12.1 Overview

The installation of WFO-Advanced workstations at the Denver WSFO in May 1996 represents a significant milestone in the NWS modernization. Based on AWIPS functional specifications, WFO-Advanced supports essentially all diagnostic and forecasting operations at the WSFO.

Components of WFO-Advanced include data ingest and management, user interface, display, and text generation. Each of these components needs to be monitored to ensure that the system operates as planned, providing the required support to WSFO operations. Experience gained in this monitoring effort can also be applied to the operation of the AWIPS Network Control Facility, whose responsibilities include remote monitoring of operations at AWIPS sites.

WFO-Advanced monitoring falls in three areas. The data monitor covers the ultimate success of the data ingest and management system, and is designed for both forecaster and developer use. The process monitor and restart mechanism is intended primarily for forecaster use. System performance monitoring addresses the general status of the computer system, and is of interest primarily to systems administrators and developers.

Other items not specifically monitored at this time, but for which a more extensive monitoring system is being developed, include external, network, and interprocess communication; and memory and disk usage. We intend to provide a graphical user interface to the monitor, with color used to indicate the state of various processes and subsystems, in a hierarchical display suitable for in-office monitoring.

The principal user interface for WFO-Advanced monitoring is a World Wide Web (WWW or Web) browser. As with the forecaster workstations, a design goal is to minimize window manipulation to make the monitoring job as simple as possible. During critical weather situations, we need to minimize any chances that a complex interface could interfere with a forecaster's timely issuance of warnings and forecasts.

Our concept of how the monitors would be used has evolved with experience at the Denver WSFO. Initially, we envisioned forecasters running a Web browser in one of the workstation's other workspaces, but it turned out that they never looked at the monitor. Subsequently, we added a color Xterm dedicated to monitor and restart functions. Still, we found that forecasters rarely used it. Office policy now is that the HMT (hydromet tech) on shift has responsibility for monitoring the system, and the monitor terminal has been moved into the HMT work area.

The current monitors are illustrated in the next section. At present, there are separate data and ingest-process monitors. We plan to incorporate these into a single, full-screen display, schematically illustrated in Figure12.1. In addition to bringing all components into a single interface, this layout provides a place for data notices, announcements, and other information that should be brought to the attention of the operations staff. Just as the 5-pane display enhances operational efficiency by minimizing window manipulation, this single-panel monitor will more effectively communicate essential information to the forecasters.

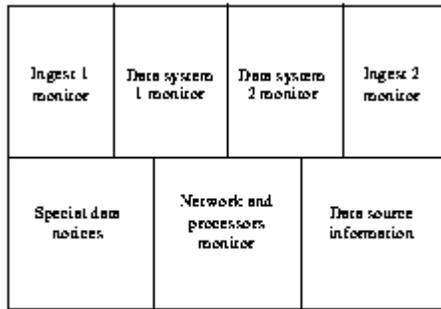


Figure12.1 WFO Advanced monitor Web page.

12.2 Data Monitor

The data monitor (Figure12.2) provides an overview of the success of all data ingest. In the example shown here, three data ingest machines at FSL (cmsdata1, yarmouth, and fsldata1) and two at the Denver WSFO (dendata1 and dendata2) are monitored (Ingest was not running on yarmouth or dendata2 at the time of this snapshot.) Each item on this display presents a summary of several related datasets, indicating that all component datasets are up to date (check mark) or that at least one of the components is somewhat late (caution triangle) or seriously late (red X).

FX-Advanced Product Monitor

Thu Dec 19 18:25:01 GMT 1996

CMSdata1	State	Yarmouth	State	FSLdata1	State	Dendata1	State	Dendata2	State
Radar Data									
Point Data		Point Data		Point Data		Point Data		Point Data	
Grid Data		Grid Data		Grid Data		Grid Data		Grid Data	
Sat Data		Sat Data		Sat Data		Sat Data		Sat Data	
FSL Data		FSL Data		FSL Data		FSL Data		FSL Data	
Text Data		Text Data		Text Data		Text Data		Text Data	
Reallink Graphics		Reallink Graphics		Reallink Graphics		Reallink Graphics		Reallink Graphics	
Data Disk Usage		Data Disk Usage		Data Disk Usage		Data Disk Usage		Data Disk Usage	

Figure12.2 WFO Advanced data monitor.

