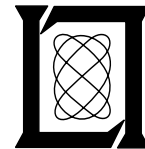# ASR-9 Weather Systems Processor Software Overview

O. Newell

20 October 2000

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LEXINGTON, MASSACHUSETTS

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE

ASR-9 Weather Systems Processor Software Overview

5. FUNDING NUMBERS

C — F19628-00-C-0002

6. AUTHOR(S)

O. Newell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Lincoln Laboratory, MIT
244 Wood Street
Lexington, MA 02420-9108

8. PERFORMING ORGANIZATION REPORT NUMBER

ATC-264

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of Transportation
Federal Aviation Administration
AND-420
800 Independence Ave., S.W.
Washington, DC 20591

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

This report is based upon studies performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, under Air Force contract F19628-00-C-0002.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

This document is available to the public through the National Technical Information Service, Springfield, VA 22161

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The ASR-9 Weather Systems Processor (WSP) augments the weather detection capability of existing ASR-9 radars to include low-level wind shear warnings, storm cell tracking and prediction, and improved immunity to false weather echoes due to anomalous propagation (AP). To economically develop and field an operational system at the 34 WSP sites, the FAA is pursuing a strategy that leverages the software written during the 10-year R&D phase of the project. To that end, the software developed at Lincoln Laboratory has been "hardened" to ensure reliable, continuous operation, and has been ported to a "Phase II" prototype built around the latest generation of COTS hardware.

A significant number of the hardened software modules are being used in the production version of the WSP with only minor modifications. This document provides a high-level description of these software modules, with an emphasis on how the modules fit together in the WSP system. Descriptions of the hardware environment in which the software executes are also provided.

14. SUBJECT TERMS

Airport Surveillance Radar
Wind Shear
Gust Front
Radar data
Microburst

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unclassified

ABSTRACT

The ASR-9 Weather Systems Processor (WSP) augments the weather detection capability of existing ASR-9 radars to include low-level wind shear warnings, storm cell tracking and prediction, and improved immunity to false weather echoes due to anomalous propagation (AP). To economically develop and field an operational system at the 34 WSP sites, the FAA is pursuing a strategy that leverages the software written during the 10-year R&D phase of the project. To that end, the software developed at Lincoln Laboratory has been 'hardened' to ensure reliable, continuous operation, and has been ported to a 'Phase II' prototype built around the latest generation of COTS hardware.

A significant number of the hardened software modules are being used in the production version of the WSP with only minor modifications. This document provides a high-level description of these software modules, with an emphasis on how the modules fit together in the WSP system. Descriptions of the hardware environment in which the software executes are also provided.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

The ASR-9 Weather Systems Processor (WSP) augments the weather detection capability of existing ASR-9 radars to include low-level wind shear warnings, storm cell tracking and prediction, and improved immunity to false weather echoes due to anomalous propagation (AP). To economically develop and field an operational system at the 34 WSP sites, the FAA is pursuing a strategy that leverages the software written during the 10-year R&D phase of the project. To that end, the software developed at Lincoln Laboratory has been 'hardened' to ensure reliable, continuous operation, and has been ported to a 'Phase II' prototype built around the latest generation of COTS hardware. A significant number of the hardened software modules are being used in the production version of the WSP with only minor modifications.

This document provides a high-level description of the Lincoln-developed software modules, with an emphasis on how the modules interact. Familiarity with the basic functions of the WSP is assumed. Readers unfamiliar with the WSP are referred to [1-9] as an introduction. An overview of the software modules, and the hardware/software environment in which they execute, is provided in section 2. The remainder of the document is organized into sections based on common functionality. Section 3 describes all the 'core' software modules involved in the generation and display of the WSP's product output stream. Section 4 describes the system's recording and playback capabilities. Section 5 describes the lower-level software libraries shared by many of the algorithm modules. Section 6 provides a software directory map, and details regarding the software build environment. Lastly, appendices provide information regarding the coding standards used and a number of the key data stream and file formats. Note that descriptions of the software modules related to the WSP system control and monitoring functions are not provided. This software, though based on code originally developed for the prototype system, has been substantially modified by the WSP production contractor.

2. OVERVIEW

The WSP produces estimates of low-level windshear (microbursts and gust fronts) and storm cell tracking/ prediction information, and provides graphical and textual displays of this information at the Tower/TRACON facility. The majority of software modules execute in the RDP, a VME chassis containing a variety of processing elements that resides at the radar site. A smaller number of modules execute in external workstations residing primarily in the ATC facility.

This section provides a brief overview of all major software modules, including a description of the hardware and software environment in which the software executes. A more detailed discussion of each module is provided in subsequent sections.

2.1 RADAR SITE SOFTWARE MODULES

A block diagram of all major software modules running at the radar site is shown in Figure 1. Time-series data enters the system at the upper left, from where it is time-stamped and distributed to multiple copies of the clutter filtering/autocorrelation function. This function is very compute-intensive, and is therefore spread across multiple processors. The data distribution module also provides a data stream to the time-series recording module. Following the filtering/autocorrelation step, 'base data', consisting primarily of reflectivity (dBZ), velocity (V), and processing 'flags' data, are passed to a post-processing module, which recombines the output from the multiple processors, performs some spatial smoothing and outputs the base data to the downstream weather algorithms and the base data recording module. The post-processing step also includes the integration of wind sensor information into the base data stream. This design results in base data recordings that contain all relevant system inputs synchronized in time, allowing for offline playbacks to accurately recreate the real-time system performance. Note that the base data recording module maintains a 20-hour base data history on disk, and provides a mechanism for transferring some or all of the data to tape on command.

The six-level weather server module is responsible for performing smoothing/contouring operations on the WSP base data and generating the six-level weather data stream. The data is output in polar form to the ASR-9, from where it is transmitted to the controllers scopes in the normal manner. The data is also converted to cartesian form, and transmitted to the storm motion algorithm module, the TWIP module, and the situation displays (via the product multiplexor).

The Storm Motion algorithm accepts six-level weather input in the form of cartesian images, and tracks storm cells as they evolve and move through the terminal area. Motion vectors are generated for each storm cell, as well as storm cell edge contour lines for the current time and 10 and 20 minutes into the future. The output of the storm motion algorithm is sent to the SD, as well as to the microburst detection algorithm.

The ASR-9 Microburst Detection Algorithm (AMDA) processes the polar velocity and reflectivity data, in conjunction with a motion grid provided by the storm motion algorithm. The velocity field is scanned for regions of divergent flow, and additional storm features such as storm cell motion and storm cell edge location are considered to confirm/disconfirm microburst detections. Output from the microburst detection algorithm is transmitted to the alert generation module.

The inputs to the Machine Intelligent Gust Front Algorithm (MIGFA) include polar-format base data containing reflectivity and velocity information, and the current center-field wind estimate as taken from the WSP's wind sensor interface in the tower equipment room. The algorithm searches the reflectivity and velocity fields for the 'thin-line' and other related signatures typically present at gust front boundaries, and

uses the detections in conjunction with the center field wind estimate to generate a wind-shift estimate product. MIGFA outputs gust front boundaries in the form of line segments to the alert generation module.
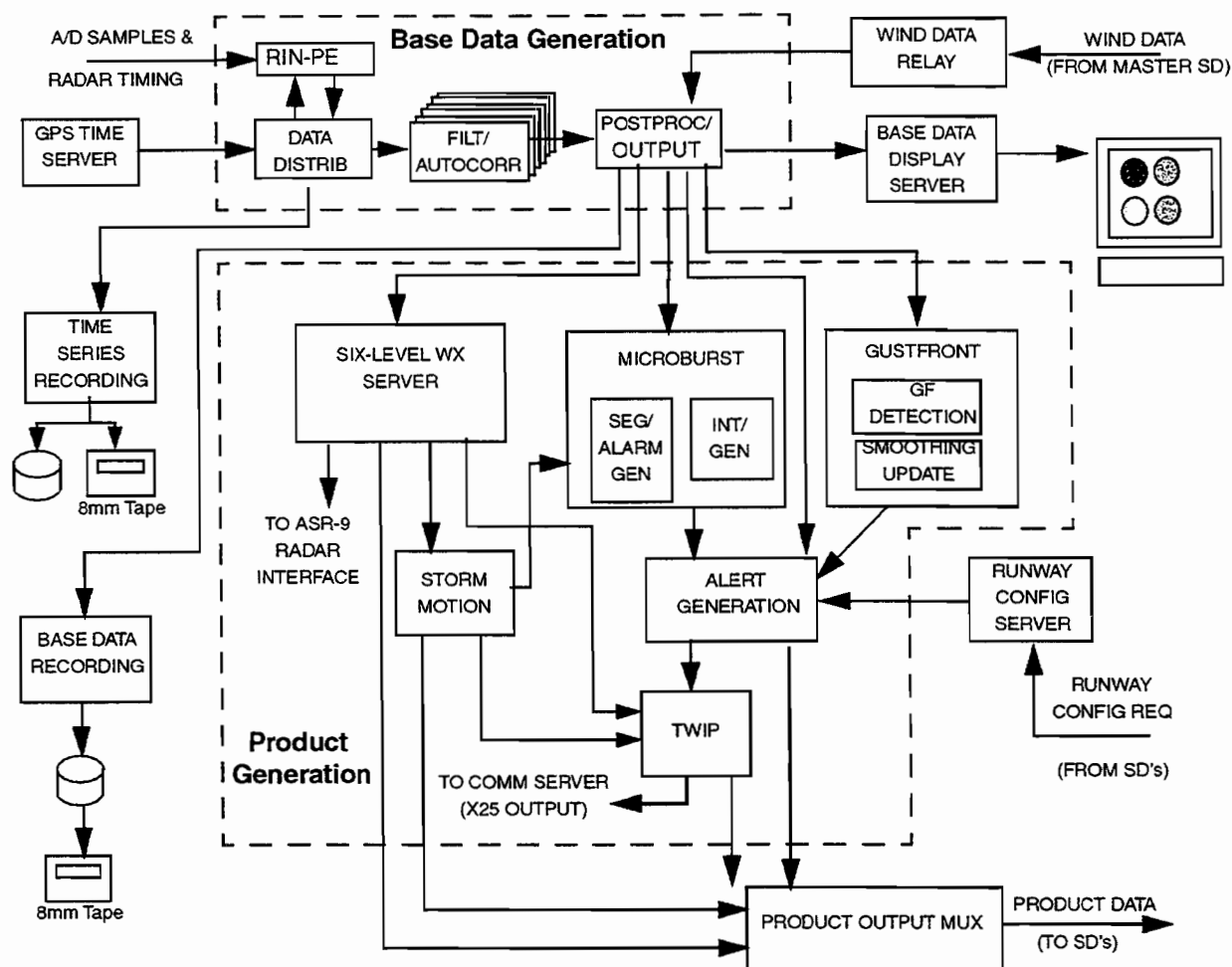
Figure 1.   Phase II WSP Software Block Diagram - Radar Site Modules

The alert generation module gathers microburst, gust front, wind sensor, and runway configuration information, determines if any microburst and gust front detections intersect the currently active airport runway arenas, and generates textual message suitable for output to the ribbon display in the tower. The messages are output to the product multiplexor. Note that the alert generator also passes information received on it's multiple input streams to the single output stream, effectively serving as a data stream concentrator. This is done for efficiency reasons. The information passed through is used downstream by the situation display when drawing the graphical representations of microbursts and gust fronts.

The Terminal Weather Information for Pilots (TWIP) module is a full implementation of TWIP using the WSP as the input data source. The TWIP module converts the six-level weather images, along with the microburst, gust front, and storm motion information, to a format suitable for output to aircraft using the ARINC network.

The Product Multiplexor module serves to gather all data destined for the SD's and output it on a sin-

gle stream. A dedicated 'relay' module in one of the SD's at the TRACON facility reads the stream and provides it to all SD's at the facility. This design results in a single transfer of data via the 128KBaud communication line between the radar site and the TRACON, as opposed to a separate transmission for each SD.

2.2 TOWER/TRACON SOFTWARE MODULES

A block diagram of the software modules running at the tower/tracon facility is shown in Figure 2. Data arriving from the radar site via the 128K communications link are fed into a relay/junction module running on the tower (master) SD, The relay/junction module is responsible for distributing product data to all other machines at the tower/tracon facility (relay function), as well as automatically switching over to a wind-data-only output if the link to the radar site should become impaired (junction function). This allows the tower SD to continue display of center-field wind information on the ribbon displays, even when the rest of the WSP system is unavailable.

The winds reader module, also running on the tower SD, reads data from the airport's LLWAS or ASOS wind sensor and outputs the data to the radar site. The data is also fed to the local relay/junction process to provide the backup wind data path.

The display module reads the input data and creates graphics and text representations of the microburst and gust front information. It also provides text-only output to any ribbon displays connected to the workstation's serial ports. User interface elements are provided for changing a variety of display parameters, such as range, background color, and overlay maps. The ability to access and modify the current runway configuration is also provided.

The product recording module taps into the product data stream and records the data to disk and/or tape. The display settings for the machine on which the recording is being performed are also archived. Sufficient storage is provided to allow for a 15-day history to be maintained. Each SD separately maintains a local product archive. The SD in the TRACON is equipped with an 8mm tape unit for permanent archival of the data.

The external user product relay module taps into the product data stream, converts the TCP/IP data packets to a serial byte stream, and outputs the data to up to 9 serial ports using the communication server in the tower equipment room.
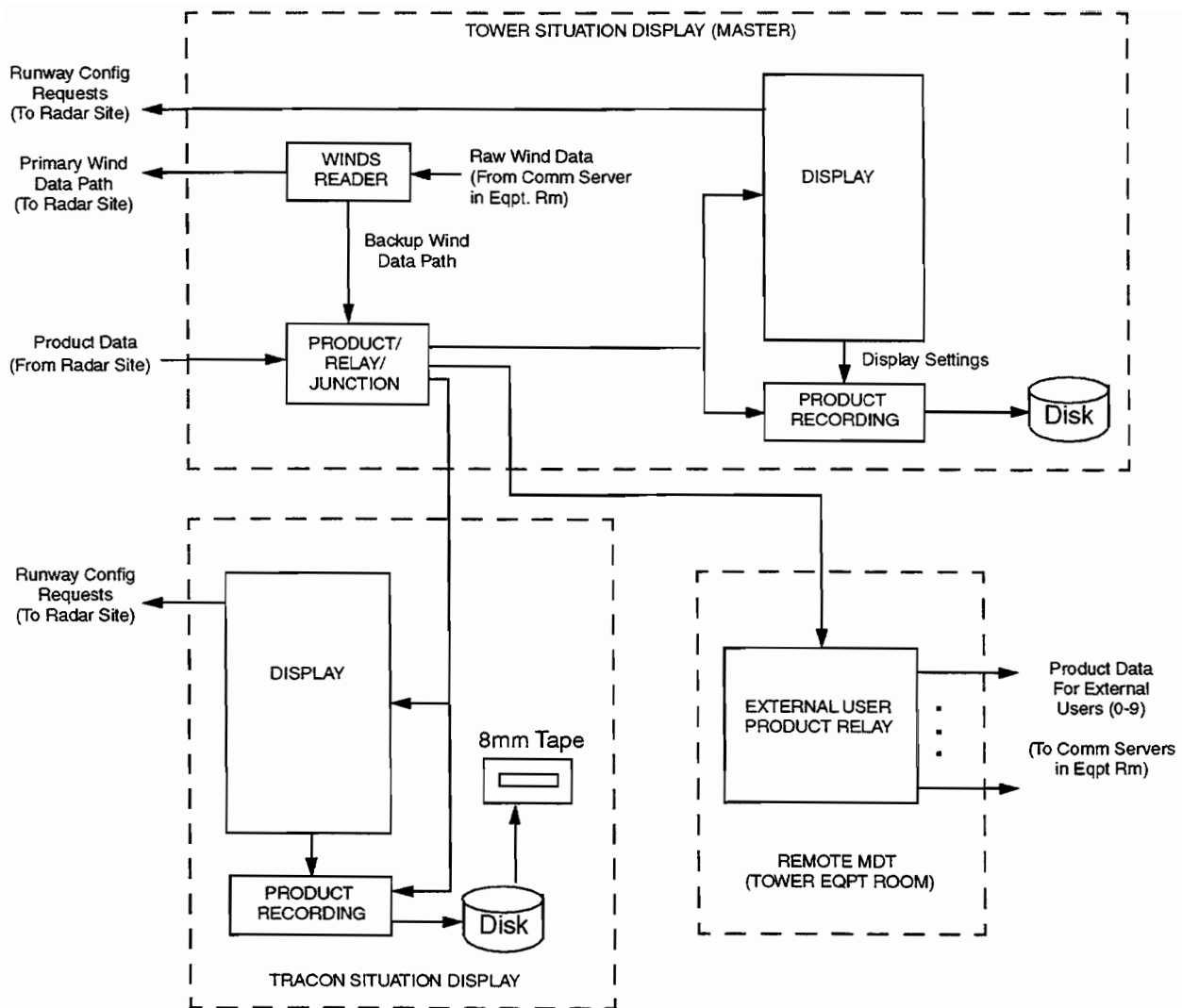
Figure 2. **Phase II WSP Software Block Diagram - Tower/Tracon Modules**

2.3 RADAR SITE HARDWARE COMPONENTS

A block diagram of the Phase II WSP prototype hardware that resides at the radar site is shown in Figure 3. The majority of the WSP functionality is contained in a single VME chassis, occupying 11 of the 21 available slots. All components are available as commercial, off-the-shelf (COTS) products. The WSP VME chassis hardware consists primarily of two board types, PowerPC-based array processing boards from Mercury Computer Systems and SPARC-based single-board computers (SBC's) from FORCE Computers. Other cards include a 128Mb bulk memory card, and a GPS satellite receiver board.

The Mercury boards are responsible for handling the time-series data I/O and performing the com-
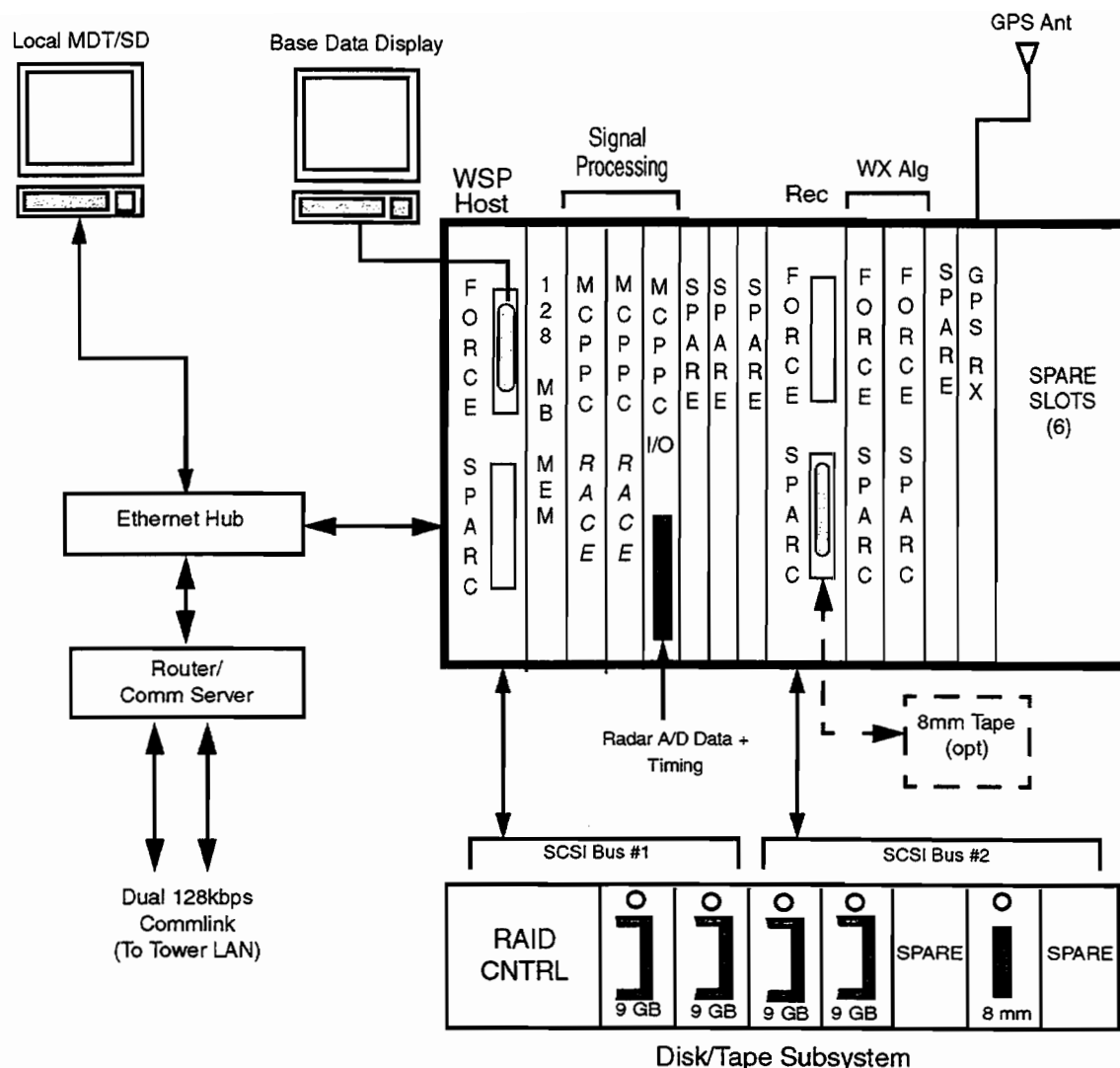
Figure 3.   WSP Phase II Prototype

pute-intensive signal processing operations required to produce the base data stream. Each single-slot Mercury board supports two daughtercards, each daughtercard containing either two PowerPC CPUs and 64MB of memory, or a single high-speed I/O interface. In the production configuration, two of the boards are populated with two daughtercards containing two PowerPC's each (4 processors per board), while the remaining board has a single, two-processor daughtercard, and a daughtercard containing a high speed interface (RIN-PE). All devices on the Mercury boards are interconnected via the 'RACEWay' a high speed (160Mb/sec.) crossbar network, allowing each device network-wide shared memory access. The RACEWay interconnect between multiple VME boards is implemented via a 4-slot RACEWay interconnect bridge (ILK-4) that utilizes the VME backplane P2 connector. In addition to RACEWay access, each device can function as a VME 32/64 master/slave, providing a large amount of flexibility for software applications.

The FORCE SBC's are used for weather algorithm image-processing, data recording, and system hosting functions. Each FORCE SBC consists of a 171 MHz/128 MB SUN compatible workstation in a

7

VME form factor, including support for two SBUS modules (requires the use of an additional VME slot). System connections (SCSI, Ethernet, serial) are available on the front panel of each card, or alternatively on the back of the chassis using an optional VME P2 interface card. Connections from SBus cards (video/ additional SCSI or Ethernet ports) are only available on the front panel. These boards are 100% software compatible with Sun's desktop machines, requiring only the addition of a VME driver to the standard Solaris 2.6 release in order to access the VME bus.

The FORCE card in VME #1 performs the 'host' function for the Mercury boards, allowing them access to disk/terminal I/O services as well as handling the software download process. This board is equipped with an graphics card & display, and also serves as the base data display. A second FORCE CPU is dedicated to time-series and base data recording. The other two boards are dedicated to the weather detection algorithms, one to the microburst and storm motion algorithms, and the other to the gust front algorithm (the most computationally intensive of the three).

A MicroMemory 6440D 128 MB memory card provides buffering for the radar data recording task, as well as a variety of other buffers used during diagnostics. The memory is dual-ported, accessible via VME or via the RACEWay. VME data transfer rates of up to 80 MB/sec. (VME64) are supported, while RACEWay transfers can operate at the full 160 MB/sec. bandwidth of the RACEWay. The RACEWay connection is implemented via the VME P2 connector. The board is expandable to 512MB, although it is not field-upgradable since the memory chips are soldered in place (for reliability) as opposed to being mounted on SIMM's.

Time tagging information is obtained from a 8-channel GPS receiver implemented on a single-slot VME card. This card acts as a VME slave, allowing other processors to obtain the current time/location information using simple VME transfers. The supplier of the card is Brandywine Communications.

A ruggedized disk/tape subsystem is used to store programs and data. The selected system is a rack-mountable unit with a RAID controller, 4 9Gb SCSI disks, and an 8mm tape drive. The unit is split into two separate SCSI buses. The first bus is connected to the RAID controller, which provides a fault-tolerant 9 Gb storage area (2 drives in mirrored configuration) for the WSP 'host' computer (the SPARC board in slot #1) All critical run-time files are stored in this area. Note that SPARC CPU's #2, #3, and #4 all boot using SPARC #1 as their file server, and they therefore share the benefit of the RAID system. The second bus in the disk subsystem is connected to the other two 9 Gb drives and the 8mm tape drive. The SPARC CPU responsible for data recording (#2) utilizes these devices to record base data. A failure of one of these devices does not impact the run-time performance of the system - the WSP's fault-detection system is notified of the failure, and the fault can be corrected at a later time.

Lastly, an optional high-speed 8mm recorder used to record the raw time-series data. The recorder connects to a fast-wide SCSI interface installed on the recording SPARC, and is capable of recording 20 Gb at a continuous 3.0 Mb/sec. rate. In the production configuration, it is anticipated that there will be a limited number of high-speed recorders, and they will be shared within each FAA region to help analyze site-specific data quality issues.

2.4 TOWER/TRACON HARDWARE COMPONENTS

The typical set of hardware at a Tower/Tracon facility is shown in Figure 4. A pair of router/comm servers located in the tower equipment room are used to provided the link to the LAN at the radar site, as well as RS232 output ports for up to nine external users. A Sun workstation in the equipment room serves as the remote MDT, and can double as a training machine for the Situation Display.

A single SD resides in the tower cab. Attached to the SD via a single serial line are 2 Ribbon Display Terminals (RDT's) from DALE electronics. Note that up to 8 ribbon displays may be daisy chained on a single serial line. A single SD also resides in the TRACON area. This SD is normally set up at the TRACON supervisor's position, along with a single RDT. Attached to the SD is an 8mm tape drive (Exabyte) to provide the capability to save the most recent 15-day product archive to tape. An Ethernet hub in the TRACON allows for additional SD's to be installed if requested by the air traffic controllers (as was the case for the early test sites).
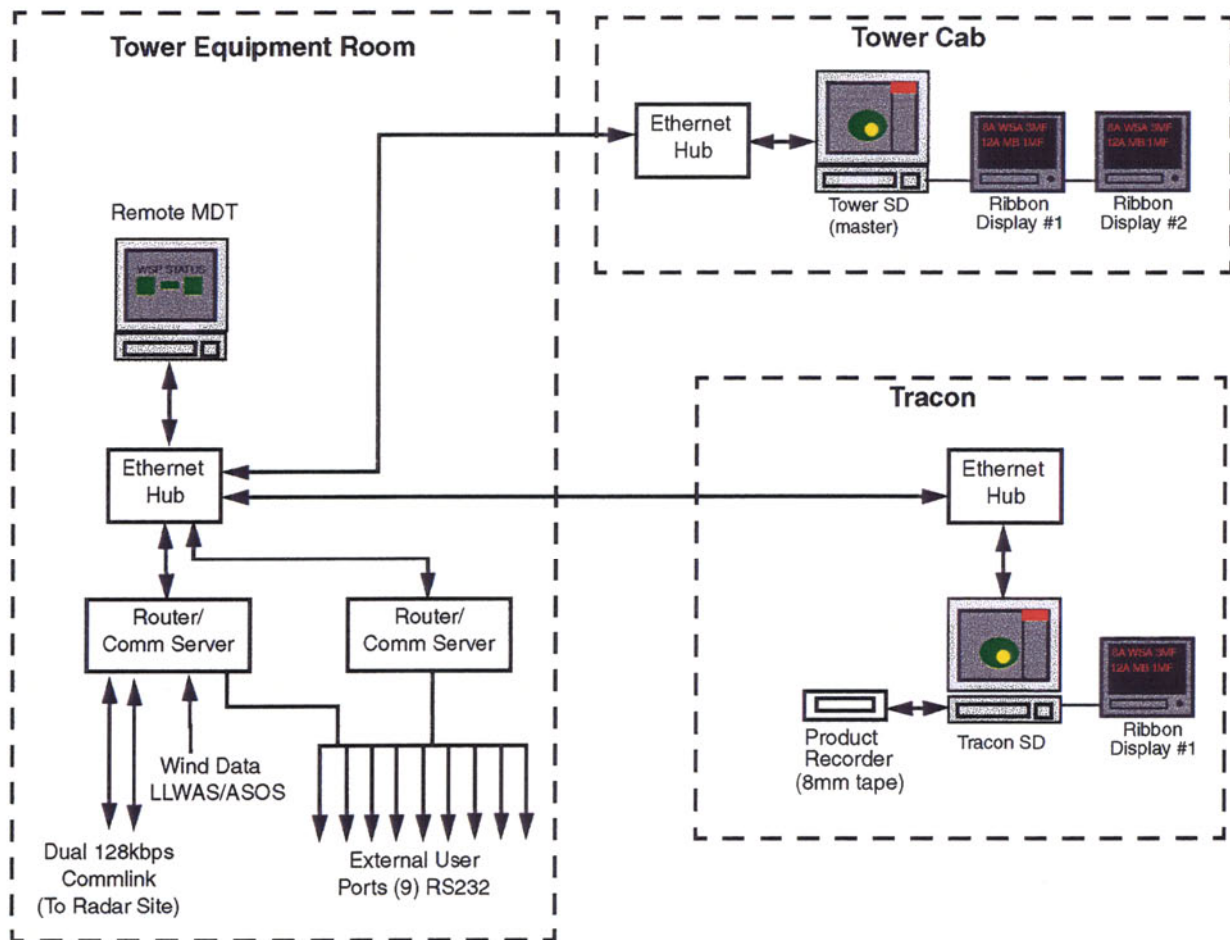
Figure 4.   *Tower/Tracon Hardware Components*

2.5 SOFTWARE ENVIRONMENT

2.5.1 Operating Systems

There are two primary operating systems used within the WSP system. The Mercury processors run MC/OS, a proprietary real-time operating system optimized for the RACEWay architecture. The FORCE single-board computers, and the external Sun workstations used for situation displays, run Sun Microsystems's version of UNIX (currently Solaris 2.6), an operating system which has historically possessed excellent development tools and has now been extended with real-time multi-tasking capabilities. Both

operating systems support the POSIX-standard for system calls [10], resulting in a large degree of code sharing between the two architectures.

Virtually all of the WSP software that runs under Solaris has also been compiled and run on a LINUX system. Although LINUX is not used in the context of the production system (with the exception of the radar interface, not described in this document), the ability to run the software on a PC is useful in the context of maintenance and working at remote sites using laptop computers.

2.5.2 Languages

The majority of code in the WSP is written in ANSI C or C++ [11,12]. Code running on or shared by the Mercury processors is written in C, while the weather algorithms running on the FORCE SBC's are coded in C++, a more natural choice for the recode of the original object-oriented Lisp-based implementation.

Another language/toolkit of significance is Tcl/Tk [13,14], a scripting language with support for rapidly implementing user interfaces. This language, in combination with C/C++ code, is used to implement the WSP's base data, analysis, and situation displays, as well as the maintenance data terminal.

Lastly, a number of UNIX shell scripts (csh) are used to implement the majority of the system startup/and shutdown tasks.

2.5.3 Software Development Tools

The software development package supplied with the Mercury boards consists of a C compiler from Metaware, Intel's assembler and linker tools, and a customized version of the Free Software Foundations debugger, 'mcgdb'. The C compiler is capable of generating highly optimized code for the PowerPC architecture, including support for the chip's single-cycle multiply-accumulate operation. The gdb-based debugger has been extended to provide support for translation of PowerPC assembler instructions and examination of registers, while retaining the familiar gdb command set. Rounding out the Mercury development environment is an optimized vector processing library, essentially eliminating the need for assembly language programming.

On the FORCE boards and Sun workstations, the GNU C/C++ compiler/debugger tool suite (gcc/ g++/gdb) serves as the primary development toolset. Although a 'freeware' product, these tools are widely used for many commercial products (witness Mercury's adaptation of the GNU debugger), and have proven to be extremely robust. As previously mentioned, the Tcl/Tk package is also used, primarily as a user interface development tool. This is also a freely available product, and, like the GNU tools, is mature and well supported (currently being maintained/enhanced by Ajuba Solutions, a commercial company run by the original developers of Tcl/Tk).

Two code analysis tools, Purify and Quantify from Pure Software, are used to detect coding errors and optimize the code running on the FORCE boards. Purify is the (UNIX) industry's most popular tool for detecting memory leaks, uninitialized memory references, and other common C/C++ programming errors. Quantify is perhaps less well known, but equally effective for code optimization tasks, allowing the user to conveniently view CPU usage at the source code level. The combined use of these tools greatly aids the production of error-free, efficient code.

3. CORE SOFTWARE MODULES

3.1 GPS TIME SERVER

The time server process reads the current time from the GPS receiver board, and stores it in a public VME location for use by other processes. This process is made necessary by the fact that only one process can read data from the GPS board at any given time, and multiple processes in the WSP require time stamps.

The primary use of the time stamp is to time stamp the data as it flows through the processor. This is done at the front end of the system by the PowerPC distributor module (see Figure 1.). Each pulse of radar time-series data is tagged with the current time using 1-second resolution. This time stamp is carried through the entire processing chain, all the way to the final outputs to the situation displays. A secondary use of the time stamp is to automatically synchronize the real-time clocks on all of the SPARC CPU's to correct for any long-term drift. The time server process is responsible for synchronizing the clock on SPARC CPU#1, and all other SPARC CPU's (in the VME chassis or external) update their clocks by periodic execution of the UNIX 'rdate' utility referencing SPARC#1 as the 'master'.

The GPS hardware is designed to tolerate occasional loss of satellite signal, and continues to provide an accurate time via it's internal reference. The return of a valid satellite signal causes the board to transparently return to normal operation.

More serious GPS errors, such as a prolonged lack of satellite signal, a non-responsive board, or a bad time stamp, cause the time server process to fall into a backup mode, where time is obtained from the on-board clock on SPARC CPU#1. The WSP will continue to function normally, with the exception of a GPS maintenance alert on the MDT.

3.2 BASE DATA GENERATION

Base data generation encompasses the radar data acquisition and input to the PowerPC processing array, subsequent clutter filtering, autocorrelation, and spatial smoothing operations, and output of base data to the downstream weather algorithms. This section illustrates the data flow through the base data generation process, providing specifics as to processor partitioning and associated data rates. As stated earlier, this document does not contain many specifics regarding the signal processing algorithms used. Readers are referred to [8] for a thorough treatment of the algorithms themselves.

The WSP outputs a variety of base data types. A summary of base data outputs with their common abbreviations is shown in TABLE 1. The abbreviations shown are used throughout the remainder of this document.

3.2.1 Time-Series Data Acquisition and Distribution

The ASR-9 radar interface supplies the WSP with a logical 32-bit data stream consisting of 10 header words and 1920 data words for each radar pulse (2 32-bit words per range gate x 960 range gates). The data stream operates synchronously with the ASR-9 range gate clock. The data is physically fed to the WSP using a 16-bit parallel cable compatible with Mercury's RIN-PE interface. Since there are two 32-bit words transferred per range gate, and the physical interface is actually only a 16-bit interface, the signals on the input cable are clocked at four times the 1.3 MHz range gate clock, or 5.2 MHz .

11

TABLE 1

Base Data Types

Type Abbrev	Description
LO_DBZ	Low beam reflectivity data (dBZ)
LO_VEL	Low beam velocity data (m/s)
HI_DBZ	High beam reflectivity data (dBZ)
HI_VEL	High beam velocity data (m/s)
DUAL_DBZ	Dual beam reflectivity data (dBZ)
DUAL_VEL	Dual beam velocity data (m/s)
FLAGS	Data quality flags. (AP, second-trip, clutter breakthrough)

The 10-word header included within each radar pulse contains information such as antenna position, RF switch status bits, counters incrementing at the1.3 and 10.3 MHz ASR-9 clock rates, and a pulse counter that increments once per pulse. Words containing fixed bit patterns (0xaaaa5555, 0x5555aaaa) are also included for error checking/synchronization purposes. Data for each range gate consists of the A/D samples from the WSP's receiver and ASR-9 target channel receiver, and a bit containing the position of the front-end high-speed beam switch for the range gate. The basic layout of the pulse data is shown in Figure 5. See Appendix C for a detailed description of the time-series data format.

In addition to the data lines, the radar interface provides clock, SYNC, and VALID signals, as required by Mercury's RIN-PE interface. The SYNC signal coincides with the first header word in a radar pulse, and is used by the WSP software to synchronize to the start of a radar pulse. The VALID signal is set to true for the 10 header words and the 1920 data words, then set to false during the remainder of the inter-pulse period. This signal is used to 'trim off' data following the 1930'th word of each pulse (range gate 960), preventing excess data words from being accepted into the RIN-PE FIFO. The use of VALID groups the data into 1930 word, fixed-size frames, regardless of the radar PRF. This eliminates the need for a SYNC operation for every pulse (not guaranteed to have sufficient inter-pulse time to operate the RIN-PE in that manner), since the position in the data stream is known following the first SYNC.

It should be noted that the above strategy of using fixed-size frames does not allow for access to data during the radar pulse dead time. If access to such data is desired, as it may be at some point in the future to allow processing of ASR-9 test targets, then the software controlling the RIN-PE will need to be modified to ignore the SYNC and VALID signals, accept variable-sized pulses into the input buffers, and perform the synchronization to the start of the pulse entirely in software. It does appear that there is sufficient CPU bandwidth to implement such a strategy on the PowerPC node that controls the RIN-PE.

The RIN-PE interface does not have CPU of it's own, nor any memory other than 4Kx32-bit FIFOs. Because of this, one of the Mercury PowerPC nodes must be used to control the interface and serve as the distributor of data to the other nodes. This node also distributes data to the time-series recording function, and contains a software time-series simulator for testing purposes. The computational load of the signal processing algorithms is spread out over N processing nodes by feeding each node a range 'slice' of the

incoming data via a 128-pulse circular buffer. To minimize per-pulse transfer overhead, each transfer contains 16 pulses. Beyond the range of the wind shear detection algorithms (240 range gates, or 15 NMI), not all range gates are processed. Processing only every 4th gate beyond a range of 15 NMI is sufficient to generate accurate six-level weather estimates. Therefore, a secondary job of the data distribution process is to decimate the data in range as appropriate when outputting the data to the compute nodes. A block diagram of the data distribution throughout the front end is shown in Figure 6.
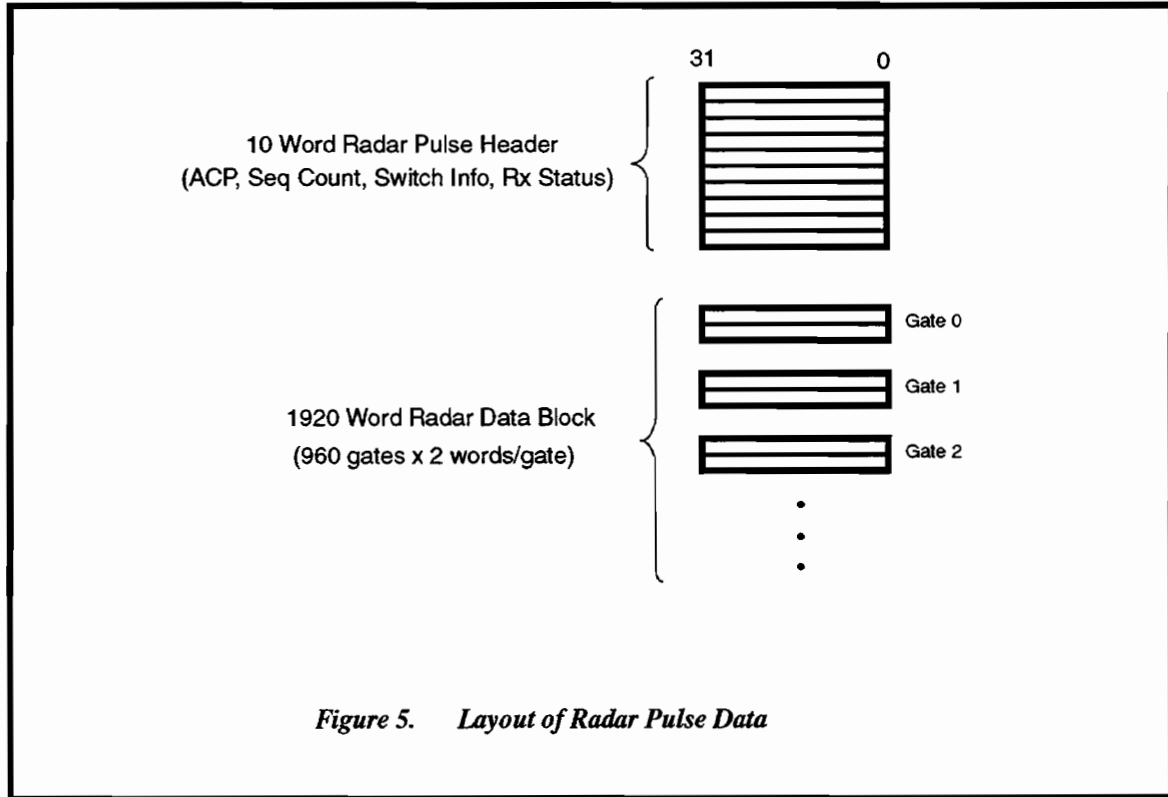
Figure 5. *Layout of Radar Pulse Data*

The current antenna position, as derived from the incoming radar headers, is transmitted to the 'collector' node responsible for base data range-slice reassembly and post-processing. This is used to drive the collector task, and also provides a 'reference clock' for determining the peak processing delay in the compute node array. Lastly, a time-series simulator is provided for testing of the basic clutter filtering/autocorrelation functionality. The simulator is interactively controlled via the MDT, allowing a variety of user-specified test scenarios to be injected at the front end of the system.

3.2.2 Clutter Filtering and Autocorrelation

Each node in the PowerPC compute array monitors it's onboard input queue, converting the data as it arrives to IEEE single-precision floating-point format and correcting for STC attenuation. The 1.3 MHz clock in the radar pulse headers is monitored to determine the current PRF set and locate CPI boundary crossings. When a 27-pulse 'extended' CPI (ECPI) of data is ready for processing, the clutter filtering and autocorrelation operations are performed and the results transmitted to a shared memory buffer on the 'collector' node for range-slice reassembly and post-processing. A block diagram of this sequence is provide in Figure 7.

13

Radar Pulses From Digital Interface
(1930 Words x 32 bits)

RIN-PE

4 K-word
FIFO

DMA

IQ Data
Simulator

Array of PowerPC Compute Nodes
(Clutter Filtering, Autocorr, Z, V, AP Correction)

'Collector'
PowerPC Node

DMA
Cmds

Raw Pulse
Buffer

Reassembly
Buffer

Control Program

Range
Slice
Buffers

DMA

Current Antenna Position

"Distributor' PowerPC Node

128 MByte
Recording Buffer
(VME Board)

Base Data
Simulator

Spatial Filtering

Wind Sensor Data

(from VME ring buffer)

Base Data
Formatting

Base Data Output

Figure 6. **Base Data Generation Data Flow**
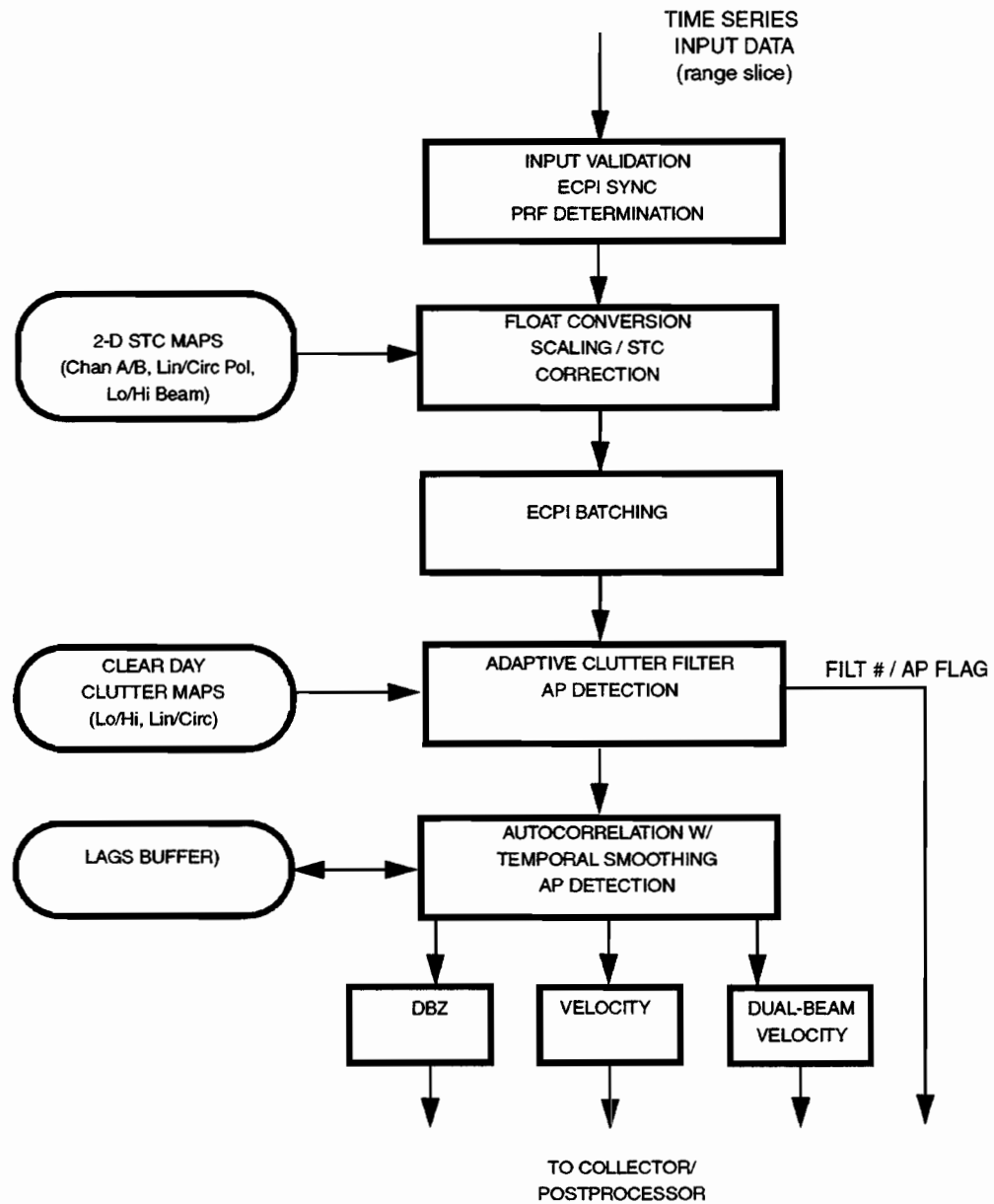
Figure 7. *Clutter Filtering and Autocorrelation*

3.2.3 Post Processing and Base Data Output

The collector task monitors the current antenna position, and performs post processing on sectors of base data at (current sector - 32), where a sector represents a 1.4 degree base data 'wedge'. This 45 degree delay allows time for the compute nodes to complete all processing operations for a given sector. Although the collector's main task is the output of base data to its various destinations, the output job requires only approximately 5% of the CPU. To take advantage of this relatively unused PowerPC node, several of the less intensive signal processing operations are carried out on the collector node prior to the output of base data. The operations include completion of the DUAL_DBZ and FLAGS products, as well as the spatial filtering of the DUAL_DBZ and DUAL_VEL products (the other base data products are not subject to median filtering). An additional benefit of carrying out the spatial filtering on this node is that the data is continuous in range, no longer broken up into range slices. This negates the need for overlapping the range rings within each compute node in order to properly implement the spatial filter. A block diagram of the postprocessing operations is shown in Figure 8.

The collector task is also responsible for integrating the wind sensor input into the base data output stream. Wind data is made available to the collector via a VME ring buffer that is filled by a process running on SPARC board #1 (it reads the data from the tower via a TCP/IP socket connection). Integration of the wind data into the base data at this point allows for base data recordings to contain *all* the data necessary for repeatable, synchronized base data playbacks.

A base data simulator is also implemented on this compute node. The base data simulator provides for increased programming flexibility when compared to the time-series simulator, since the amount of data that must be generated is approximately an order of magnitude less. This simulator can produce reasonably realistic storm cells with associated microbursts and gust fronts. Signal contamination due to second-trip echoes, AP, and ground clutter breakthrough can also be specified. In general, this is the simulator of choice when testing the basic functionality of the weather detection algorithms. The simulator is controlled via the MDT, where the system operator is presented with a number of canned scenarios, as well as the option of creating new custom scenarios.

3.3 BASE DATA DISPLAY

The base data display software converts the polar reflectivity, velocity, and flags data into cartesian images, and displays the images on the main WSP X-windows console. Two processes are involved, the base data display server process and the base data display daemon process. The server process reads the data from the VME-resident shared memory buffer, performs the cartesian conversion, and sends the images to the display daemon via shared memory. The display daemon reads the incoming images, renders the images on the display, and responds to user display requests (zoom, pan, etc...). The basic update rate of the display is every two antenna scans (~9.2 seconds) (a more frequent update is of limited use given the alternating beam strategy employed by the WSP receive chain). An example of the base data display is shown in Figure 9. The user can view any four of the base data products simultaneously, along with the appropriate color scales. Readouts track the cursor range/azimuth, as well as the underlying data value in each window. All four windows can be independently zoomed and panned, or can be zoomed/panned as a group.