In order to see the details, the user clicks on a link, revealing a dataset-by-dataset display, as illustrated in Figure12.3.

Point Data on denapp1.fsl.noaa.gov

Thu Dec 19 18:30:08 GMT 1996

State	Last Update	Description	Info
	96/12/19 18:28:23	Lightning Plot	
	96/12/19 18:30:09	MRTAR Plot	
	96/12/19 18:14:09	Profiler Plot	
	96/12/19 15:04:05	RAOB Plot	

Figure12.3 Data monitor detail page, showing status of each dataset.

For numerical models, each check-mark can represent hundreds or thousands of individual grids. To provide more useful information to the user, a number is appended to the check, indicating the percentage of expected grids actually available.

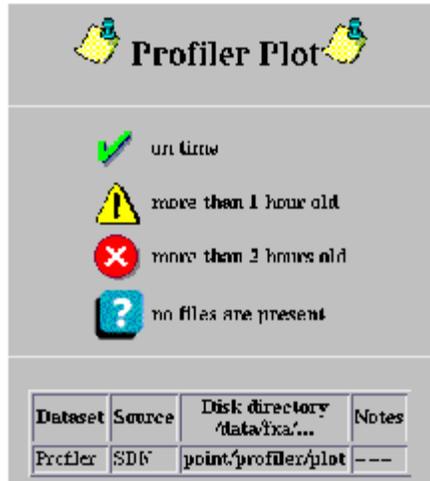


Figure12.4 Timeliness criteria.

Clicking on an Info box provides information on the timeliness criteria for the specific dataset, as well as information on its source and storage location (Figure12.4). Not shown in the Figure is contact information, directing the user to a source to call if data are missing or late.

12.3 Ingest Process Monitor

Tardy or missing data can be a result of a problem in collection, transmission, or receipt and processing. The WFO-Advanced Ingest Process Monitor tracks much of the data processing occurring in the system. This monitor is shown in Figure12.5.

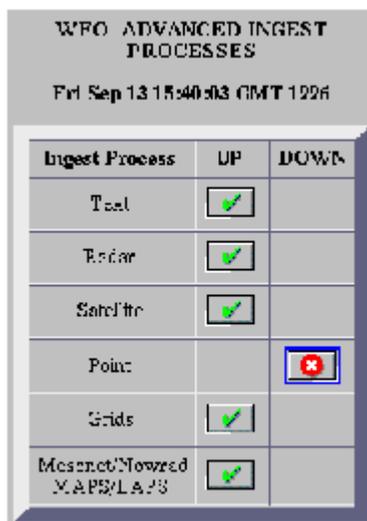


Figure12.5 Ingest Process Monitor

In the example shown here, at least one of the processes classed under "Point" is not running. The user can click on a hyperlink (not shown) to view a log of processes that have been down, or can just click on the red X to bring up the restart menu illustrated in Figure12.6. The All Ingest choice also resets some system resources.

In some cases, the workstations must be restarted in order for arrival notifications/auto-update to continue. A dialog informs the user of this necessity.

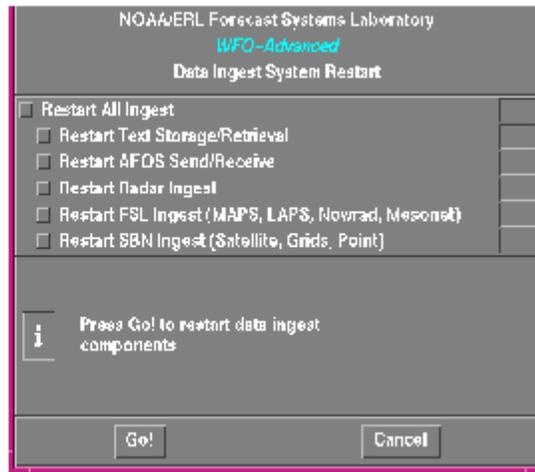


Figure12.6 WFO A Data Ingest Restart menu.

12.4 System Performance Monitor



Figure12.7 Performance monitor selection menu.

Information on system performance is made available to forecasters, systems administrators, and developers. The interface is illustrated in Figure12.7. Users select one or more items from each area and a script reads the stored performance statistics to generate charts or graphs.