Z,V, FLAGS FROM PowerPC COMPUTE NODES 1.2,3....N

COLLECTION BUFFER
(SHARED MEMORY)

CURRENT ANTENNA POSITION
FROM DISTRIBUTOR CPU

INPUT PROCESSING

DATA FOR (ANTPOS - 45) deg

DUAL DBZ PROCESSING
(HI/LO BEAM COMPOSITE)

SPATIAL FILTERING

WIND DATA
FROM VME BUFFER

BASE DATA OUTPUT
(VIA VME RING BUFFERS)

MB ALG   GF ALG   WX SERVERS
(POLAR/CART)   DISPLAY   RECORDING
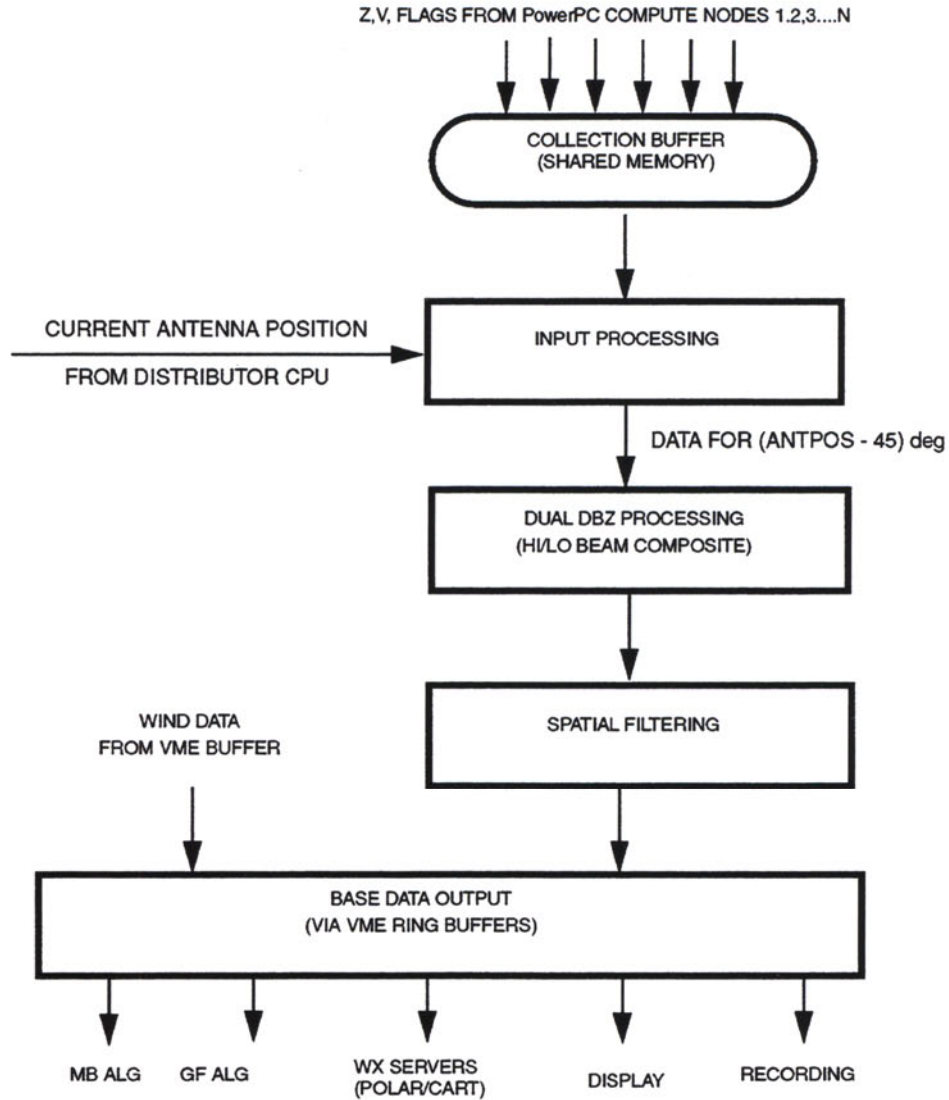
Figure 8.   Base Data Collection /Post-Processing / Output

Figure 9.   Base Data Display of Simulated Microburst and Gust Front
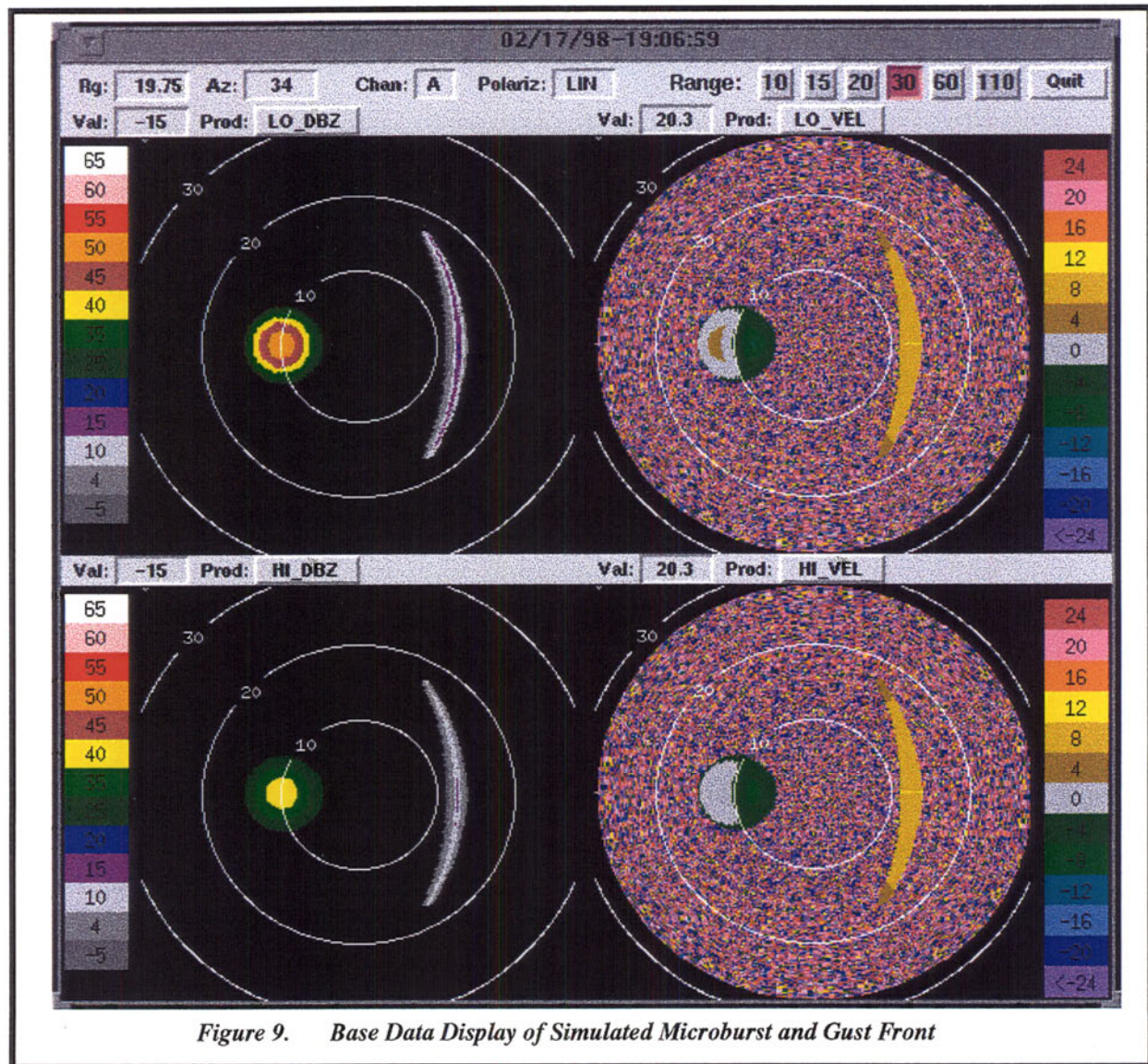
The above example also illustrates the functionality of the base data simulator.   This simulated scenario contains a single storm cell which is producing a microburst as it tracks to the east at 5 m/s. A gust front has propagated out in front of the storm cell, presumably simulating an earlier microburst 'pulse' by the isolated storm. Note the ability to specify different reflectivity/velocity patterns in the low and high beams, a necessity when testing the detection algorithms.

3.4 SIX-LEVEL WEATHER SERVER

The six-level weather server converts the WSP data from dBZ units to the six-level equivalent and performs spatial smoothing and contouring operations similar to those used in the original ASR-9 weather channel. The resultant six-level data are output to the ASR-9 via the radar interface, and are also transmitted to the storm motion algorithm, the TWIP algorithm, and the situation displays (via the product multiplexor).

18

The inputs and outputs of the six-level weather server are shown in Figure 10. A new set of polar base data is input to the server every other scan (~9.2 sec. update rate). The resolution of the data is 1/4 NM in the range dimension, and 1.4 degrees in the azimuthal direction (360/256). Following the temporal and spatial smoothing, contouring, and NWS level conversion operations, three separate TCP/IP output streams are generated. The stream directed to the ASR-9 interface consists of 120x256 polar images with a range resolution of 1/2 NM (same resolution as the ASR-9 CD-2 format), and an update frequency of 6 scans (~28 sec.). The polar images generated for this stream are aligned to magnetic north, the ASR-9 convention.

The SD/TWIP output stream consists of 480 x 480, 1/4 NM cartesian images, updated every 6 scans. The images are aligned to magnetic north, as the SD and TWIP both present their displays using the magnetic north convention. Note that although this stream is logically a single output stream, multiple 'clients' can connect to it due to the use of the server-client TCP/IP software described in the software libraries section of this document. This is true for essentially all the TCP/IP data streams used throughout the WSP system.

The storm motion output stream consists of 240 x 240, 1/2 NM cartesian images, updated every 6 scans. In this case, the resolution is reduced from the input resolution of 1/4 NM to reduce the size of the image required to cover the full radar range. The 1/2 NM resolution is sufficient for the storm motion algorithm to accurately track storm cells, and the reduced memory footprint due to the smaller images is significant since the algorithm maintain a 5-10 minute history of images in memory to compute the motion field. Note that these images are aligned to *true* north, as WSP algorithm processing (as opposed to display) is generally done in the true north domain.

The mapping of the data from dBZ to NWS level is shown in TABLE 2 The table reflects the standard thresholds used by the NWS. The thresholds are, however, set up as variable site parameters in the event that non-standard threshold levels are desired.

TABLE 2

DBZ To NWS Six Level Weather Mapping

DBZ Value	NWS Level
< 18	0
18 - 30	1
30 - 41	2
41 - 46	3
46 - 50	4
50 - 56	5
> 56	6

```
ASR-9 INTERFACE OUTPUT STREAM

120 (RNG) x 256 (AZ) POLAR IMAGES
1/2 NM BIN RESOLUTION
MAGNETIC NORTH ALIGNMENT
6-SCAN (~28 SEC) UPDATE RATE


SD/TWIP OUTPUT STREAM

480x480 CARTESIAN IMAGES
1/4 NM BIN RESOLUTION
MAGNETIC NORTH ALIGNMENT
6-SCAN (~28 SEC) UPDATE RATE


STORM MOTION OUTPUT STREAM

240 x 240 CARTESIAN IMAGES
1/2 NM BIN RESOLUTION
TRUE NORTH ALIGNMENT
6-SCAN (~28 SEC) UPDATE RATE
```

BASE DATA (VME RING BUF)
240 (RNG) x 256 (AZ) POLAR IMAGE
1/4 NM BIN RESOLUTION
TRUE NORTH ALIGNMENT
2-SCAN (~9.2SEC) UPDATE RATE

SIX-LEVEL
WX SERVER

Figure 10.   Six-Level Weather Server Input/Output

3.5 STORM MOTION ALGORITHM

The ASR-9 storm motion algorithm computes the motion of operationally significant storm cells and outputs a set of motion vectors as well as the 10 and 20 minute extrapolated positions of the storm cells. The vectors and storm extrapolated positions (SEP's) are displayed on the SD (see Figure 20.) as a traffic planning aid. The algorithm also produces a 'gridded vector' product, essentially a pixel-by-pixel motion field the same size as the input image. The gridded vector product is utilized by the microburst algorithm.

A block diagram of the algorithm is shown in Figure 11. Input to the algorithm consists of a stream of cartesian six-level weather images, nominally spaced six scans (~28 sec.) apart. Each image is 240x240 pixels, with a pixel resolution of 0.5 NM. The algorithm is partitioned into three major functional blocks, all running within a single process. The cross-correlation tracker cross-correlates pairs of images and determines a motion field by maximizing the output of the correlation. Because it is difficult to accurately determine storm motion over a 28-second interval (insufficient SNR), the algorithm uses a sliding 5-minute window, where the images being compared are always at least 5 minutes (nominal VSP setting) apart. The second functional block delineates storm regions via a mixture of contouring and a pixel-by-pixel boundary detection. The third stage of the algorithm utilizes the outputs of the previous two stages, creating a set of motion vectors for each storm cell as well as a set of SEP's. These data are output to the SD and the TWIP algorithm once every 28 seconds. The gridded vector motion field produced by the cross correlation tracker is output via a separate stream to the microburst algorithm, to allow that algorithm to remove motion effects from it's storm cell feature detectors.

SIX-LEVEL WX IMAGES
240x240x0.5NM
28 SEC UPDATE RATE
TCP/IP STREAM

CROSS CORRELATION
TRACKER

STORM ANALYSIS

'GRIDDED VECTOR'
MOTION FIELD

STORM VECTOR/SEP
GENERATOR

MOTION GRID OUTPUT
TO MICROBURST ALGORITHM
28 SEC UPDATE RATE
TCP/IP STREAM

VEC/SEP OUTPUT TO SD/TWIP
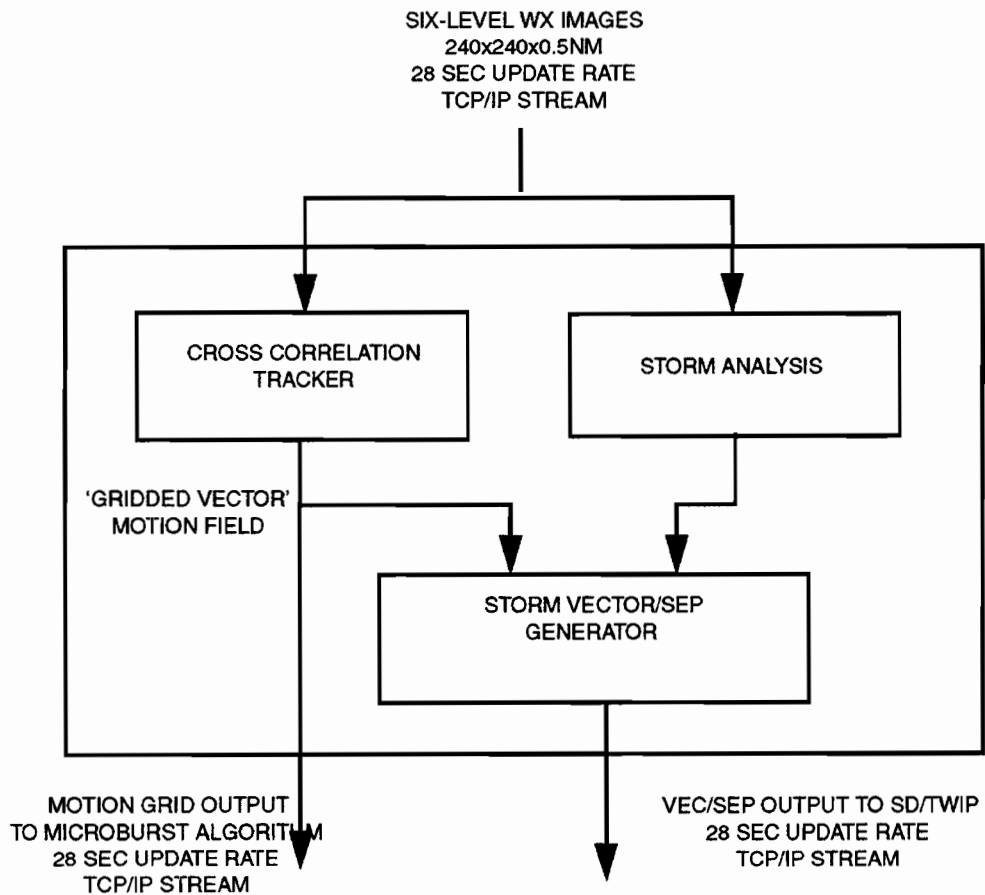28 SEC UPDATE RATE
TCP/IP STREAM

Figure 11.   Storm Motion Algorithm Block Diagram

21

3.6 MICROBURST ALGORITHM

The ASR-9 Microburst Detection Algorithm (AMDA) processes the dual-beam velocity and reflectivity base data, detects regions of low-level divergent wind-shear, and outputs the detections to the WSP's situation display alert generation module. The algorithm is divided into two threads of execution, a low-latency segment/alarm generation thread that runs at the same 4.6-second update rate as the antenna itself, and a slower, 'interest' generation thread that runs once every 12 scans (~55 sec.). The low-latency thread is intended to quickly detect the onset of wind shear utilizing primarily the velocity field, while the interest generation thread extracts microburst-producing storm cell features that change more gradually, such as local maxima in the reflectivity field and cell edge location. These additional features are used to validate and/or enhance microburst signatures detected in the velocity field, providing an increased level of confidence in the final output.

A block diagram of the algorithm is shown in Figure 12. The segment/alarm generation thread receives base data via a shared-memory ring buffer, and accumulates single scans of data in a scan buffer. The products contained in the base data stream vary with the type of scan. The DUAL_DBZ, DUAL_VEL, and FLAGS products are sent on every scan, whereas products only used by the interest generation task (LO_DBZ, LO_VEL, HI_DBZ, HI_VEL) are sent only every 12th scan. When the segment generation thread sees the extra products in the input stream, it forwards the scan to the interest generation thread via a second shared-memory ring buffer. The second input to the interest generation thread, the storm motion gridded vector motion field, arrives via a TCP/IP socket connection to the storm motion algorithm.

Although the microburst algorithm is split into two separate threads of execution, it really consists of three separate stages. The first stage of microburst detection occurs in the segment/alarm generation thread, and detects regions of divergence in the velocity field for each 4.6-second scan. The second stage occurs in the interest generation thread and results in a new combined interest image every 12 scans. The final stage, alarm generation, combines the segments and interest features, performs some temporal smoothing, and outputs a set of microburst information every 6 scans (~28 seconds).

AMDA output is broadcast over a TCP/IP socket to the alert generation module, which generates the textual warnings for the ribbon displays as well as messages directing the situation displays to 'light up' a runway when it is being impacted by a microburst. The alert generation module outputs the text messages, along with the original AMDA alerts, to the SD's via the product multiplexor.

During the development of the AMDA algorithm, an analysis display was developed to allow the user to view the reflectivity and velocity input data, the intermediate interest images, and the detected microburst features. An example of the display is shown in Figure 13. The display was built using the same low-level facilities as the WSP base data display, and, like that display, can be interactively zoomed and panned in a number of ways. The display is capable of running in real-time, with relatively modest CPU/ memory requirements, and has been included as part of the production system. The display runs is accessible on a dedicated virtual desktop on the main WSP base data display.

Base Data Input Stream
(VME Ring Buffer)

Storm Motion Input Stream
(TCP/IP)

Scan Buffer

DBDZ, DBV
LBDZ, HBDZ
FLAGS

(55 sec. update)

Speed-Inhibit

DBDZ, DBV
FLAGS

(5 second update)

Peakiness

Absolute DBZ

Cell Edge

Zero-Crossing

Non-Wx Mask

Generate
Combined
Interest

Divergence
Segment
Detection

**Interest
Generation
Thread**

**Segment/Alarm
Generation
Thread**

Combine Segments
w/ Interest Features

Generate Alarms

Microburst Detection Output
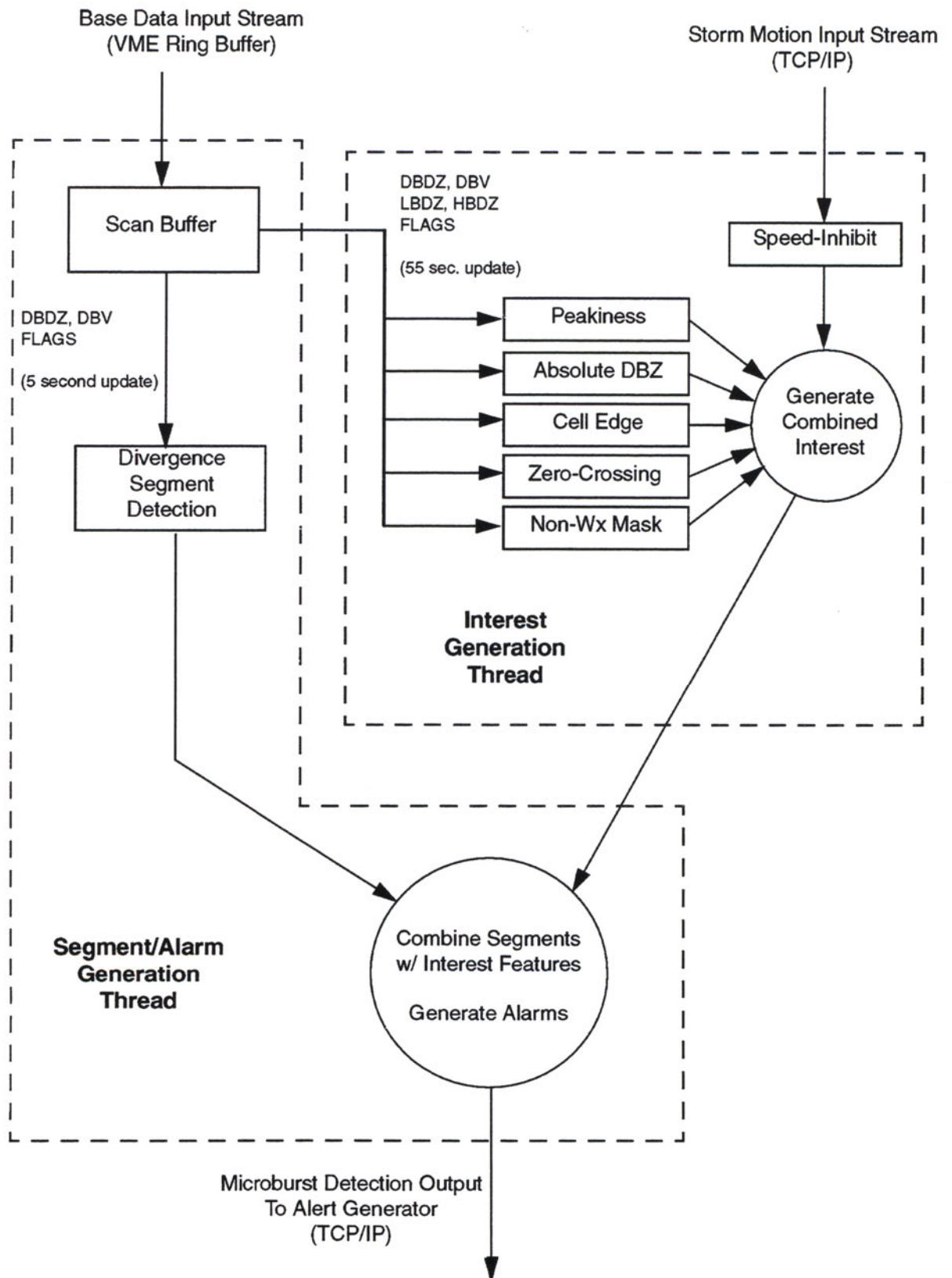To Alert Generator
(TCP/IP)

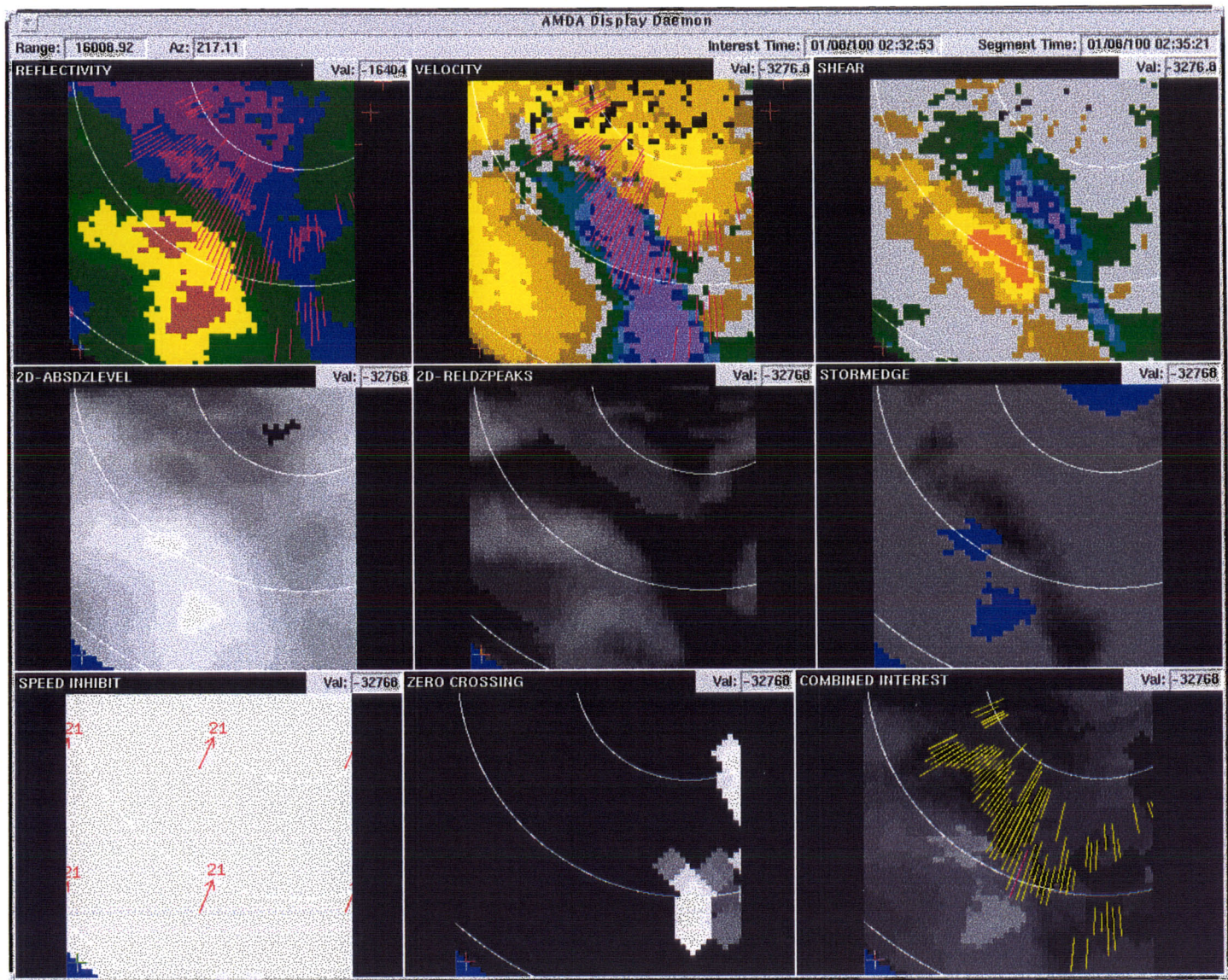*Figure 12. AMDA Flow Diagram*

Figure 13.   AMDA Analysis Display

3.7 GUST FRONT ALGORITHM

The ASR-9 Machine Intelligent Gust Front Algorithm (MIGFA) processes the reflectivity and velocity base data, detects thin-line features characteristic of gust-front events, and outputs any detections to the WSP's situation display. Like the microburst detection algorithm, MIGFA is split into two tasks, the basic detection task, and a second, 'gust front update', task. The basic detection process performs the computationally-intensive image processing portion of the algorithm, and cycles once every 24 scans (~2 min.). The second task performs some spatial smoothing, and outputs one-minute 'updates' of gust-front detections to the situation display. Partitioning the algorithm in this manner allows for the display to be updated in a smooth and deterministic manner without requiring the computational portion of the algorithm to be run at one-minute intervals.

A block diagram of the algorithm is shown in Figure 14. The detection stage of the algorithm reads base data input from a VME-resident buffer at two-minute intervals. Once an entire scan has been read in, a set of feature detectors is run on the data, producing a set of 'interest images'. Each feature detector is essentially a 2-dimensional matched filter, and the interest image values correspond to the filter outputs. The reflectivity thin-line detector 'rings' on 0.5-2.5km wide features with reflectivities in the -5 to 20 dBZ range. The velocity variance thin-line detector is similar, matching thin-lines of coherent signature in the velocity field. The high-level weather detector locates regions where the power return from the high beam significantly exceeds that in the low beam, an indication that the region is *not* part of a low-level wind shear event. The motion detector is used to differentiate moving thin-lines (actual gust fronts) from stationary thin-line features (ground clutter breakthrough, road traffic, etc...). A velocity convergence detector provides additional wind-shift information in cases where the radar is producing valid wind estimates both ahead and behind a gust front. A non-weather mask is applied to the output images of all the feature detectors to mask out regions of clutter breakthrough, second-trip, and anomalous propagation. A feedback loop containing information about previous detections is used to sensitize regions of probable gust front activity. The interest images from all the detectors are then combined using a set of fuzzy logic weighting rules, and passed to the feature extraction step. The feature extraction step performs a pixel-wise threshold on the final interest image, and creates gust front detections for any groups of image pixels matching the known gust front signature. Current detections, as well as a set of 1-min. forecasts extending 20 minutes into the future, are output to the gust front update task.

The gust front update task reads the detections/forecasts from the detection task, and smooths the gust front curves using a cubic-spline smoothing algorithm. The smoothed detections are then output to the SD at a steady one-minute update rate. Each one-minute output contains the 1-minute detection or forecast matching the current time, plus the forecasts for 10 and 20 minutes into the future. The SD displays the current detection as a solid purple line, while the 10,20-min. forecasts are displayed as dashed purple lines. As mentioned above, the main reason for this task is to provide for a smoothly updating situation display without placing awkward scheduling constraints on the image processing portion of the algorithm.

MIGFA, like AMDA, supports an analysis display tool running in a dedicated virtual desktop on the main WSP base data display. An example of the display is shown in Figure 15.

Base Data Stream

Center Field Wind
Data Stream

Gust Front Detection Task

LO_DBZ, LO_VEL, HI_DBZ, HI_VEL,
DUAL_DBZ, DUAL_VEL, FLAGS
(2min update rate)

Scan Buffer

Feature Detection

Reflectivity Thin Lines

Velocity Variance Thin Lines

Thin-line motion

Velocity Convergence

High-Level Weather

Detection History

Generate
Combined
Interest

Feature Extraction

Detection Feedback

GF Detections/Forecasts @ ~2-min. update rate

Gust Front Update Task

Gust Front Update

Current GF Detections + 10, 20 min. Forecasts to SD
1-min. update rate

Figure 14.   Gust front Detection Algorithm Block Diagram
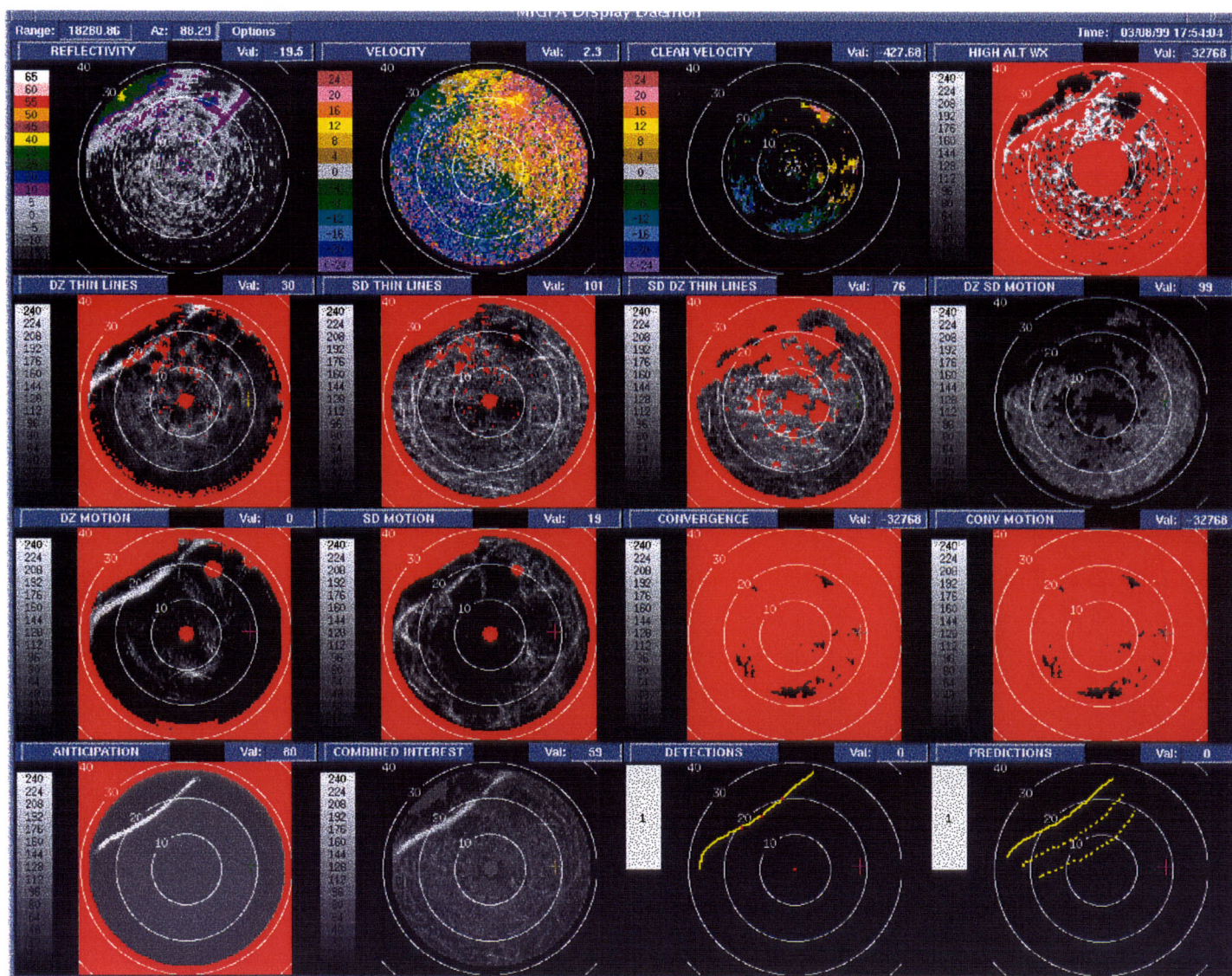
Figure 15. MIGFA Analysis Display

3.8 ALERT GENERATOR

The alert generator module combines the output of the microburst and gust front algorithms with runway configuration information to produce textual alerts for the ribbon displays. Put simply, a microburst or gust front that intersects an active runway ARENA (AREa Noted for Attention) will result in an alert message being generated for that ARENA. A step termed 'shear integration' is also performed to reduce the magnitude of events that only partially intersect a runway arena. The alert generator also reads the wind data via a VME ring buffer base data feed, and incorporates the center field wind data into the current alert message. Note that the alert generator is continually outputting messages, even when no hazardous condition exists. In general, a new alert message is created and output whenever a new microburst (~28 sec. update rate), gust front (~55 sec. update rate), or wind data (10 sec. update rate) record is received. This results in an output message at *least* once every 10 seconds.

A block diagram illustrating the alert generator input and output streams is shown in Figure 16. For efficiency reasons the alert generator serves a secondary role as a data stream concentrator. Not only are new alert messages created and output, but the original input data are also included via a pass-through and merge mechanism.
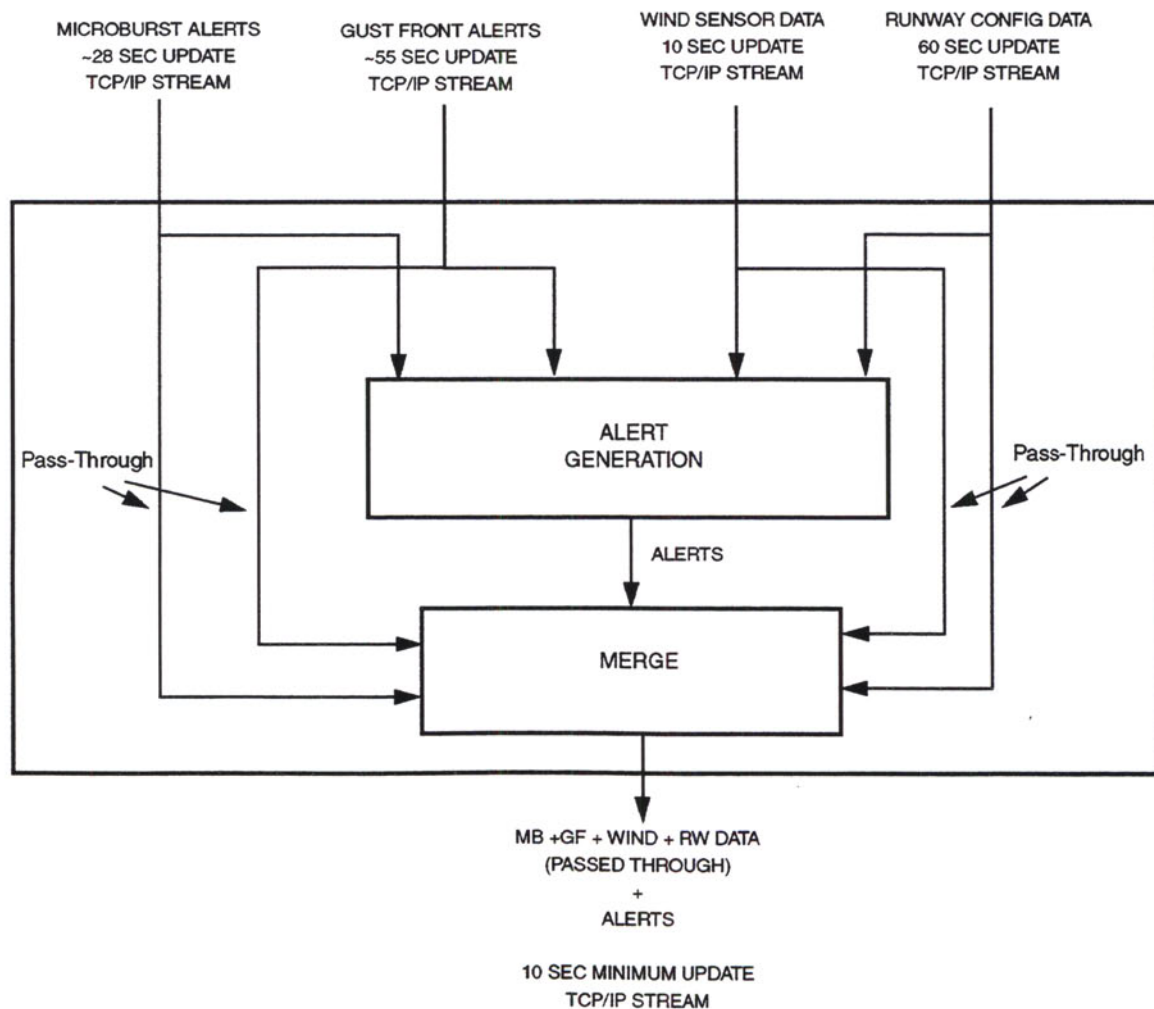
Figure 16.   Alert Generation Block Diagram

3.9 TERMINAL WEATHER INFORMATION FOR PILOTS

The Terminal Weather Information for Pilots (TWIP) module converts the WSP product data to a form suitable for display on ACARS cockpit displays and cockpit printers. TWIP data are sent to the ARINC database via the FAA's NADIN II packet-switched network, from where they can be uplinked to aircraft on demand. TWIP output from the WSP is compatible with the TDWR/ITWS TWIP product, providing a consistent product for all TWIP users.

The TWIP text message intended for cockpit displays (20 characters wide by 10 lines high), is shown in Figure 17. The left side of the figure shows the weather situation and the right side shows the corresponding text message.

The first two lines indicate that the message is for Washington National Airport (DCA) and the universal time is 18:10. In this case a 30 knot microburst is impacting one of the runways. Moderate (level 2) precipitation is touching the airport and extends from the north through the east; heavy precipitation (level 3 or greater) is 1 nm northeast of the airport. The storm is moving west at 15 knots.

An example of the TWIP character graphics depiction is shown in Figure 18. In this case there is a microburst-producing cell to the west of the airport. The moderate precipitation is indicated by '-', the heavy precipitation by a '+', and the microburst by the letter 'M'. Note that the attenuated precipitation symbol, '.', is not supported by the WSP due to the systems relative immunity to attenuation when compared with a TDWR operating at a wavelength of 5 cm wavelength. A gust front impacting the airport in this case is delineated using the character 'G'. The runway location is indicated by the 'X' symbols, except where the gust front impacts them as indicated by an '*'. A scale is provided in nautical miles in the horizontal and vertical directions along with a key to the symbols. Lastly, a textual storm motion information string is provided on the bottom line of the printout.

A block diagram of the WSP-TWIP software modules is shown in Figure 19. The processing is split into two separate processes, a main processing task and a NADIN II output task. Data enters the main task via three TCP/IP streams. Six-level weather and storm motion data arrive on separate streams, while microburst, gust front, and runway alerts arrive via a single, third stream originating from the alert generator module.

The software within the main process is broken down into four major modules. The three modules on the left side of the figure comprise the text message generation process, while the single module on the right is responsible for the character graphics generation.

The Storm Cell Detector module detects storm cells in the vicinity of the airport by contouring areas of higher reflectivity. Storm cells with reflectivities exceeding NWS levels 2 and 3 are reported as moderate and heavy precipitation, respectively. The Storm Impact Processor determines if a storm cell is currently impacting or is expected to impact the airport and approach/departure areas, and if so, what the intensity is.

The Text Message Generator combines the output of the Storm Cell Detector and the Storm Impact Processor with the runway alert information and generates the TWIP text messages.
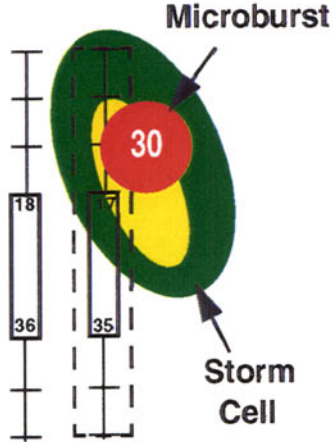
Weather Situation	TWIP Text Message
	DCA 1810 TDWR TERMINAL WX INFO *MICROBURST ALERT 30 KT LOSS BEGAN 1805 -STORM(S) ARPT N-E MOD PRECIP 1 NM NE HVY PRECIP

Figure 17.   Example of TWIP Text Message

Weather Situation	TWIP Character Graphics Map

	DCA 1810 MAP 15NM
	
	WSP TERMINAL WX INFO
	
	15 10 5 NORTH 5 10 15
	
	10 --
	
	            ------- 	
	
	        ----++--- G
	
	5 ---+++++--- G
	
	        ---++MM+-- X X G

Figure 18.   Example of TWIP Character Graphics Depiction

The Character Graphics Generator processes all input data types, with the exception of the runway alert data, and generates the character graphics map. Output from the Character Graphics Generator and the Text Message Generator is output to the NADIN II reformatter/output task and the WSP SD via TCP/IP. (Although not a strict requirement, TWIP display capabilities are provided on the SD as a convenience for tower personnel).

3.10 PRODUCT DATA MULTIPLEXOR

The Product Data Multiplexor serves as a data stream concentrator (fan-in), ingesting data from the weather algorithm and runway configuration TCP/IP data streams and multiplexing it to a single TCP/IP output stream. This reduces the number of connections required by the SD's and product recording module, and thereby simplifies system maintenance and fault isolation. It also helps to guarantees faithful playback of recorded data, since the recording software will record and play back the data packets in the same order as seen by the SD during real-time operation.

3.11 PRODUCT DATA RELAY/JUNCTION

The Product Data Relay is a close cousin of the Product Data Multiplexor. It exists to reduce the amount of TCP/IP traffic across the 128 Kbaud communications link between the radar site and the tower facility. In it's capacity as a relay, it reads data from a single input stream, and outputs it on a different stream to all clients requesting the data (fan-out). In it's capacity as a junction, it allows the substitution of a backup data stream in the case of a loss of data on the primary input stream. In the WSP design, the relay/junction module is run primarily on the tower (master) SD. During system development and testing, the relay/junction process has also been used to distribute data at remote monitoring sites. In general, the relay/junction may be used at any point in the WSP LAN where a fan-out operation is required.

3.12 WIND DATA SERVER

The wind data server process reads the raw LLWAS or ASOS data stream, reformats it slightly, and transmits the data to the radar site. Whether LLWAS or ASOS data is used as the source, the data appears as a simple serial stream, and is read using one of the ports on the WSP's serial communications server in the equipment room. The comm server essentially acts as a remote serial port on the WSP LAN, allowing any connected host computer to read the incoming wind data. The reading of the data is performed by the SD workstation in the tower. This allows for display of winds information on the tower SD even in the event of a failure of the communications link between the tower and the radar site.
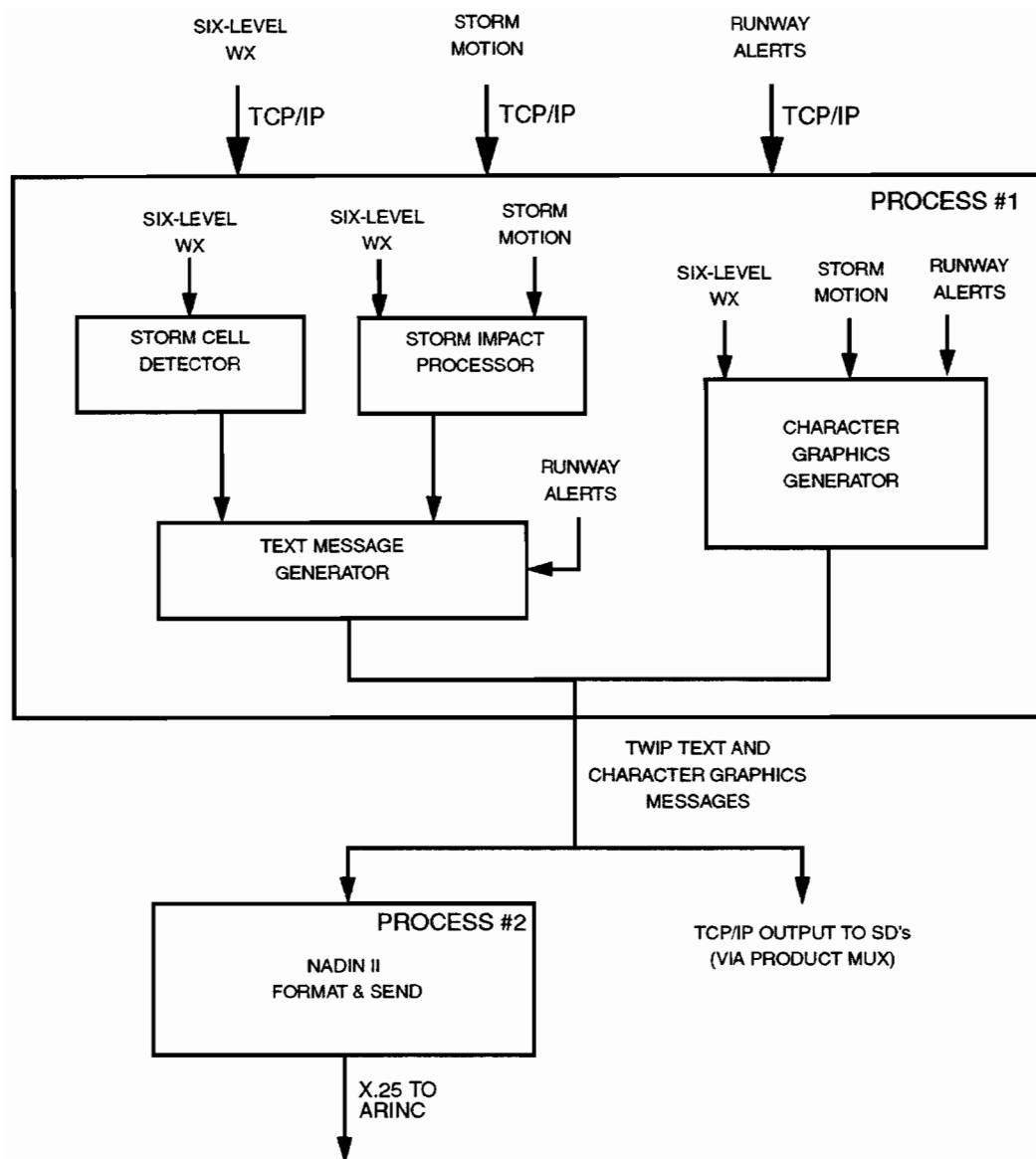
Figure 19.   WSP-TWIP Software Modules

3.13 SITUATION AND RIBBON DISPLAYS

The Situation Display (SD) is the final output device for the WSP products. The primary situation displays are built around a Sun Workstation platform with an auxiliary text-only 'ribbon' display terminal from DALE Electronics (the same terminal used for the TDWR system). A combination of graphical and textual weather information is provided to the controllers at the Tower/Tracon facility. Graphical information includes precipitation maps, microbursts, gust fronts, storm motion vectors, storm extrapolated positions, and center-field or (optionally) LLWAS winds. Textual information consists of concise messages describing the weather status for each runway/approach corridor particular runways.

The basic appearance of the SD graphical display is illustrated in Figure 20. Graphical representations of the six-level weather, microbursts, gust fronts, storm motion and extrapolated positions, are shown in the main panel. An overall warning status box is provided at the top right. A range selection area allows the user to set the range to one of the four possible values. A panel at the lower-right indicates the overall health of the system on an product-by-product basis. A menu bar at the top of the display provides access to a number of additional features, such as geographical overlays, the runway configuration editor, and a TWIP display window.
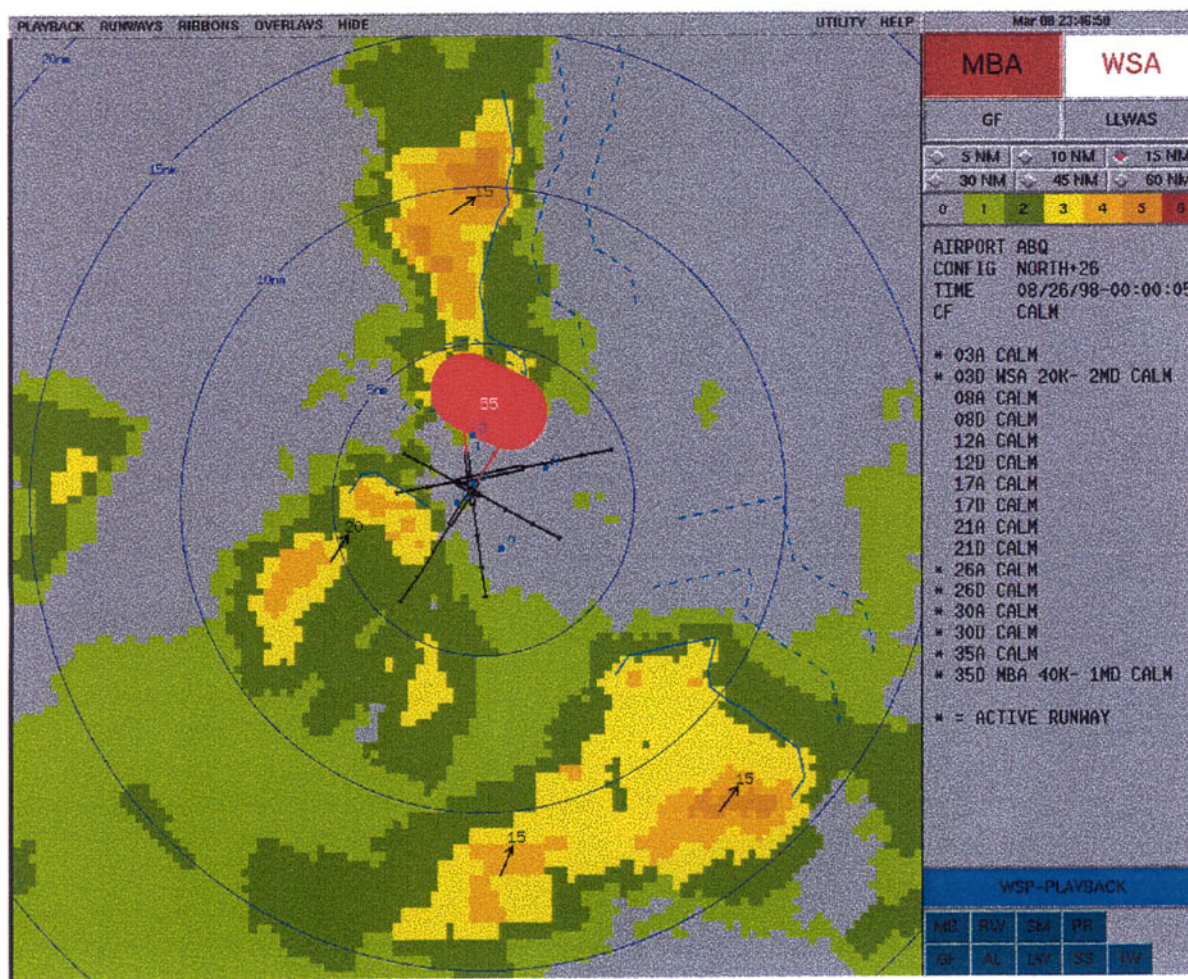
Figure 20.   Situation Display Graphics Screen

33

Closely related to the SD are the ribbon displays, separate flat panel display terminals used to present textual information regarding weather events in a highly-readable form (large letters). The large-format is required since the ribbon displays are the source of the information relayed to pilots on final approach, and the information must be easily accessible by all tower personnel. For convenience, multiple ribbon display terminals are often provided in a single tower cab. This is easily accomplished since multiple, daisy-chained ribbon displays can be driven by the Sun workstation via a single serial connection.