Data are collected every 15 minutes by a readily-available program called "sar" and stored on disk for 30 days. CPU-usage statistics (as illustrated in Figure12.8) can be helpful in isolating times when unusual activity is taking place. Such information also can help in balancing loads among the various systems.

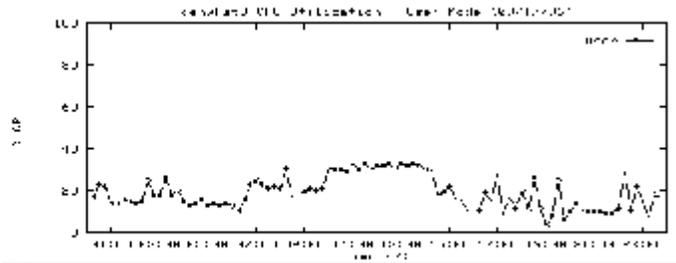


Figure12.8 One day time series of workstation CPU usage.

12.5 Prototype AWIPS Monitors

A concept developed for national monitoring features a map with a dot representing each site. The color of the dot indicates the overall status of the site, as in other monitors, with a green dot indicating all is well, yellow for some problems, etc. Clicking on any dot would bring up that site's monitor. An example of the latter is the monitor that is currently available to NCF and NWS headquarters, shown in Figure12.9.

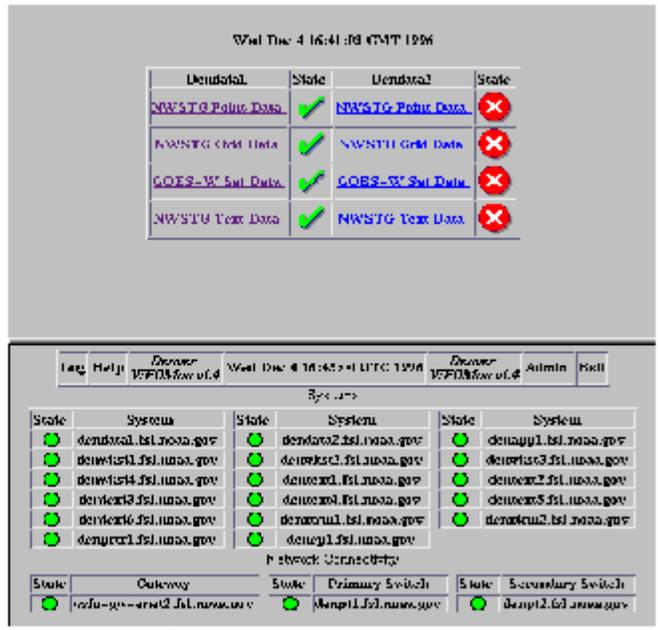


Figure12.9 Denver monitor currently in use at AWIPS Network Control Facility.

In addition to data ingest, we anticipate that the monitor would report on the health of systems, as illustrated in Figure12.10. Here, each node in the network is represented by an icon, and the color indicates the overall health of that node. Clicking on an icon will allow the user to examine the various processes, disks, etc. on that node.

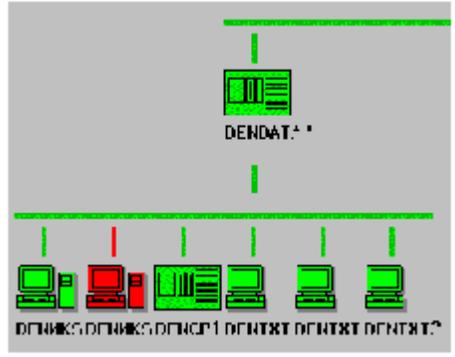


Figure12.10 Denver monitor display

A monitor of this sort is currently under development at FSL.

12.6 Concluding Remarks

FSL has been working to develop monitoring capabilities for its WFO-Advanced systems. Examples of the current operational systems have been illustrated here. Other interactive monitors are also under

development, which are intended to allow users to monitor the system with more granularity and provide better control.

Work in this area continues, and the monitoring systems will grow along with AWIPS.

CHAPTER 13 Text Subsystem

13.1 Overview

The text subsystem consists of the text display, the Informix database, and the storage/retrieval of text messages.