The appearance of the ribbon display is illustrated in Figure 21. Note that the grille on the lower left houses the audible alarm that is set off when a microburst or gust front first appears.
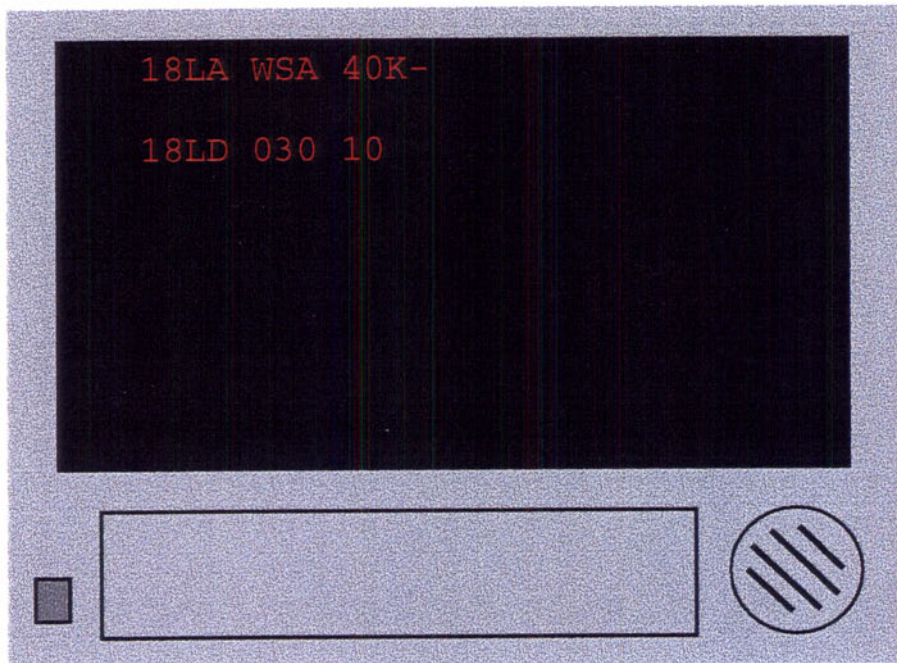
Figure 21.   Situation Display Ribbon Display Terminal

A block diagram of the core SD software is shown in Figure 22. The SD software is partitioned into three separate tasks, the main display 'server' task, the user interface task, and the runway configuration task. The display server task is responsible for reading the input product stream, creating the graphical and textual respresention of the data, and transmitting the results to the user interface task using a combination of shared memory and built-in Tcl/Tk interprocess communication protocols. The user interface task, implemented using Tcl/Tk in conjunction with the **imgsh** extension, actually displays the graphics information and provides the user interface elements necessay to control basic display parameters and airport runway configuration. The runway configuration client task handles the aynchronous communication between the SD and the runway configuration server process residing in the RDP at the radar site.
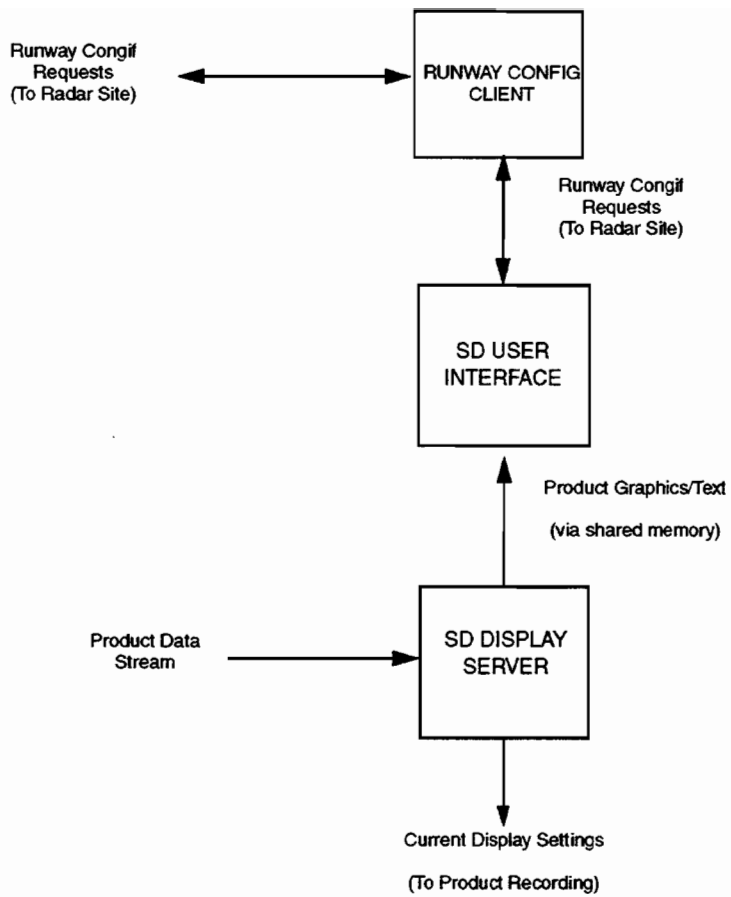
Figure 22.   Situation Display Block Diagram

4. DATA RECORDING AND PLAYBACK

The WSP is capable of recording and playing back three types of data, time-series data, base data, and product data. Up to two hours of time-series data (~20 Gb) can be recorded using a high-speed 8mm tape drive (optional equipment). Time-series recordings are normally made only at the operators discretion. Base data are continuously recorded to disk at all WSP sites. Sufficient disk storage (18 GB) is provided to allow the most recent 20-hours of data to be stored. The archived base data can be transferred at any time to an 8mm tape drive that is present at all sites. Product data is continuously recorded by each SD in the Tower/TRACON. Sufficient disk storage is provided to allow a 15-day history to be maintained (~300 Mb max). The SD in the TRACON is equipped with an 8mm tape drive to for permanent archival of the 15-day product data history on command.

4.1 TIME-SERIES RECORDING SUBSYSTEM

The time-series recording subsystem must be capable of recording sufficient data to accurately recreate the original base data and windshear alerts when it is played back through the signal processor. This means that *all* data within the 240-gate range of the gust-front and microburst algorithms must be recorded, as well as at least *some* of the data in the less critical region beyond 240 range gates. Recording only a subset (one scan every two minutes) of the long range data is somewhat of a compromise, since the scan-to-scan averaging of the lags data will not perform identically to real-time when data is played back, but it is felt that the data will be sufficient to reproduce the original long-range six-level weather and storm motion products with sufficient accuracy. The benefit to this approach is a reduction of the average data rate to the point where it falls below the 3MB/sec. max transfer rate of the new high-performance 8mm tape drives.

Two factors complicate the task of recording the time-series data - the alternating beam mode, and the 105 degree precession of the alternating beam switch point used to 'spread out' the effects of the corrupted data at the beam switch location. The approach taken here is to group a full low-beam cycle (360 + 105 degrees) and a full high-beam cycle (also 360+105 degrees) into a single 'volume scan', allowing each volume scan to stand on it's own. Given that the antenna scan rate is typically 78.3 degrees/sec., each volume scan represents approximately 11.9 seconds worth of data (~5 scans/min.)

Data rate for limited range volume scans (continuous):

10 header words + (240 gates @ 1-gate spacing) * 2 channels = 490 words/pulse = 1960 bytes/pulse

1960 bytes/pulse * 1200 Avg PRF = 2.35 MBytes/sec.

Data rate for full range scans (once/min.):

10 header words + ((240 gates @ 1-gate spacing) + (180 gates @ 4-gate spacing)) * 2 channels =

850 words/pulse = 3400 bytes/pulse

3400 bytes/pulse * 1200 Avg PRF = 4.1 MBytes/sec.

Average data rate:

(4 * 2.35 + 1*4.1)/5 scans = 2.7 MBytes/sec.

This data rate is within the 3MB/sec. max transfer rate of the high-speed 8mm tape drive (Exabyte

'Mammoth' 8mm drive). Given a tape capacity of 20 GB, this data rate translates to a recording duration of roughly 2 hours/tape.

The offline analysis of time-series data requires some additional system information, namely the VSP settings in effect at the time of recording, as well as the STC and clutter maps. For convenience, the recording software stores this information at the start of each time-series tape. See Appendix C for a detailed description of the time-series data format.

A block diagram of the time-series Recording/Playback subsystem is shown in Figure 23. During recording, time-series data is placed in a large buffer (~16MB) residing on an external VME memory board. The SPARC CPU responsible for recording retrieves the data from the buffer into a second, on-board buffer, bundles the data into volume scans, and writes the volume scans to the disk or tape device. During playback, the process is essentially reversed. Note that the PowerPC distributor node is responsible for 'throttling' the data in playback mode since it has an accurate on-board high-resolution timer. This reduces the SPARC processors job to simply reading the data from disk/tape and writing it to the PowerPC node as fast as the PowerPC accepts it.

Figure 23. Time-series Recording/Playback Block Diagram

38
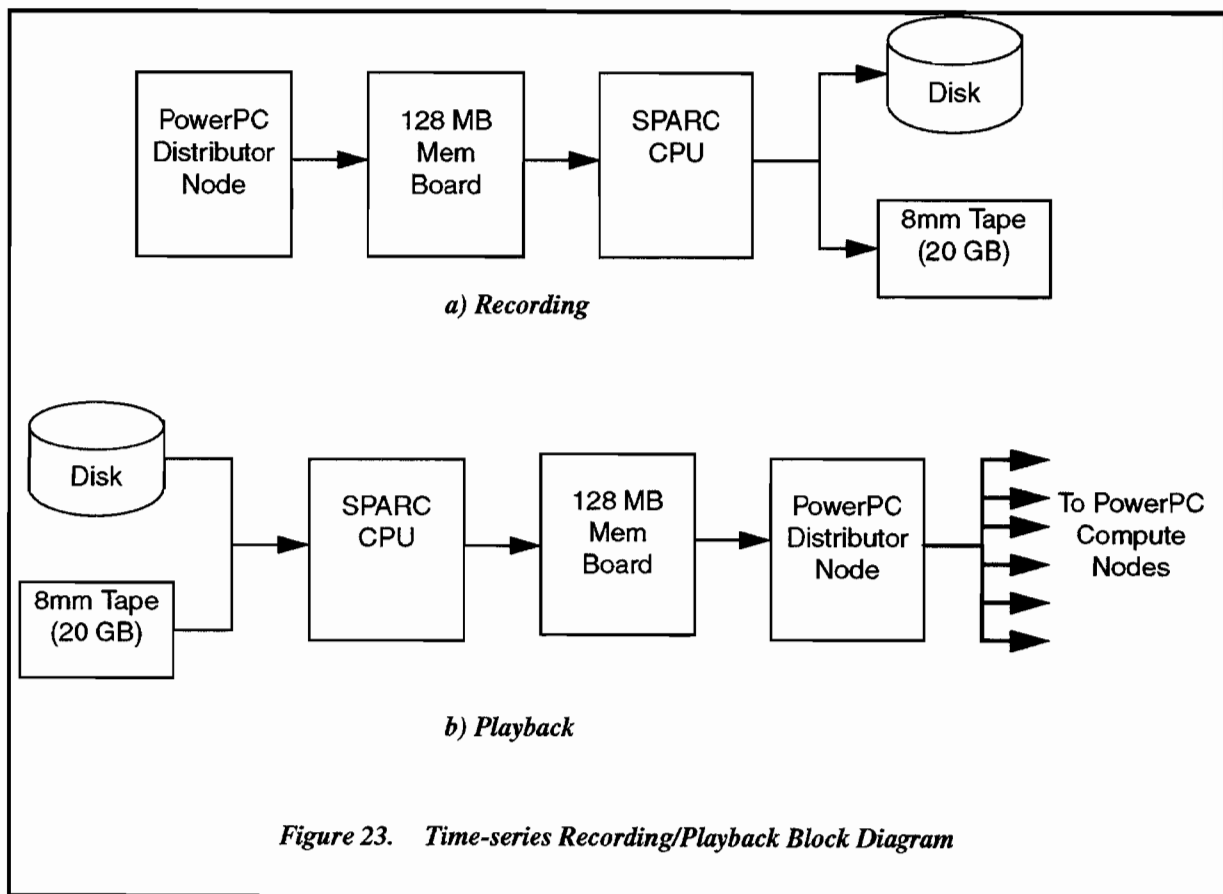
4.2 BASE DATA RECORDING SUBSYSTEM

The primary goals of base data recording are to enable accurate recreations of weather events and allow for algorithm evaluation and refinement. The volume of data recorded can be reduced through knowledge of the algorithm requirements. The microburst algorithm, for example, requires continuous recording of seven base-data products out to a range of 160 gates, while the storm motion algorithm requires full-range data only every 6 scans (~28 sec.).

For convenience, base data files typically contain a number of scans, referred to in the following discussion as a 'scan group'. Each scan group containing 12 physical antenna scans, corresponding to approximately one minutes worth of data. The first scan in each group contains all the products out to full range. All other scans contain only the data required by the microburst algorithm, with the exception of the sixth scan, which contains DUAL_Z and FLAGS data out to the full range of the radar for the benefit of the six-level weather generation algorithm. The nominal contents of each of the scans in the group is shown in table form in Table 3. Note that the format is flexible enough to handle many other product/range combinations. Additional details regarding the format are provided in Appendix C.

TABLE 3

Base Data Scan Group Contents

Scan(s)	Products/Range	Destination Algorithm(s)
1	LO Z, LO V, HI Z, HI V, DUAL Z, FLAGS 240 gates @ 1-gate spacing + 180 gates @ 4-gate spacing DUAL V 240 gates @ 1-gate spacing	All Algorithms
2-5	DUAL Z, DUAL V, FLAGS 160 gates @ 1-gate spacing	Microburst Algorithm
6	DUAL Z, FLAGS 240 gates @ 1-gate spacing + 180 gates @ 4-gate spacing DUAL V 160 gates @ 1-gate spacing	Microburst Algorithm, Six-Level Wx Algorithm
7-12	Same as scans 2-5	Microburst Algorithm

A list of recorded base data products is shown in TABLE 4. Reflectivity values are stored using a single byte per gate, while all other data types are stored using two bytes/gate.

TABLE 4

Base Data Products

Product Name	Product Code	Bytes/ Gate	Description
LO_Z	0	1	Low Beam Reflectivity
LO_V	1	2	Low Beam Velocity
HI_Z	2	1	High Beam Reflectivity
HI_V	3	2	High Beam Velocity
DUAL_Z	4	1	Dual Beam Reflectivity
DUAL_V	5	2	Dual Beam Velocity
FLAGS	8	2	Data Quality Flags

Using information from the two tables, the average data rate can be computed as follows:

Scan 1   (11 bytes/gate * 240 gates + 9 bytes/gate*180 gates)* 256 radials/scan = 1.1 Mb/scan

Scans 2-5,7-12   5 bytes/gate * 160 gates * 256 radials/scan = 0.2 Mb/scan

Scan 6   (5 bytes/gate * 160 gates + 3 bytes/gate*260 gates)* 256 radials/scan = 0.4 Mb/scan

Avg Rate = (1.1 +.2*10 +.4) Mb / 12 scans * (1 scan/4.6 sec.) = 63 Kbytes/sec.

The average data rate requirement of 64 KBytes/sec. is easily met using a Exabyte 8505XL 8mm tape drive unit (capable of transfer rates of up to 500 KByte/sec., uncompressed). The nominal uncompressed capacity of these drives is 7.0 GB/tape. This translates to a tape capacity exceeding 24 hours, a convenient size for base data archives. As stated earlier, base data recordings at commissioned sites will normally be maintained on disk, and only dumped to tape under operator control.

The base data recording/playback subsystem is illustrated in Figure 24. Data is transmitted from the PowerPC base data output node to a 4 MB buffer residing on the VME memory board. The SPARC recording CPU copies the data into local memory, bundles it up into scan groups, and outputs the scan groups to a 'staging' disk, from where they can be copied to tape by a background process. The use of the staging disk serves to isolate the recording process from long (30 second) 8mm device startup times, as well as provide the mechanism for maintaining the base data history at commissioned sites.

a) Recording

b) Playback

Figure 24.   Base Data Recording/Playback Subsystem
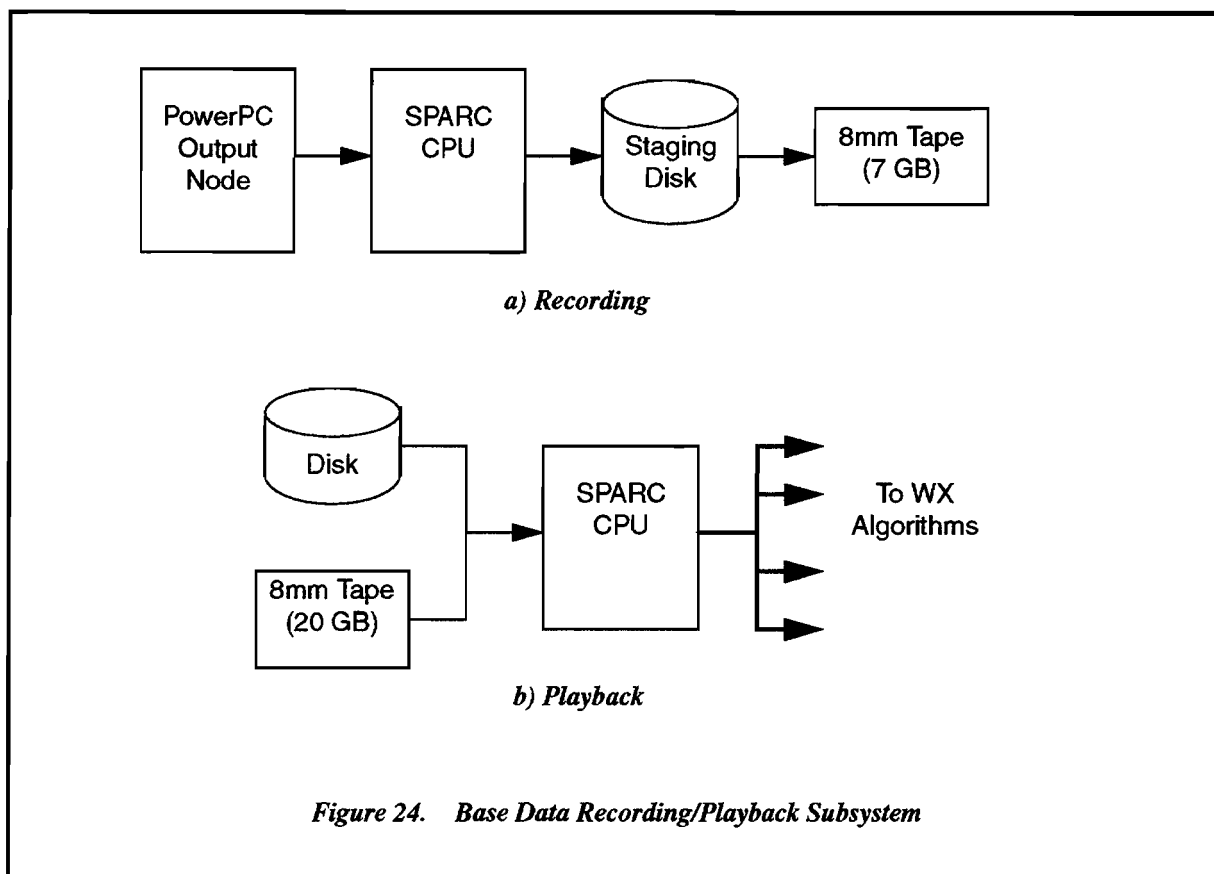
Playback of base data is shown with a very general "To WX Algorithms" final output. This is due to the fact that playback capability includes, but is not strictly limited to, playbacks in the real-time system. A single fast UltraSPARC or Pentium Pro machine, for example, could be used to read the data and run the algorithms offline at near real-time rates. To support all types of playbacks, playback code is set up to have configurable outputs (System V shared memory, VME shared memory, TCP/IP).

4.3 PRODUCT RECORDING SUBSYSTEM

In order to accurately reconstruct events without resorting to a base data playback (VME Processor required) the WSP is required to maintain a separate archive of all product data (precip, microburst, gust front, etc...) displayed on the situation displays. Each SD records all data arriving the product input stream on it's local disk, maintaining a continuous 15-day history (one file is recorded per day). In addition to the product information, which is identical for all SD's, SD-specific information regarding the display settings in use at any given time is recorded. This allows for an exact reconstruction of an event with regard to how it appeared on a particular SD. The SD machine in the TRACON is equipped with a 8mm tape drive to allow product archives to be transferred to tape. The 8mm tape drive may be attached to other SD's if the display-specific information on a machine other than the TRACON SD is deemed important enough to save to tape. A block diagram illustrating the default recording configuration is shown in Figure 25.

41

Figure 25. *Product Data Recording Block Diagram*
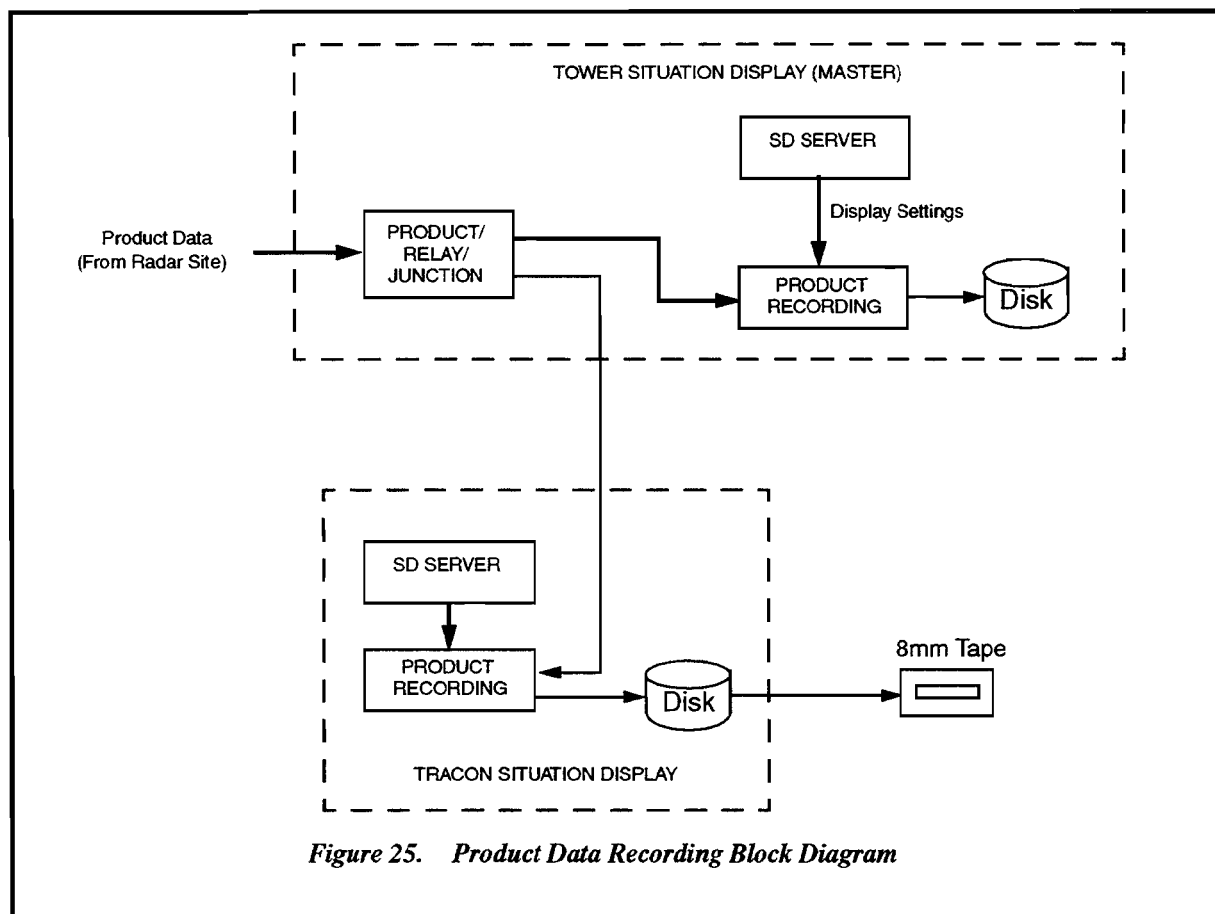
The data rate for an active weather period can range from 10-20 Mb/day. The maximum expected disk/tape space requirement for storage of 15 days worth of data is therefore 20 Mb/day * 15 days = ~300 Mb. This is a small amount of data given the capacities of current disk and tape units, and in fact, the disk space available on the SD machines significantly exceeds this requirement.

A variety of options will be available for reviewing the product data. Basically, any machine that can be configured as a situation display will have the capability of functioning as a playback device. This includes the SD's at the tower and TRACON, as well as the MDT/SD in the tower equipment room.

Use of the tower display for playbacks may be restricted during some periods, since the WSP will essentially be offline when the tower (master) SD is in playback mode. Playback at remote locations such as Lincoln Laboratory or the FAA's Program Support Facility (PSF) will be possible via machines set up as dedicated WSP playback displays.

42

5. SUPPORT LIBRARIES

Many of the software modules in the WSP system share a common set of support libraries. Facilities include memory allocation, message logging, list and time handling functions, interprocess communication, and graphical display.

5.1 MEMORY ALLOCATION

Dynamic memory allocation using system-supplied versions of **malloc()/free() (or new/delete** in the case of C++) is often not a good idea for real-time applications that run for an indefinite period. The system-supplied routines are not very efficient, and constant allocation/deallocation can lead to memory fragmentation and eventual memory exhaustion. Out approach is to allocate memory up-front to the maximum extent that it is practical (within the *AlgInit()* procedure), and also use customized versions of **malloc()/free()** with different names (**Malloc()/Free()**) everywhere within the algorithm code. This allows the memory allocation implementation to be easily customized by the application. In the case of C++, the renaming is unnecessary, since C++ mechanisms exist to override the system's **new/delete** calls. The CSketch library overrides **new/delete** to efficiently implement the image processing functions, which may heavy use of dynamic memory allocation.

Our particular implementation of a custom memory allocator works under the assumption that memory is allocated/freed in blocks of similar sizes, and the allocation/free pattern is repeated throughout the lifetime of the application. The allocator simply maintains a private cache of memory blocks, sorted by block size.

The first request for a block in a particular size range is satisfied by a request to the system's **malloc()** call. A subsequent request for a block of the same size (assuming the previous block has been freed) will return a block from the memory allocator's cache, effectively bypassing the operating system. Fragmentation is prevented by using a separate cache for each block size supported by the allocator. By default, block sizes are spaced 32-bytes apart for requests up to 4096 bytes, following which they are spaced 1024 bytes apart. (This means that a request for 4097 bytes will actually use up 4096+1024 bytes of system memory, but to date the wasted memory hasn't proved significant for us).

5.2 MESSAGE LOGGING

During algorithm development, a variety of printf()-type messages find their way into the code. The messages can typically be split into four classes, informational, warning, error, and debug. In a real-time environment, a display may not be available for output of such messages, and it is desirable to provide a mechanism for enabling/disabling certain message classes as well as allowing for logging the messages to a file or socket connection. UNIX provides the **syslog** facility to accomplish this task. Because of our portability requirement and our desire to enable customization of the message handling mechanism, we have implemented a **syslog**-like facility for the WSP project. From an algorithm's point of view, the facility is quite simple. A C/C++ statement like:

```
Log( LOG_INFO, "Info msg%d\n", 1 );
```

outputs the message to the system log. Like the syslog facility, four classes of messages are supported, **LOG_INFO, LOG_WARN, LOG_ERR, and LOG_DBG.** In addition to the default debug level (1), three additional levels of debug support are supported (LOG_DBG2, LOG_DBG3, LOG_DBG4). Debug messages can be enabled on a per file or per function basis through the use of the logging configuration

43

file. Other features of our implementation include selective output to stdout/stderr and/or disk files, limited log file sizes, and automatic creation of backup log files across multiple program runs.

5.3 INTERPROCESS COMMUNICATION

The interprocess communications methods used within the WSP system can be broken down into two basic categories, shared memory-based communications, and UNIX socket-based communications. Shared memory is the primary communications method used for medium to high bandwidth connections within the WSP VME chassis, while the socket-based methods are used for lower bandwidth connections in the VME chassis as well as all connections between nodes on the WSP LAN.

5.3.1 Shared Memory Ring Buffers.

Within the VME chassis, there are three basic mechanisms for shared memory communication:

- .Shared memory communication between two PowerPC compute nodes via the RACEWay.

- Shared memory communication between an PowerPC and a SPARC, or between two SPARC CPU's, via the VME bus.

- Shared memory communication between multiple processes running on a SPARC CPU using System V shared memory mechanism.

Each of these mechanisms tend to have certain restrictions with regard to data alignment and initialization sequence. For example, to allow DMA transfers to be efficiently used between PowerPC nodes, the ring buffer record addresses must be known in advance, mandating the use of fixed size records. Use of the SPARC 5's DMA controller requires that records be aligned on 64-*byte* boundaries. To provide a consistent interface to all these mechanisms, a library of ring buffer routines has been written that isolates the higher-level code from the hardware-specific details. These routines implement a simple single-writer/single-reader shared memory ring buffer, where either the writer or reader is considered to be the 'master', and is responsible for the creation and initialization tasks (record size, number of records, etc...). A ring buffer 'slave' simply attaches to an existing ring buffer, and reads/writes data as the application demands. The structure of the ring buffer is shown in Figure 26. In this example, the write pointer leads the read pointer by two frames, indicating that there are two frames waiting to be read.
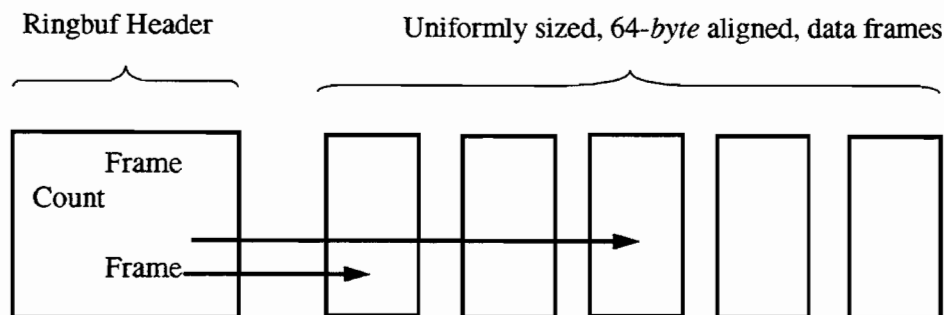
Figure 26. Ring Buffer Layout

5.3.2 Server-Client communications using TCP/IP and UDP.

The Server-Client interprocessor communications package allows a server process to broadcast data to multiple clients over a Local Area Network using either the TCP/IP or UDP protocols. TCP/IP is used exclusively in cases where reliability of the communications pathway is of paramount importance, as is the case when transmitting data to the Situation Displays at the Tower and TRACON facility. The drawback of TCP/IP, a connection-based protocol, is that it requires a separate data transmission for each connected client, utilizing more and more of the LAN's bandwidth as each client is added. Therefore, to efficiently support multiple non-critical remote situation displays, a UDP-based broadcast protocol is also supported. The two protocols are implemented using a layered approach, illustrated in Figure 27. The choice of protocol is typically specified at run-time via a configuration file.
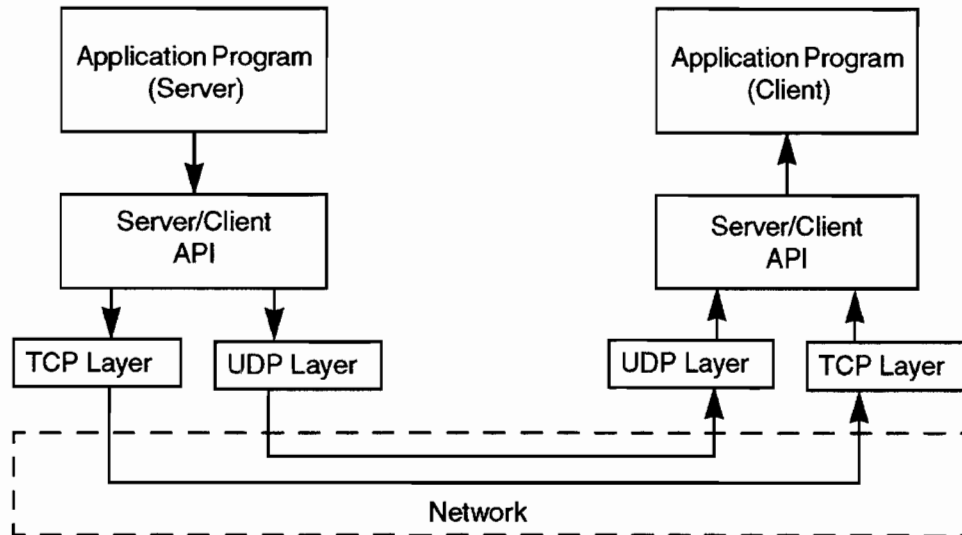
Figure 27. Server-Client Communications Layers

In the WSP, applications programs functioning as data 'servers' typically do not want to block when sending data. Likewise, applications functioning as clients may not wish (or be able) to constantly monitor a port for incoming data. To satisfy these requirements while at the same time avoiding the loss of data, some form of buffering is necessary. This buffering is handled somewhat differently for the TCP/IP and UDP protocols in order to handle some basic differences between the two communications methods.

5.3.2.1 TCP Implementation

When a TCP/IP-based server is connected to multiple clients, each client may be connected via network paths of varying speeds. A client on the same physical machine, for example, utilizes the 'loopback' network interface (very fast), while another client may be connected via a 56 or even 19.2 KBaud dial-up PPP network connection. Simply transmitting data to each client in a serial fashion in such a configuration would result in an overall latency for each 'send' operation equal to the sum of all the transmission delays for each device. Instead a separate child process is created for each connection to provide true concurrent serving of the data, while at the same time providing a buffer between the server and the client to allow the server application to quickly resume its normal processing. The server simply transmits the data to each

45

child process via a UNIX pipe (minimal latency) and is free to continue. This design is depicted in Figure 28.
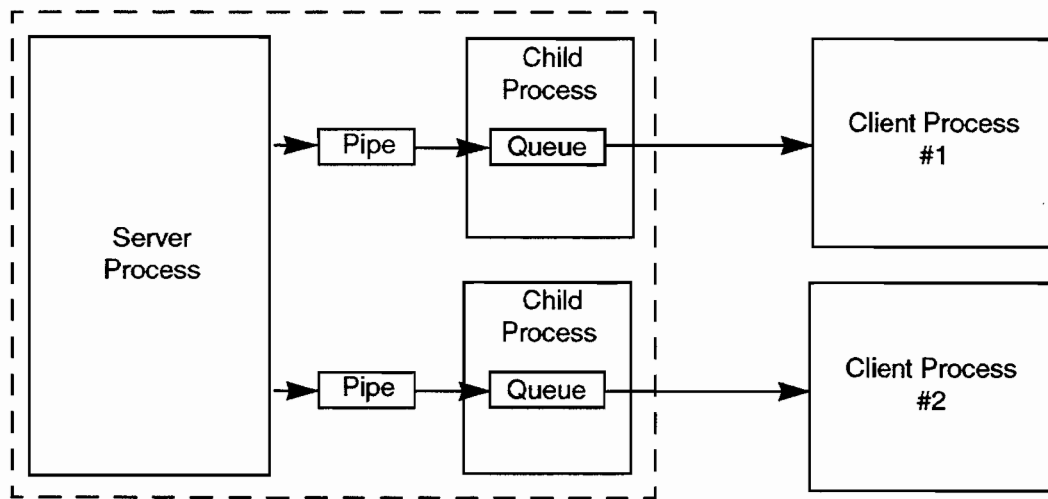
Figure 28. Server-Client TCP Implementation

5.3.2.2 UDP Implementation

The UDP-based server implementation utilizes the internet broadcast mechanism to allow multiple clients to 'listen' to a single data transmission on a single network. In this case, the assumption is made that the actual latency due to the transmission of each UDP packet is minimal, and no buffering is performed on the server side.   Instead, buffering is performed on the client side, preventing the loss of data if a client is busy processing data when new data arrives. Once again, a separate child process is used to implement the buffering mechanism, although shared memory is used for child-parent communications in place of a UNIX pipe for efficiency reasons (the code was inherited from an application requiring high bandwidth). The UDP design is illustrated in Figure 29.
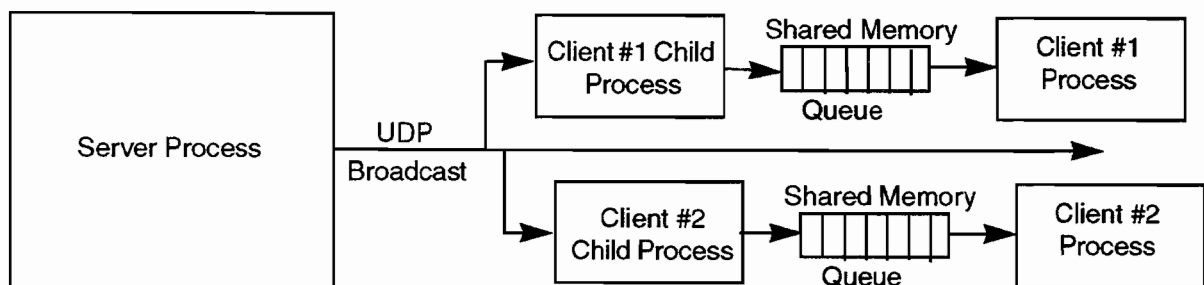
Figure 29. Server-Client UDP Implementation

This implementation currently does not support retransmission of missing packets. Given the projected ethernet loading of the dedicated WSP LAN, it is not anticipated that a significant number of packets will be lost.

5.4 CSKETCH IMAGE PROCESSING LIBRARY

The CSketch library is a port to C++ of an earlier LISP-based library developed at Lincoln Laboratory to solve a variety of image processing problems. The library includes support for common image processing functions such as erosion and dilation, as well as more sophisticated techniques which blend traditional kernel-based convolution with fuzzy logic operations. The C++ version of the library is centered around a single multi-dimensional array class which shields application code from many of the low-level coding details. The overloading of common operators, such as '+' and '*', aids in the construction of terse, highly-readable application code. The use of C++ 'templates' allows for the support of multiple array types (short, int, float, etc...) from the same code base, simplifying the code maintenance task.

The general appearance of code written using the library is shown in the following C++ code fragment. In this example, two arrays are created, filled with a constant value, added together, and the sum printed.

The program output follows the example.

```
//
// File: example1.C
//
// Simple sample program to add two arrays together
// and print result.
//

#include <stdio.h>
#include <iostream.h>
#include <skarray.h>

void main( int argc, char **argv )
{
  SKArray<short> a(3, 3) ;
  SKArray<short> b(3, 3) ;
  SKArray<short> c;

  a.RampFill(1);
  cout << "A:" << endl;
  cout << a << endl;

  b.RampFill(10);
  cout << "B:" << endl;
  cout << b << endl;

  c = a + b ;
  cout << "A+B:" << endl;
  cout << c << endl;
}
```

```
A:
   7   8   9
   4   5   6
   1   2   3


B:
  16  17  18
  13  14  15
  10  11  12


A+B:
  23  25  27
  17  19  21
  11  13  15
```

5.4.1 Functional Template Correlation

Functional Template Correlation (FTC) is worthy of special mention, since it is the pattern recognition technique used most frequently by the weather detection algorithms. It is a blend of kernel-based convolution and fuzzy logic.

Consider the case of the detection of a thin-line 'feature' in the reflectivity data (oftentimes an indicator of a gust-front). A simplistic detection algorithm could be stated as 'find regions of reflectivity where there is a corresponding *lack* of reflectivity data on either side. A possible FTC implementation of the algorithm is shown in Figure 30. A 3x7 kernel is convolved with the raw reflectivity image data, and the scoring function specified in the kernel is used to map the underlying reflectivity values to a pattern match 'score'. The scores for the entire kernel are added and averaged to produce a final score for each point in the input image. In this example, regions of constant high reflectivity will return a low score, as the kernel edge pixels utilizing scoring function 0 will tend to cancel out the positive scores returned by the center pixels. A similar result will occur given a thin-line of *low* reflectivity. The only situation returning a high overall score is where the scores from the two regions of the kernel add constructively, a reflectivity thin-line.

Note that this process will only detect a thin-line at a particular orientation (north-south in this case). In practice, the process is repeated multiple times using rotated versions of the same kernel to detect the pattern at any orientation. In many cases, the number of repeated passes can be reduced due to processing symmetry - a kernel rotated 180 degrees is often identical to it's 0 degree counterpart. Nevertheless, the need for multiple passes contributes significantly to the overall processing requirement.

Scoring function 0: Return positive value for dBZ < 20

Thin-line feature detector kernel

0	1	0
0	1	0
0	1	0
0	1	0
0	1	0
0	1	0
0	1	0

+

-
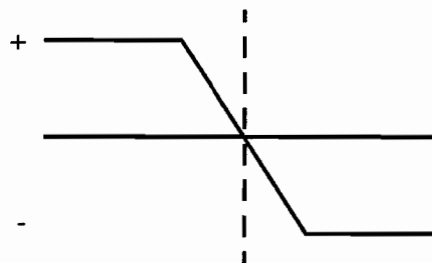
Scoring function 1: Return positive value for dBZ > 20

+

-

20 dBZ
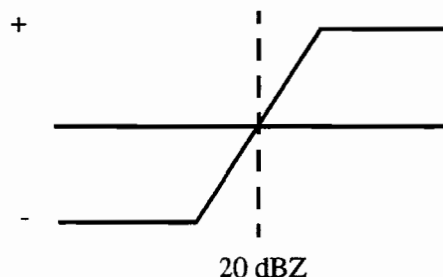
Figure 30. Functional Template Correlation Example

5.5 TCL/TK IMAGE DISPLAY EXTENSION

The Tcl/Tk package provides facilities for embedding a simple command language into a C/C++ program, as well as a method for building Graphical User Interface (GUI) front ends to the WSP algorithm and control software. A number of the software modules require efficient, real-time display of 2-D images with overlaid graphics and text. While the toolkit does provide a canvas 'widget' for these situations, it was found to be lacking in a number of areas. Built-in support for panning and zooming, flicker-free image updates, and support for transmittal of drawing commands via shared memory were a few of the features not well supported by the 'stock' Tcl/Tk software.

One of the strengths of Tcl/Tk is it's extensibility. There are established methods by which user-defined widgets (written in C) can be cleanly integrated into the package, resulting in a modified software package that has all the features of the original, *plus* some additional capabilities. In the case of the WSP software, an extension of the Tcl/Tk WIndowing SHell (**wish**) called **imgsh** has been written to provide for our image display needs. **Imgsh** is used in the implementation of the base data display, the situation display, the (auxiliary) algorithm analysis displays, and a number of other miscellaneous utilities such as the clutter and STC map display programs.

There are two chief modes of usage of the **imgsh** software. For simple programs, the entire application is often written entirely in Tcl and executed using the imgsh interpreter. No C/C++ code is typically required, except perhaps to reformat image data prior to display. A more complicated usage involves the setting up of a separate display 'daemon' process that listens for arriving graphics commands via UNIX

49

shared memory. This is the method commonly used by all real-time display software, as it is desirable to decouple the data processing function from the display function in order to keep the display responsive to user interaction at all times (In other words, a lengthy data processing computation will not cause a display tool to appear to 'hang' for periods of time while the computation completes).

5.5.1 Simple Imgsh Usage

Imgsh incorporates several new commands/widgets to the standard **wish** interpreter to allow 2-D images to be conveniently displayed. Image data is associated with an **llimage** 'object', and displayed in one or more **llimagewin** widgets. An additional new widget, **llcolorscale**, is provided to produce color-scales for the various image windows (the 'll' prefix, denoting "Lincoln Laboratory", is used to differentiate the new commands from existing Tcl command names). Display colors are controlled via an ASCII colormap file, which associates certain values of the image data with specified colors.

The quickest way to understand the basic capabilities of **imgsh** is via example. A simple standalone display script utilizing the basic features is shown below. Comments are provided to make the script self-explanatory. The ASCII colormap used by the script is also shown. The screen output produced by the script is shown in Figure 31. As demonstrated by this example, the **imgsh** extension provides a reasonably concise method of implementing useful graphical displays. Note that this sample script is very similar to those used by the clutter and STC map display utilities, though those programs display data in multiple windows simultaneously.

Example **imgsh** script

```
#
# Initialize an object to hold an 64x64, 1-Km bin, 16-bit signed image.  Set it up so that (0,0) in world
#  space corresponds to the center of the image (32,32)
llimage testimg -nbins 64 64 -binsize 1.0 1.0 -databits 16 -datatype signed -refloc 32 32 0.0 0.0 \
    -dataclass INTEREST

# load the image data from a disk file into the image object.  Data file contains test image with
# negative values at bottom of image, gradually ramping up to positive values at the top of the
# image.
testimg load testimg.dat

# Create a 240x240 window for displaying the image, and issue the command to actually display the
# image in the window
llimagewin .win -geometry 240x240 -colormap testMap
.win display testimg

# Add range rings starting at 0,0 with a spacing of 10 Km
.win rings -center 0.0 0.0 -spacing 10.0

# Add a colorscale
llcolorscale .scale -geometry 240x40 -colormap testimgcolors -dataclass INTEREST

# Pack the image window and colorscale widgets into the main window, starting from the top down
pack .win .scale -side top
```

The ASCII colormap utilized by the example is shown below:

50

```
#
# Simple colormap definition for the dataclass named 'INTEREST'. Negative data values are
# shades of blue, while positive data values are shades of red. The syntax for each line is:
#
#    <StartValue> <Red> <Green> <Blue> <Bar Label Text>
#
# In this example, values from -256 to -191 have an RGB value of (0,0,255), while values from
# -192 to -129 have an RGB value of (0,0,224), etc...
#

INTEREST

-256  0  0  255   <192
-192  0  0  224   <128
-128  0  0  192   <64
 -64 96 96  160   <0
   0 160 96  96   >0
  64 192  0   0   >64
 128 224  0   0   >128
 192 255  0   0   >192
```
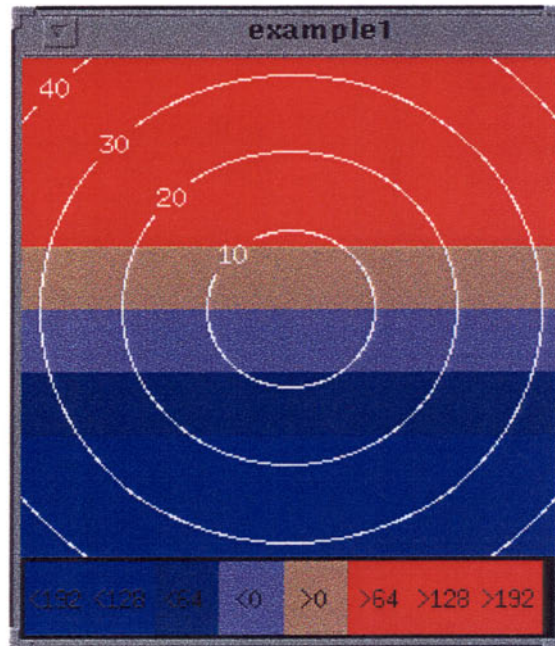

Figure 31.   Imgsh Example Screen Output

5.5.2   Using Imgsh as a Display Daemon

As mentioned above, a common usage of **imgsh** is to set up a **imgsh**-based display daemon and transmit images and graphics to the daemon process from a C or C++ application. A set of C functions is provided for this purpose, allowing for sending of image data and graphics commands via a shared mem-

ory 'channel'. Image data is transmitted more or less directly, while graphics commands are buffered for increased efficiency. The C interface is flexible enough to allow multiple processes to interact with a single display daemon, a single process to interact with multiple daemons, or finally, multiple processes to interact with multiple daemons. An attempt is made to conserve resources when possible. Unless otherwise specified, each application uses a single Tcl interpreter for communication with all display daemon processes. In addition, all image/graphics data sent from a given application to a given server is typically multiplexed onto a single shared memory 'channel' to conserve shared memory, a somewhat scarce UNIX resource.

One possible configuration, two application processes communicating with a single display daemon, is depicted in Figure 32. The AMDA analysis display is set up using this model, with a dedicated display daemon process receiving interest image data from the interest generation module and graphical microburst data (shear segments and microburst shapes) from the alarm generation module. Note that the display daemon most often executes a Tcl startup script that sets up all the basic image windows and control panels, making many display modifications a simple matter of editing the startup script (no recompile required).