13.2 Text Display User Interface

The text display user interface runs on a dedicated X terminal. Individual text display and editing windows are also available on the WFO-Advanced graphic display, but these windows are not fully integrated into the text product notification mechanism, and therefore event-driven capabilities like alarm/alert and auto-update are available only on the dedicated display.

13.2.1 Small processes

The D2D graphic/image display component of WFO-Advanced consists of a few rather large processes, with a large amount of functionality linked into each process. In contrast, the text subsystem processes tend to be smaller and more dynamic, forking new processes as necessary to do many tasks, like loading and storing data. This approach keeps the individual processes simpler and faster to build, which is a boon to the harried developer, but incurs some performance penalty with every process started. It also provides some additional damage control to the user in the event of a process crash. Fortunately the price of starting a UNIX process is rather low.

13.2.2 Use of Tcl/Tk

The text subsystem user interface evolved as a set of incremental prototypes. To avoid the compile and link bottleneck in the development process, a fast, interactive user interface tool kit called Tcl/Tk (Ousterhout 1994) was used to develop the user interface. Tcl is a tool command language (an interpreted scripting language) and Tk is a user interface tool kit that works with it. Tcl/Tk has proven reliable enough to use in the final system. This tool kit is fast, powerful, and freely available, and we plan to use it more in other areas of WFO-Advanced.

While the use of Tcl/Tk has been a major asset to the project, there were a few issues related to its use that caused some problems. The largest organizational problem we had with Tcl is that there are only two namespaces: global and function-local. This well-known problem with Tcl

will be remedied in future versions of the language. We addressed the problem with a set of standard global variable prefixes.

Customization of the Tk text widget was also rather difficult. The Tk text widget is the most complicated and functional of the Tk widget set, implementing essentially an entire text editor. For us, it was a classic software reuse problem of an existing component being "almost, but not quite" what we needed. The auto-wrap capability was frustratingly close to what we needed, but in the end, we turned it off and implemented our own version.

13.2.3 Software Interface to the Text Database

The interface to the text database was implemented as a UNIX command named **textdb**. Command line options **-r** and **-w** are used to read and write the database, respectively, with the product ID given as an argument. Data are read and written from standard input and standard output. The **textdb** command follows the UNIX convention of returning a status of zero upon success, and nonzero if an error occurs. This provides a simple but very flexible way of interacting with the database in a UNIX environment.

For example, the UNIX command line

```
textdb -r DENNOWDEN | more
```

will pipe the latest Denver nowcast into a display program for on-line viewing, and the command line

```
textdb -w DENWRKNOW < workFile.txt
```

will store the contents of workFile.txt (created by an application program, for example) as a nowcast work file in the text database.

13.2.4 Text Display Processes

As of this writing, the text display X-terminal client processes run on the data server. A system design change to run the text display processes on the associated graphic workstation, where available, is under consideration.

Each instance of the text display invokes the following processes, identified here by the name of the associated Tcl/Tk script used to initiate that process.

- **textWS.tcl**
This is the top-level control process for the dedicated text display. There is one of these processes for each dedicated text X display. It forks the individual text windows and the alarm/alert process, and directly controls the top-level control window.
- **textWindow.tcl**
This is the process for an individual text display/editing window. There is one of these processes for each such window. Four of these are started by textWS.tcl for the user-controlled text

display/editing windows. An additional, initially hidden, one is used by WarnGen. Additional instances can be created on request by the user.

- textAlarmQueue.tcl
This process manages messages from the text notification server. It directly handles alarm/alert user notification, and forwards notifications to registered text display windows for auto-update processing. There is one of these processes per dedicated text display.
- textNotifyExpiration.tcl
This process is forked when the forecaster stores a watch or warning product. The process sleeps until 10 minutes prior to the product expiration, then pops a dialog box to remind the forecaster of the approaching expiration. One process exists for every pending product expiration.

13.2.5 Key Text Script Files

In addition to the script files mentioned in the previous section (the top-level script files associated with the major text display processes), there are various other Tcl/Tk scripts that define related sets of functionality, but are not directly run as processes themselves. These scripts are invoked via the Tcl "source" command (similar to a C include file). Each file defines a related set of procedures and global variables that work together to provide some subset of text display functionality.