Figure 32. Application/Imgsh Display Daemon Communication
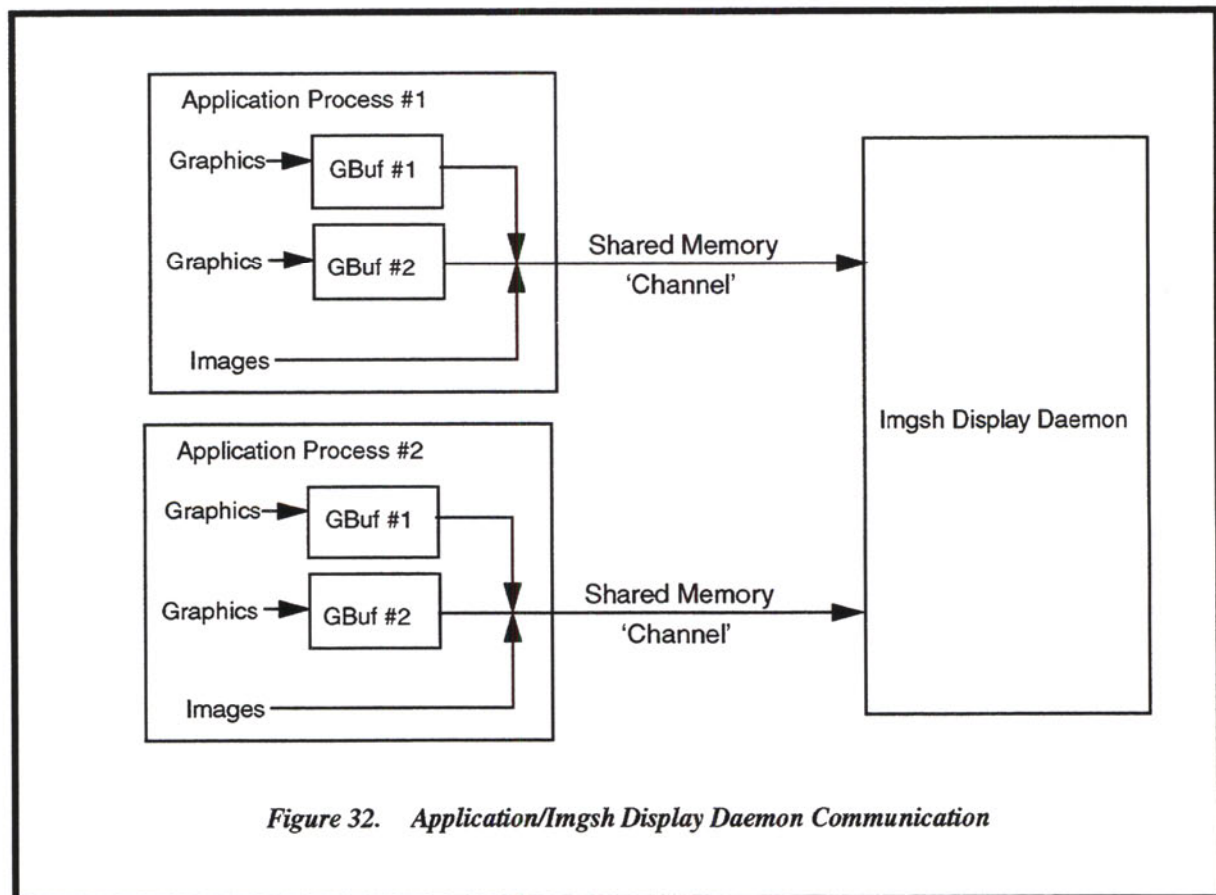
5.6 WEATHER OBJECT LIBRARY

The Weather Object Library (WxObj) contains a set of C++ classes representing the various WSP output products. Each data product is encapsulated in one or more classes, each with it's own unique ID. The primary uses for the library are product data I/O and graphical display. Product data network I/O is

52

implemented on top of the server-client package described earlier. Product data display is implemented on top of the Tcl/Tk imgsh extension described in?. In most cases, the WxObj layer is the only API used by the weather detection algorithms, effectively isolating them from the lower-level programming details.

The WxObj class hierarchy is illustrated in Figure 33. All object inherit from a common base class, **WxBase**, which contains a type (object Id) field and a set of 'virtual' *pack()/unpack()* functions which must be implemented within each derived class in order to perform file/network I/O. Many objects also inherit from the WxDisplayable class, which contains a set of the most common display attributes (line color, thickness, etc...), as well as a virtual *display()* function that must be implemented by the derived class. The classes **WxBase** and **WxDisplayable** are commonly referred to as *abstract base classes*, since they essentially function as a specification for what the derived classes must contain; there are no instances of a **WxBase** or **WxDisplayable** being used on it's own.
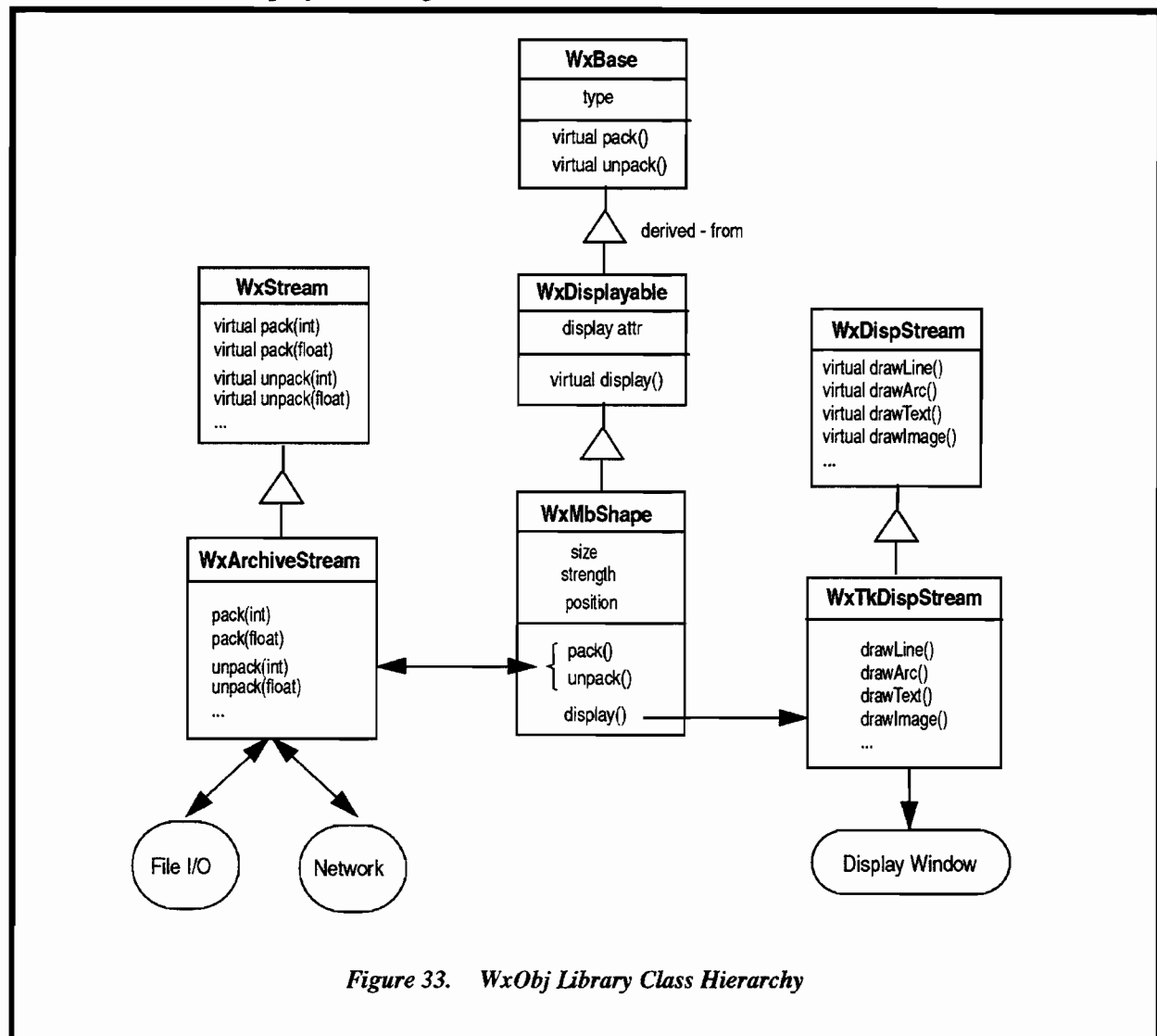
Figure 33. *WxObj Library Class Hierarchy*

Two additional abstract base classes, **WxStream** and **WxDispStream,** define the functionality required to perform file/network I/O and graphical display of product data. The WSP software currently provides a single specific implementation of each of these classes. The **WxArchiveStream** class is an implementation of an **WxStream** that packs and unpacks fundamental data types in an 'archive' format

compatible with older Lincoln Laboratory data files. The pack/unpack functions are machine-independent, and utilize big-endian byte ordering whether running on a big-endian (Sun) or little-endian (PC) platform. The **WxTkDispStream** class supplies the necessary graphics primitives to display weather products in a Tk (imgsh) display window.

Each particular weather object class contains object-specific weather event information, as well as high-level packing, unpacking and display routines. The **WxMbShape** class shown in the diagram contains microburst strength, size and location, *pack()/unpack()* routines and a *display()* function that draws the characteristic 'band-aid' shape to a display device. The high-level *pack()/unpack()* routines break the object down to it's fundamental data types, and utilize the *pack()/unpack()* routines defined in the **WxStream** class to actually transmit or receive the data. In a similar fashion, the high-level *display()* function utilizes the low-level routines in the **WxDispStream** class to perform it's job.

The abstraction of the WxStream and WxDispStream classes, while not strictly necessary for the WSP application, allows for a great deal of flexibility in terms of support for additional file formats and/or different graphical displays. For example, machine-independent I/O could also be implemented using Sun's eXternal Data Representation (XDR) protocol by defining and using an **WxXDRStream** in place of an **WxArchiveStream**. The weather object classes themselves would not require modification. Similarly, graphical output using the OpenGL graphics library could be transparently implemented by replacing **WxTkDispStream** with a **WxGLDispStream** class.

6. SOFTWARE DIRECTORY MAP AND BUILD TOOLS

6.1 SOFTWARE DIRECTORY MAP

The WSP software is organized into three top-level directories, **Gmake**, **share**, and **wsp2**. The **Gmake** directory contains the master set of build files. The **share** directory contains code that is non WSP-specific such as general-purpose communications and image processing code. The **wsp** directory contains all the WSP-specific code, including signal processing, windshear detection, and the WSP situation display.Brief descriptions of the directories and their primary subdirectories are provided below.

Gmake/	**Global make scripts used by all lower-level directories**
share/	**Non WSP-specific code. Shared by other Lincoln programs, including 'Outboard MIGFA' (AOS-250 already familiar with this code, so we are maintaining it in its own space)**
bin/	**Location for storing compiled version of binaries**
bindbg/	**Location for storing compiled version of binaries (debug variant)**
cft2/	**LL Common Format Tape (CFT) I/O library (Analysis tool support)**
csketch/	**C++ version of SKETCH image processing library**
imgsh/	**Image shell. Tcl/Tk image display extension**
inc/	**Include files for shared libraries**
lib/	**Location for storing compiled version of libraries**
libdbg/	**Location for storing compiled version of libraries (debug variant)**
llutil/	**General utilities (lists, time handling, memory management)**
misc/	**Miscellaneous function libraries (ring buffer, serial device)**
polygon/	**Geometric analysis library**
postplot/	**Plotting library (used by radar diagnostic utility programs)**
sclite/	**Network communication library**
wxobj/	**Weather 'object' library (object-based communication/display)**
wsp2/	**WSP-specific code.**
admin	**Software administration scripts**
alert/	**Alert generator**
algserv/	**Data stream configuration files**
amda/	**ASR-9 Microburst Detection Algorithm**
bddisp/	**Base data display**
bdrec/	**Base data recording code**
bdutil/	**Base data utility programs (viewer, tape inventory, etc..)**
bin/	**WSP binaries (executables) (non-debug versions)**
bindbg/	**WSP-specific binaries (debug version)**
clock/	**GPS clock reader/distribution code**
cluttutil/	**Clutter map utility programs (display)**
colormaps/	**Color maps for display programs**
control/	**System control code (see Northrop documentation for details)**
csal/	**C-coded equivalents for MC/OS vector library**
env/	**WSP user environment setup**
filtDesign/	**Clutter filter design software**
gfup/	**Gust Front Update Algorithm (works with MIGFA)**
inc/	**WSP system-wide include files**
iqdiag/	**Time-series data diagnostic tool**
iqdistrib/	**Time-series data distribution code**
iqrec/	**Time-series recording code and utilities**
lib/	**WSP libraries (non-debug versions)**
libdbg/	**WSP-specific libraries (debug version)**
logs/	**Directory for runtime logging**

maps/	Directory to hold clutter and STC maps (initially empty)
mdt/	Maintenance Display Terminal code (see Northrop documentation for details)
migfa/	WSP Machine-Intelligent Gust Front Algorithm
misc/	Miscellaneous WSP system-wide source files
mon/	System monitoring code (see Northrop documentation for details)
params/	General-purpose parameter directory
scripts/	startup/shutdown/general purpose scripts
sd/	Situation display
sigproc/	Signal processing code (runs on Mercury processors)
sim/	Offline simulator utilities
stcutil/	STC map utility programs (display, map generation)
stmmot/	Storm Motion Algorithm
twip/	Terminal Weather Information for Pilots Algorithm
users/	Default WSP user home directories
util/	Miscellaneous utility programs
wx/	Six-level weather code

The majority of the directories listed above contain a set of common subdirectories, organized to allow building the software for multiple-architectures (Solaris, Linux, PowerPC) and multiple variants (debug/non-debug). The use of a common directory structure for the entire WSP software tree allows for global software builds to be easily automated using a common set of high-level UNIX scripts and make-files. The basic directory structure is shown below. It is organized as a set of high-level *module* directories, each one containing a variable number of *sub-module* directories. The *module* category is generally reserved for larger-scale software entities, such as the microburst and gust front algorithms. Module-level directories contain their own **inc/**, **bin/**, and **lib/** subdirectories, used for posted versions of the executables and libraries. Sub-module level directories contain the source code and working versions of executables and libraries for logically related pieces of the higher-level module. Note that a number of the directories under **wsp2** are not classified as higher-level modules, but are instead considered sub-modules of the **wsp2** parent directory. This is generally true in the case of code that is shared among multiple applications, such as the utility functions in the **wsp2/misc/** and **wsp2/util/** directories.

```
<Module1>/
   inc/              Posted #includes for each architecture
   lib/              Posted libraries for each architecture
       solaris/ i586/ ppc/
   bin/              Posted binaries for each architecture
       solaris/ i586/ ppc/
   libdbg/           Posted libraries for each architecture  (debug variant)
   bindgb/           Posted binaries for each architecture  (debug variant)
   scripts/

   <SubModule1>/     Submodule containing source for libraries, binaries, and <optional> test code
       inc/              Include file source
       obj/              object files and working versions of libraries and binaries
           solaris/ i586/ ppc/
       objdbg/           object files and working versions of libraries and binaries (debug variant)
           solaris/ i586/ ppc/
       src/              source code and Makefile
       test/ (optional test directory)
           obj/              object files and working versions of test program binaries
               solaris/ i586/ ppc/
           src/              source code and Makefile

   <SubModule2>/     Submodule 2 (same structure as #1)
       .
```

<SubModuleN>/	Submodule N (same structure as #1)
<Module2>/	Major module 2 (same structure as #1)
.	
.	
<ModuleN>/	Major module N (same structure as #1)

6.2 SOFTWARE BUILD TOOLS

The primary build tool used for WSP software development is the UNIX 'make' utility, specifically, the version available from the GNU free software project, **gmake**. Additional C-Shell and Perl scripts are used to augment the basic capabilities of **gmake** and automate the build process for the entire software tree.

6.2.1 Common Makefile Description

Each **src** directory contains a UNIX makefile specifying all the targets for the directory. The makefile follows a specific format that allows for each target to have it's own logically separate set of compile time options.   A simple makefile with a single library target and a single binary target that depends on the library is shown below:

```
#
# Simple Makefile
#

#
# This line always included first to load global definitions used
# within these Makefiles.
#
include $(GMAKE_HOME)/MakeHeader

# Comment this out for verbose mode
QUIET = @

# List of all targets in this directory
TARGETS = libsample.a progsample

# Optional target to build everything in directory
all: $(TARGETS)

#
# Targets used by global build scripts.  lib and bin targets separated
# to allow libraries to be built prior to binaries
#
lib: libsclite.a
bin: progsample
# Files and compile options for 'libsample.a' library target
OBJ.libsample.a           := libfile1.o libfile2.o
INCPATH.libsample.a    := -I../inc
INCPOST.libsample.a    := ../inc/file1.h
CFLAGS.libsample.a      := -Wall
DEP.libsample.a            :=
LIB.libsample.a            :=
```

```
# Files and compile options for 'progsample' binary target
OBJ.progsample          := progfile1.o progfile2.o
INCPATH.progsample      := -I../inc
INCPOST.progsample      :=
CFLAGS.progsample       := -Wall
DEP.progsample          := ../$(OBJDIR)/$(TARG_ARCH)/libsample.a
LIB.progsample          := -L../$(OBJDIR)/$(TARG_ARCH) -lsample

# This line always included last to include Make rules and dependencies.
include $(GMAKE_HOME)/MakeTrailer
```

Generation of debug vs. non-debug variants is handled within the high-level **Make** scripts (see ./ Gmake/MakeHeader). When **gmake** is invoked with the -DDEBUG option, the scripts are set up in such a way that object files, libraries, and executables end up in separate directories than their non-debug counterparts (objdbg/libdbg/bindbg). Once the upper-level **Make** scripts have been configured for a particular compiler, no additional statements in the source directory makefile are normally required.

Some WSP software executables require the use of a cross-compiler. In other words, the architecture of the program library or executable does not match that of the current host (true for executables that are run on the Mercury PowerPC boards). In this case, the Makefile must contain explicit information specifying the target architecture. This is done using the ARCH.<TARGET> option of the make system. The example below shows the Makefile statements required to support a PowerPC variant of the 'progsample' target shown above. Note that if a cross-compiled version of a program is included in the same directory as a native-compiled version, they must be assigned different target names. Also note that explicitly defining an architecture for a target causes the $(TARG_ARCH) string to expand to the architecture specified at the time of the actual build.

```
OBJ.progsample_ppc      := progfile1.o progfile2.o
INCPATH.progsample_ppc := -I../inc
INCPOST.progsample_ppc:=
CFLAGS.progsample_ppc  := -Wall
DEP.progsample_ppc      := ../$(OBJDIR)/$(TARG_ARCH)/libsample.a
LIB.progsample_ppc      := -L../$(OBJDIR)/$(TARG_ARCH) -lsample
ARCH.progsample_ppc     := ppc
```

When creating a new Makefile, it's possible to make an error that causes the build to abort with non-obvious error messages. In such a case, it is useful to comment out the QUIET line in the Makefile. This will cause the make scripts to become much more verbose, hopefully making the error more obvious.

```
# Comment this out for verbose mode
#QUIET = @
```

More subtle problems may require additional knowledge regarding the GNU make utility. Refer to the GNU make user's manual for details.

6.2.2   High-Level Make Scripts

When working with multiple modules (or installing a new software release as described earlier), it is often desirable to perform automated builds of the entire software tree, or a significant subset of the tree. To accommodate this, the makefile system includes a script, ./Gmake/MakeAll, that implements a recursive build starting from the current directory and working its way downward through all subdirectories.

A **MakeAll** issued from the top-level project directory builds all files for the entire project. By default, the script does *not* build debug variants or cross-compiled targets. Options are used to control this behavior. The options to **MakeAll** are shown below:

Usage: MakeAll [-h] [-dbg] [-i860] [-ppc]

Options:
[-h]   Print usage message
[-dbg] Build debug versions of libraries and executables
[-i860] Build Mercury versions of libraries and executables (i860)
[-ppc] Build Mercury versions of libraries and executables (PowerPC)
[-mc]   Build Sparc-hosted binaries that interact with MC/OS

A global build is normally invoked in four stages. The first stage builds all the targets for the Solaris architecture that are completely independent of the Mercury OS. This stage may be run on any Solaris-based workstation having access to the GNU development tools. The second stage builds all targets for the Mercury PowerPC architecture, while the third stage builds the SPARC-hosted programs that interact with the Mercury CPU's. Note that the stages two and three require access to the Mercury development toolset. Lastly, the **BuildSetPerms** script is run to set the proper permissions for those executables that require super-user privileges, such as the timeServer process responsible for reading the GPS clock and updating the UNIX system time as necessary. The commands for the four build stages are shown below.

> MakeAll
> MakeAll -ppc
> MakeAll -mc
> BuildSetPerms (run as super-user)

It is often useful to redirect the output of the **MakeAll** runs to a file for later validation of a successful build. For example:

> MakeAll >& make.log'   redirects all normal and error output to the file 'make.log'
> grep -i error make.log'   checks the logfile for any compiler error messages.

6.2.3 Manual Build Process

When working on a single module and making only minor changes, it is sometimes convenient to manually specify the individual steps in the build. This is especially true when building only a single target in a directory containing many targets. Following the first global build, the following commands can be used to manually rebuild and post a specific target:

> gmake <target>
> gmake POST=1 <target>

Other usages are available for slightly more complicated operations. For convenience, a summary of WSP-specific gmake flags is available by issuing a gmake command with either no build target, or a build target of 'help', from within a module's ./src directory. Help text similar to the following will be output:

> gmake -help

Makefile Command Summary:

gmake help	Print this file
gmake clean	Clean out ../obj/<arch> directory
gmake DEPEND=1 lib bin	Build dependency file for all library & binary targets. File is of the form Make.dep.<arch>'. This command must be run at least once following an initial checkout of a software module.
gmake lib	Build all library targets in a given directory (non-debug version).
gmake bin	Build all library targets in a given directory (non-debug version).
gmake DEBUG=1 lib	Build all library targets in a given directory (debug version).
gmake DEBUG=1 bin	Build all library targets in a given directory (debug version).
gmake POST=1 lib	Post all library targets and associated include files to higher level directories (typically ../../lib/<arch> and ../../inc)
gmake POST=1 bin	Post all library targets to higher level binary directory (typically ../../bin/<arch>)
gmake POSTINC=1 lib	Post all include files for all library targets to a higher level directory (typically ../../inc)

Note that any single target may be specified in place of the 'lib' and 'bin' targets shown above (when only a single part of a module is being worked on).

APPENDIX A
CODING STANDARDS

In accordance with good programming practice, a standard set of file and function headers are used throughout the WSP GFE code. The exact format differs slightly depending on the language being used (C, C++, Tcl, csh), but the same basic information is present in all cases.

Examples of the standard file and function headers for a C file are shown below:

```
/*$(
 *============================================================================
 *
 *   (c) Copyright, 1997 Massachusetts Institute of Technology.
 *       This material may be reproduced by or for the
 *       U.S. Government pursuant to the copyright license
 *       under the clause at 252.227-7013 (Oct. 1988).
 *
 *       $RCSfile: secondTrip.c,v $ $Revision: 1.1.1.1 $  $Date: 1997/09/14 19:54:50 $
 *
 *============================================================================
 *
 *
 *   FILE:  secondTrip.c
 *
 *   DESCRIPTION:
 *
 *   Routines to compute 2-D spatial variance in velocity field.
 *   Due to the ASR-9's micro-stagger, the phase of second-trip echoes
 *   is essentially random.  The 2-D spatial variance of the velocity
 *   estimate can be used to distinguish between second-trip and normal
 *   weather echoes.
 *
 *
 *   FUNCTIONS:
 *
 *   SecondTripDetectInit()
 *   SecondTripDetectReset()
 *   SecondTripDetect()
 *
 *
 *   NOTES:
 *
 *   Second-trip detection logic may be modified in the future to
 *   use separately computed Lo/Hi PRF reflectivity estimates.
 *
 *============================================================================
 *$)
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
```

61

```
/*${
*==============================================================
*
*   FUNCTION:   SecondTripDetect()
*
*   DESCRIPTION:
*
*   Detect second-trip Wx using the spatial variance of the velocity field.
*   The variance at a given gate is calculated using the nearest 9
*   neighbors, using the formula:
*
*     variance =  SUM( V^2 )/9  - VMEAN^2
*
*   The output of this routine (variance) lags the input by one sector.
*   Following a reset, the first two calls to this routine just serve to
*   prime the data pipeline - the variance array is left untouched.
*
*
*   INPUTS:
*
*   v0          Velocity vector for currSector-1
*   v1          Velocity vector for currSector
*   v2          Velocity vector for currSector+1
*
*   OUTPUTS:
*
*   flags       Modified flags array with SECOND_TRIP_FLAG bit set
*               where second-trip was detected.
*
*   RETURNS:
*
*   Nothing
*
*==============================================================
*$}
*/

void SecondTripDetect( short *v0, short *v1, short *v2,
                short *flags, int nGates )
{
```

APPENDIX B
WSP MEMORY MAPS

B.1 VME Memory Map

A variety of boards occupy the WSP's VME chassis. Many of these boards communicate across the VME bus. Memory on each board visible from the VME bus must be set up at it's own address within the 4GB VME space. Table B-1 provides the address window of each board. Note that the FORCE CPU's are not currently accessed as VME slaves, so the addresses specified are simply those reserved for possible use. Each of the Mercury DSP boards has up to 4 CPU's, each with 16 Mb of RAM. All the memory is made visible to the VME bus, requiring a total window size of 64 Mb/board. The exception is the Mercury DSP board that contains the RIN-T parallel interface in place of 2 of the CPU's, which requires only a 32 Mb window.

TABLE B-1

VME Board Address Windows

Board Name	Base Address	Window Size
FORCE CPU #1 (System Control)	0x30000000	16 Mb
FORCE CPU #2 (Recording)	0x31000000	16 Mb
FORCE CPU #3 (Microburst Algorithm)	0x32000000	16 Mb
FORCE CPU #4 (Gustfront Algorithm)	0x33000000	16 Mb
Bulk Memory Board	0x20000000	128 Mb
Mercury DSP #1	0x10000000	64 Mb
Mercury DSP #2	0x14000000	64 Mb
Mercury DSP #3	0x18000000	64 Mb
Mercury DSP #4	0x1C000000	32 Mb
GSP Clock Board	0x2F000000	64 Kb

B.2 Bulk Memory Board Memory Map

The 128 Mb VME bulk memory board is used as a general purpose communication area, and is split up into numerous dedicated regions. The addresses of those regions are shown in Table B-2. Note that as currently specified, the base data output and time-series recording buffers are oversized. In future releases of the software, the buffers will be 'tightened up' to free up room for future expansion. Changes to the address map are effected by modifying the header file ~wsp2/share/inc/wsp_comm.h.