- textGlobals.tcl
This file defines a set of global constants used throughout the text display, including some local site configuration parameters, like the local site id (CCC).
- textBrowser.tcl
This file implements a product selection browser user interface for the text display/editing window. It is sourced by the textWindow.tcl script.
- textAlarmAlertList.tcl
textAlarmDisplayWindow.tcl
textCurrentAlarmQueue.tcl
These files implement a set of procedures sourced by textAlarmQueue.tcl, to implement the Alarm/Alert functions of the text display.
- textBindings.tcl
This script defines a set of procedures used to control key bindings in the text window. Bindings change as the window enters and exits editor mode, are fairly involved, and were developed after the textWindow.tcl script was already mature, so they were put into a separate file sourced from textWindow.tcl.
- textScriptEdit.tcl
textScriptHelp.tcl
textScriptInterpreter.tcl
textScriptWindow.tcl
These files implement the user-defined scripting capabilities, available from every text window.
- textIsSpellInterface.tcl
textSpellWindow.tcl
These files implement a spell checker for the text editor, used by the textWindow.tcl process script.
- textLoadProcs.tcl
This file, sourced by the textWindow.tcl script, implements a set of routines for loading text products to the display window.

- `textHeaderBlock.tcl`
This file is sourced by the `textWindow.tcl` script, and implements a dialog box for editing the product header block prior to or while editing a product.
- `textJournal.tcl`
This file implements journaling for the text editor. The editor contents are dumped to a journal file once per minute while in editing mode, ensuring that no more than one minute of editing changes are lost in the event of a software failure. The script is sourced by the `textWindow.tcl` script.
- `textFileSelectDialog.tcl`
This file implements a file selection dialog box, used for user-defined scripts, and for importing and exporting between the text editor and UNIX files.
- `textPrint.tcl`
This file defines a simple procedure to send text to a printer.
- `textVersion.tcl`
This one-line file is written by the Makefile when the text display software is built. It is then used with the help menu of the `textWindow.tcl` process to identify the build date of the software.

13.2.6 The textWish Tcl/Tk Interpreter

The text display Tcl/Tk scripts are run with a version of the Tcl/Tk interpreter that includes a number of extensions to the usual interpreter (named `wish`). The extensions include the following.

- Logging via the `LogStream` Class,
- A special XIO handler that logs some fatal X connection events,
- Standard WFO-Advanced IPC, and
- Software interface to the text notification server.

13.3 Informix Database

WFO-Advanced stores decoded text products in an Informix database. The database works on a circular buffer basis, storing the newest version of each product over the oldest. The number of versions of each product or category of products is specified in a table, `versions_lookup_table.dat`. To improve performance, the storage space is fragmented based on the frequency of requests for a category of products; the typical read response time is 1 to 2 seconds. This database consists of three tables:

- `text_product_info` This table stores the version information for each product that is stored in the database. There are 5 columns in the table: `ccc_id`, `nnn_id`, `xxx_id`, `versions_to_keep`, and `latest_version`.
- `text_product` This table holds the individual messages decoded from the raw data. There are 6 columns in this table: `ccc_id`, `nnn_id`, `xxx_id`, `version_number`, `create_time`, and `product`. The `version_number` field corresponds with the `latest_version` field from the `text_product_info` table for the corresponding product.
- `state_match` - This table contains a listing of all `xxx_id` and `ccc_id` combinations for a state. There are three columns in the table: `state`, `xxx`, and `ccc`.

13.4 Text Server

A text server handles the communications between the text displays and the database. Two servers are run, in read and write modes, to simplify processing and minimize response time. The server is written using a combination of C++ and ESQL/C code.

13.4.1 Text Read Server

The text read server processes AFOS-like commands from the text display and retrieves the corresponding products for those commands. The read server uses two classes to accomplish this, the AFOSParser class and the TextDB class. The AFOSParser class breaks the command from the display down and determines what product(s) and version(s) need to be sent back to the display. The TextDB class handles the actual communication between the display and the database using DMQ and the network.

13.4.2 Text Write Server

The text write server receives decoded messages from the text decoders and sends them to the database. The server determines whether a new version of the product needs to be created or if the decoded message should be written over an old version of the product. The server communicates with the decoders and the database using DMQ and the internal Informix connectivity capabilities.