63

TABLE B-2

Bulk Memory Board Address Map

Base Address	Window Size	Description
0x20000000	512 Kb	System monitoring area
0x20080000	5.5 Mb	Base data output buffer (to base data display)
0x20600000	4 Mb	Base data output buffer (to six-level weather gen)
0x20A00000	6 Mb	Unused
0x21000000	8 Mb	Base data output buffer (to microburst algorithm)
0x21800000	8 Mb	Base data output buffer (to gustfront algorithm)
0x22000000	16 Mb	Base data output buffer (to recording process)
0x23000000	10 Mb	Ramdisk for clutter/STC maps
0x23A00000	4 Mb	Clutter map generation buffer
0x24000000	48 Mb	Time-series data recording buffer
0x27000000	16 Mb	Unused

APPENDIX C
FILE FORMATS

C.1 STC Map File Format

The WSP interface hardware includes support for a 2-dimensional mapped STC. The mapped STC function allows the STC attenuator to be 'opened up' in regions of relatively weak ground clutter. This improves the WSP receive chain sensitivity in those areas, benefitting the gust-front detection algorithm as well as the microburst detection algorithm in dry environments.

In keeping with the target channel STC capability, separate maps are maintained for channels A and B. Because of the different characteristics of ground clutter when the radar is operating in circular polarization, separate maps are maintained for linear/circular polarizations. Lastly, separate maps are used for the low and high beams, for a total of 8 maps. In addition to being stored in flash memory within the WSP radar interface, the maps are stored as disk files on the WSP RAID system. This allows the WSP processor to conveniently access the information required to correct for STC attenuation prior to clutter filtering and autocorrelation.

STC map files consist of a file header, followed by a variable number of maps, each map corresponding to a certain channel/polarization/beam combination.

C.1.1 STC File Header

The C structures for the file header are shown below. These structures, along with the structures for the maps themselves, can be found in the file **~wsp/inc/stc_defs.h**

```
#define STC_FILE_MAX_MAPS  8
#define STC_MAP_NSECTORS   256  /* Fixed for now. */

typedef struct
{
    int  bytes ;          /* Total bytes in file, including header */
    int  magicNum ;   /*  STC_FILE_MAGIC_NUM (5566) */
    int  version ;
    int  spare ;

 /*  Target Channel STC Info  (now obsolete - see note below) */
 STCInfoASR9  targChanSTC[N_CHANS][N_BEAMS];   /* N_CHANS = N_BEAMS = 2 */

 int nMaps;    /* Number of WSP Wx channel maps to follow */

} STCFileHdr ;

/* Structure for ASR9 STC settings  (now obsolete - see note below) */
typedef struct {
    int rangeOffset ;
    int maxAtten ;
    int refAtten ;
    int stcSlope[NUM_ZONES=8] ;   /* NUM_ZONES=8 */
} STCInfoASR9 ;
```

Note that the above structure is no longer used, but is included for backwards compatibility with older STC

map files. The ASR-9 STC target channel data is now available from the site's VSP database.

C.1.2 STC Map

Each map in the STC map file consists of a header, followed by the map data.   The C structure for the header is shown below. Note that although the file supports a varying number of range gates and sectors, the most common map will contain values for gates -8 to 248 and sectors 0-255.

```
typedef struct
{
    int   hdrBytes ;      /* Bytes in this header. */
    int   mapBytes ;      /* Total bytes in this map (including header) */
    int   version ;       /* In case the format changes. */
    int   chan;           /* 0 = A, 1 = B */
    int   polariz;        /* 0 = LIN, 1 = CIRC */
    int   beam ;          /* 0 = Low, 1 = High */
    int   bytesPerGate ;  /* 2 (sizeof(short)) */
    int   startGate ;     /* -8   (allows setting of STC for 8 gates prior to main bang */
    int   nGates ;        /* Total gates for this map. */
    int   gateStride ;    /* 1 = every gate, 2 = every other gate, etc.. */
    float gateSize ;      /* Size of each gate in meters. */
    int   startSect ;     /* (0-255) */
    int   nSects ;

} STCMapHdr ;
```

The map data immediately follows the header record in the file. Data are ordered by range gate, then by sector (azimuth), i.e.:

C.2 Clutter Map Format

The WSP adaptive clutter filtering scheme relies heavily on the use of site-specific clear-day clutter maps. The maps are created at WSP installation time, then periodically updated to reflect changes in seasonal and/or man-made ground clutter. At processor startup, the maps are loaded into the memory of the DSP boards, where the clutter filtering occurs.

Each map contains the 4 clutter filter bank outputs (all-pass, 20 dB notch, 40 db notch, 60 dB notch) measured on a clear day. Azimuth resolution of the map is 1.4 degrees, resulting in 256 azimuths, or 'sectors', per scan. The format supports separate maps for each beam/polarization/channel combination. In practice, the need for separate maps for A/B channels has not been apparent, and the WSP system is set up to use the same values for both channels. However, the format of the map file does support channel-dependent maps should there be a need for them in the future.

Each map file contains a file header, followed by a variable number of clutter maps. The number of maps and their type is provided in the file header. The nominal set of maps, in the default order in which they are stored, is shown below:

Linear Polariz / Low Beam / Channel A&B
Linear Polariz / High Beam / Channel A&B
Circular Polariz / Low Beam / Channel A&B

66

Circular Polariz / High Beam / Channel A&B

C.2.1   Clutter Map File Header

The C structure for the clutter map file header is shown below. The source file containing this structure is located in:

```
typedef struct
{
 int          type;     /* CLUTTMAP_ID  (39)*/
 int          hdrBytes; /* Header length in bytes */
 int          version;  /* Header version num */
 int          reserved;

 char         siteName[16];  /* "ABQ", "AUS" "BOS", etc... */
 WSPTime32        time;      /* Time of map data */

 int          nMaps;  /* Number of maps to follow */

} CluttFileHdr ;
```

C.2.2   Clutter Map Header

Each map has it's own header, followed by the map data. The C structure of the map header is shown below:

```
#define CLUTTMAP_MAX_RANGE_SEGS  4

typedef struct {
 int          hdrBytes;    /* Bytes in this header. */
 int          mapBytes;    /* Total bytes in map (including header) */
 int          beam;        /* (0 = Low, 1 = High) */
 int          chan;        /* (0 = A, 1 = B) */
 int          polariz;     /* (0 = LIN, 1 = CIRC) */
 int          spare;

 int          nScans;      /* # of scans used to produce this map. */
 int          nFilts;      /* Number of filters in map (4) */
 int          totalGates;  /* Total number of range gates in map */
 int          nRangeSegs;
 CluttMapRangeSeg  rangeSegs[CLUTTMAP_MAX_RANGE_SEGS];


} CluttMapHdr;

/*
 * Structure describing  multiple range segments within the map.   Typical maps have two range
 * segments, one with single gate spacing from gate 0 to gate 239, and one with 4-gate spacing from
 * gate 240 out to the full range of the radar.
```

67

```
     */
    typedef struct
    {
      int firstGate;
      int nGates;
      int gateSpacing;      /* 1 = every range gate, 2 = every other gate.... */
      int spare;

    } CluttMapRangeSeg;
```

C.2.3 Clutter Map Data

Clutter map data are stored immediately following the map header. Data are stored in order of filter/range-gate/sector. Clutter map values are stored as floating-point quantities, with each value representing the power residue for filter N for the particular range cell (linear units - not dB).

```
    Sector 0  Gate 0, Filter 0
    Sector 0  Gate 0, Filter 1
    Sector 0  Gate 0, Filter 2
    Sector 0  Gate 0, Filter 3


    Sector 0  Gate 1, Filter 0
    Sector 0  Gate 1, Filter 1
    Sector 0  Gate 1, Filter 2
    Sector 0  Gate 1, Filter 3
```

etc....

C.3 Time-Series Data Format

Time-Series data consists of the raw A/D samples from both the ASR-9 target channel receiver and the WSP's own weather receiver. Data is recorded using a high-speed 8mm tape recorder, capable of a sustained recording rate of 3 Mb/sec. Sufficient data is recorded to accurately recreate the original base data and winshear alerts when a base data tape it is played back through the system. This means that *all* data within the 240-gate range of the gust-front and microburst algorithms is, as well as *some* of the data in the less critical region beyond 240 range gates (in order to reduce the bandwidth requirement below the 3 Mb/sec upper limit). Recording only a subset (one scan every minute) of the long range data is somewhat of a compromise, since the scan-to-scan averaging of the lags data will not perform identically to real-time when data is played back, but it is felt that the data will be sufficient to reproduce the original long-range six-level weather and storm motion products with reasonably good accuracy.

Two factors affect the design of the format - the alternating low/high beam mode used for WSP operation, and the 105 degree precession of the alternating beam switch point used to 'spread out' the effects of the corrupted data at the low/high beam switch location. The chosen format groups a full low-beam cycle (360 + 105 degrees) and a full high-beam cycle (also 360+105 degrees) into a single 'volume scan', allowing each volume scan to stand on it's own. Given that the antenna scan rate is typically 78.3 degrees/sec, each volume scan represents approximately 11.9 seconds worth of data (~5 volume scans/min.)

Each volume scan file contains a file header, followed by a volume scan's worth of radar pulses. Time-series data tapes are simply a collection of time-series data files, with an EOF marker between each file on the tape, and a double EOF (End-Of-Tape) following the last file. Prepended to the beginning of each tape are several files containing the auxiliary information needed to accurately process the time-series data. These auxiliary files include:

1. The current VSP database file
2. The current STC map
3. The current clutter map

These files are stored using a generic file header block (contains file name), followed by a byte-for-byte copy of the corresponding disk file, using a tape blocksize of 64K. As with the time-series files, each auxiliary information file on the tape is followed by an EOF.

C.3.1 Time-Series Data File Header

The C structure describing the file header is shown below:

```
/*
 * The most common range spacings for most IQ files will be:
 *
 * Limited range (to save space)
 *
 *  One segment of 240 gates, single gate spacing.
 *
 * Full range
 *
 * Two segments.  First segment is 240 gates with single gate spacing.
 * Second segment is 180 gates with 4 gate spacing.
 */

#define IQDATA_ID          12

#define IQ_VERSION_NUMBER     4
#define IQ_FL3_FORMAT      0
#define IQ_ASR9_FORMAT      1
#define IQ_PHASE2_FORMAT      2
#define IQ_MAX_RANGE_SEGS     4
#define IQ_LOW_BEAM        (0x1)
#define IQ_HIGH_BEAM       (0x2)
#define IQ_BOTH_BEAMS       (IQ_LOW_BEAM | IQ_HIGH_BEAM)

#define IQ_AUXINFO_MAX_BLOCKS 10

/* Auxiliary info types.  (only 1 so far) */
#define IQ_AUXINFO_CPIOFFSET  1


/*
 * Number of spare words in header structure.  Make sure to decrement this
```

```
 * appropriately if some spares are used!
 */
#define IQ_NSPARES  33


typedef struct
{
    int         type;           /* IDDATA_ID */
    int         hdrBytes;       /* Header length in bytes */
    int         version;        /* Header version num */
    int         reserved;

    int         tapeId;         /* Unique ID for each 8mm tape */
    int         volumeScanNum;     /* Sequential volume scan number */
    int         volumeScanSpacing; /* 1 = every scan... */
    int         pulseHdrBytes ;    /* Probably always be 40 (10 words) */
    int         pulseBytes ;    /* hdr + data */
    int         tapeBlockSize ;    /* Multiple of pulse bytes */

    WSPTime32   startTime ;       /* Data as derived from first IQ pulse */

    int             nSegs;
    IQFileRangeSeg rangeSeg[IQ_MAX_RANGE_SEGS];

    int         reserved2;

    /*
     * Number of CPI's that beam switch precesses by on each lo/hi beam
     * 'scan'. Nominally 75 CPI's - probably won't change.
     */
    int         beamSwitchPrecessCPIs;

    char        siteName[16];      /* "ABQ", "BWI", ... */
    char        orgName[64];       /* "LL", "FAA", "NORTHROP", Etc... */

    /*
     * The following fields were originally intended to allow each IQ file to be tagged with the name of the
     * calibration (VSP), STC, and clutter map files in use when the data was recorded.  Linkage of the IQ
     * file to auxiliary files in this manner turned out to be overly cumbersome, so these fields are now
     * unused.
     */
    char        calibFile[64];
    char        stcFile[64];
    char        cluttFile[64];
    char        spareFile[64];

    char        comments[1024]; /* "Microburst's breaking out to west.." */


    /*
     * Auxiliary info index structure.  Auxiliary information blocks are typically appended to disk (not
     * tape) files by analysis programs to, for example, enable fast lookups of particular pulse/range
     * gates.  Such a lookup table can be generated by making one pass through the disk file, and then
     * the lookup table 'record' can be stored at the end of the file. For raw tape files, 'nAuxInfoBlocks'
```

```
 *  will always equal 0.
 */

    int             nAuxInfoBlocks;
    IQAuxInfoIndex  auxInfoIndex[IQ_AUXINFO_MAX_BLOCKS];

    /* New fields for version 4 */
    float       latitude;      /* degrees */
    float       longitude;      /* degrees */
    float       altitude;      /* meters */
    int         timeZone;      /* minutes behind GMT */
    float       magDeclination; /* degrees */
    int         frequencyA;    /* Channel A freq (MHz) */
    int         frequencyB;    /* Channel B freq (MHz) */

    int         spares[IQ_NSPARES];

} IQFileHdr ;


typedef struct
{
    int firstGate;
    int nGates;
    int gateSpacing;      /* 1 = every range gate, 2 = every other gate.... */
    int spare;

} IQFileRangeSeg;



typedef struct
{
  int type;
  int bytes;
  int startOffset;
  int spare;

} IQAuxInfoIndex;
```

C.3.2 Radar Pulse Data

Each pulse consists of a 10-word header and 2 words of data for each range gate that was recorded. Table C-1 describes the format of the header words. Table C-2 describes the format of the data words. Note that all words in the radar pulse data stream are 32-bits wide.

TABLE C-1

Radar Pulse Header

Word	Description
0	Radar/Switch Status Bits 16,17 SW103/104 status (00 = Lo, 01 = Hi, 10 = Circ) Bit 19 SW106 status (0 = Chan A, 1 = Chan B) Bit 20 Online channel (0 = A, 1 = B) Bit 21 Polariz (1 = linear, 0 = circ) Bit 22 Beam (0 = Low, 1 = High)
1	24-bit 1.3 MHz counter
2	16-bit pulse sequence counter

3	ACP / ARP counters
	    Bits 0-11 ACP counter (0-4095)
	    Bits 12-15 ARP counter (0-16)

4	1st half of data for range gate [-3]. The timing chain is set up so that the COHO sample from the RVP7 appears at range gate [-3]. The data from the COHO sample is used as a phase reference for all subsequent RVP7 data in a radar pulse. See the format of range gate data following this table for bit descriptions.
5	2nd half of data for range gate[-3] (RVP7 COHO sample)
6	Test Pattern (0xAAAA5555)
7	Test Pattern (0x5555AAAA)

8	Time Word #1
	    Bits 0-4 Day (1-31)
	    Bits 5-8 Month (1-12)
	    Bits 9-20 Year (years >= 2000 stored as full 4-digit year)

9	Time Word #2
	    Bits 12-17 Second (Note: time stored as GMT)
	    Bits 18-23 Minute
	    Bits 24-28 Hour

Two 32-bit words of data are stored per range gate. Each pair of words contains the A/D samples from the target channel receiver and the A/D samples from the WSP's weather receiver. A bit containing, the state of the target channel beam switch is also included, allowing the WSP to determine the correct processing path on a gate-by-gate basis.

TABLE C-2

Radar Data Word Pair

Words	Description
10 + gate*2	Target Channel Rx Data + RVP7 shift bits Bits 0-11 Targ Rx I sample (2's complement) Bits 12-23 Targ Rx Q sample (2's complement) Bits 24,25 RVP7 I sample exponent Bits 26,27 RVP7 Q sample exponent Bit 28 Targ Rx low/high beam status (0=Low beam, 1 = high beam)
11 + gate*2	RVP7 Rx Data Bits 0-14 I sample mantissa (2's complement) Bits 15-29 Q sample mantissa (2's complement)

The target receiver I,Q samples are simply stored as 12-bit signed integers. The data from the WSP receiver is split into 15-bit mantissa and 2-bit exponent portions, to allow for the receiver's wider dynamic range. The floating-point value for a WSP 17-bit quantity is given by:

$$value = (mantissa * 16^{exponent}).$$

Note that the phase of all the WSP samples is relative to the measured phase of the COHO at the start of the pulse (available in words 4,5 of the pulse header), and this must be corrected for to recover the true phase of the signal. This is accomplished by multiplying the sample at each range gate by the complex conjugate of the COHO phase sample.

C.3.3 Generic File Header

The C structure for the generic file header (used for storing auxiliary information files to tape) is shown below.

```
#define GENERIC_FILE_ID       4
#define GENERIC_FILE_VERSION  1

typedef struct
{
// 4-word header compatible with IQ, BaseData, file headers
int type;
int hdrBytes;
int version;
int reserved;

// Attribute fields to restoration of original file attributes on a restore
int attrsValid;  // 1 if following attr contain valid data
```

```
    int ownerId;
    int perms;
    int month;   // File time
    int day;
    int year;
    int hour;
    int minute;
    int second;

    char filename[512];
}
```

The filenames used for the VSP, STC, and clutter map auxiliary files are of the general form, 'vspdb.vsp', 'sigmet.stc', and 'MMDDYY-HH:MM.clutt', respectively. The name of the clutter map reflects it's creation date. Note that the file attribute and time fields of the header are not currently used by the software that creates the time-series data tapes, and are all set to 0.

C.4 Base Data Format

'Base Data' consists primarily of reflectivity, velocity, spectrum width, and quality flag data for all range gates processed by the WSP. Data from the WSP's wind sensor is also integrated into the base data stream. Base data serves as the input for all the product generation algorithms, such as the Microburst and Gust Front detection algorithms. The data format is file based, with each base data file containing a header and a number of scans (antenna revolutions) of polar radar data. Base data tapes are simply a collection of base data files, with an EOF marker between each file on the tape, and a double EOF (EOT) following the last file on the tape.

The primary goals of base data recordings are to enable accurate recreations of weather events and allow for algorithm evaluation and refinement. The amount of data recorded is minimized through knowledge of the algorithm requirements. The microburst algorithm, for example, requires continuous recording of seven of the base-data products out to a range of 160 gates, while the storm motion algorithm requires data at 12-scan (~55 sec.) intervals out to the full range of the radar. For convenience, data from multiple antenna scans are grouped into a single file covering approximately a 1-minute period (12 scans).

C.4.1 Base Data File Header

Each file contains a header record containing site-related information, as well as time information and comment field for the benefit of inventory and other access programs. Including the site-related info in every file essentially makes each file a standalone entity.

The C structures and definitions describing the file header is shown below

```
#define BASEDATA_ID          (14)
#define BASEDATA_VERSION  (1)
#define BD_MAX_SCANS      (100)

#define BD_MAX_COMMENT_LEN  (2048)


typedef struct
```

```
{
  int      type;              /* BASEDATA_ID (14) */
  int      hdrBytes;
  int      version;           /* BASEDATA_VERSION (currently 1) */
  int      spare;

  int      scanGroupNum;
  int      tapeBlockSize;     /* Block size in bytes for files stored on 8mm tapes */

  WSPTime32  startTime;
  WSPTime32  writeTime;

  /* Site information */
  BDSiteInfo siteInfo;

  /*
   * Channel and polarization info for first radial in scan group.
   */
  int    chan;        /* 0 = A_CHAN, 1 = B_CHAN */
  int    polariz;     /* 0 = LIN_POLARIZ,  1 = CIRC_POLARIZ */

  /*
   * Number of scans in group, with byte offsets to the beginning of
   * each scan for rapid access using lseek.
   */
  int    nScans;
  int    scanOffset[BD_MAX_SCANS];

  char   comment[BD_MAX_COMMENT_LEN];   /* Comment entered by user at record time */

} BDScanGroupHdr;

typedef struct
{
  char   siteName[16];    /* (e.g. ABQ,DAL,...) */
  float  latitude;            /* degrees */
  float  longitude;           /* degrees */
  float  altitude;            /* meters */
  int    timeZone;            /* minutes behind GMT */
  float  magDeclination;  /* degrees */
  int    frequencyA;          /* Channel A freq (MHz) (Note: PRF in radial hdr) */
  int    frequencyB;          /* Channel B freq (MHz) */
  int    spare;

} BDSiteInfo;
```

C.4.2 Base Data Record Format

Following the file header are a number of scan's worth of base data records, each record containing either a 'radial' of reflectivity, velocity and flags data, or a wind data sample. The starting offset for the beginning

of data records in the file is determined using the *hdrBytes* field in the file header.

```
/*
 * Union of data types carried in a base data record.
 */
typedef union
{
  BDRecordHdr hdr;      /* header only - useful when determining rec type */
  Radial      radial;   /* header + radial data */
  BDWind      wind;     /* header + wind data */

} BDRecord;


/*
 * Header for base data record.
 */
typedef struct
{
  short type;        /* 1 for radial data, 2 for control msg (not yet used),
                      * 3 for wind data
                      */
  short bytes ;      /* Length of entire record, including this header. */

} BDRecordHdr;
```

C.4.3   Base Data Radial Format

Each radial contains data for a single 1.4 degree azimuth wedge, and is made up of header and data portions. The data portion contains data for a number of different products, and itself is broken down into product header portions and product data portions. Maintaining separate headers for each product allows for each product to have separate range coverage, scaling, etc... The radial and product C structures are shown below.

```
/*
 * Bit definitions for RadialHdr->flags field.  Note that BEAM field is an indicator
 * of the LO/HI beamswitch position at time of radial creation, but radials
 * nevertheless contain products for both beams! The beamswitch position
 * is mainly provided for diagnosing switch hardware problems.
 */
#define END_OF_SCAN       (0x1)     /* generic end of scan flag */
#define BD_CHAN_MASK      (0x2)     /* 0 = A_CHAN, 1 = B_CHAN */
#define BD_POLARIZ_MASK   (0x4)     /* 0 = LIN_POLARIZ, 1 = CIRC_POLARIZ */
#define BD_BEAM_MASK      (0x8)     /* 0 = LO_BEAM, 1 = HI_BEAM */


typedef struct
{
  RadialHdr    hdr ;
  unsigned char data[1] ;  /* Product headers/data  (Variably sized array) */
```

```
} Radial ;

typedef struct
{
  short     type ;       /* 1 for radial data, 2 for control messages (not yet used) */
  short     bytes ;      /* Length of radial record, including this header. */
  short     gateSize ;   /* Gate size in meters. */
  short     sector ;     /* 0-256 */

  RadialTime  time ;

  short     scanNum;
  short     tiltNum;    /* Unused in WSP system */
  short     az;         /* Deg x 10 */
  short     el;         /* Deg x 10 */
  short     prf1;       /* Primary PRF */
  short     prf2;       /* 2nd PRF for dual-prf radars (ASR-9) */
  short     flags;      /* END_OF_TILT bit, among others */
  short     nProds ;    /* Number of products in radial */

} RadialHdr ;

typedef struct
{
  short month;
  short day;
  short year;
  short hour;
  short minute;
  short second;

} RadialTime;
```

C.4.4 Product-level data structures

```
typedef struct
{
  ProdHdr     hdr ;
  unsigned char data[1] ;   /* Variably sized */
} Prod ;


#define MAX_PROD_SEGS  4        /* Maximum number of range segments in product */

typedef struct
{
  short     type ;
  short     bytes ;              /* Total length of this product record, including this header */
  short     bytesPerGate ;
  short     scale;
```

```
    short      offset;
    short      nRangeSegs ;      /* Number of range segments in product */
    ProdRangeSeg rangeSeg[MAX_PROD_SEGS] ;
} ProdHdr ;

/*
 *  Each product radial  may have separate regions (range segments) with
 *  different gate sizes.   The gate size is a multiple of the fundamental gate size
 *   supported by the ASR-9 radar (1/16th NM, or 115.75 m).  That multiple is stored
 *   in the 'gateSpacing' field for each range segment.   A value of 1 corresponds to a
 *  gate size of 115.75m.
 */
typedef struct
{
    short firstGate ;         /* Starting gate for range segment */
    short nGates ;            /* Number of range gates in segment */
    short gateSpacing ;  /* Gate spacing in gates (1,2,3,4 etc.. ) */
    short gateSize ;         /* Unused in WSP system  */

} ProdRangeSeg ;


/* Code for bad (missing) product data.  Most negative 16-bit and 8-bit numbers */
#define BADVAL    (-32768)
#define BADVAL8   (-128)


/* Flags Product bit definitions. */

#define AP_FLAG               0x1
#define CLUTTER_FLAG          0x2
#define SECOND_TRIP_FLAG      0x4

/* Separate AP flag used in new clutter-filter-based AP detection scheme. */
#define FILT_AP_FLAG      0x80
```

C.4.5   Base Data Wind Record Format

Base data records with a type of 3 contain wind data records. The WSP system supports the acquisition of wind data from up to 16 wind stations, although at most sites, the actual number will be six (LLWAS2 network) or one (single station ASOS). The C structures associated with the wind records are shown below.

```
#define BD_WIND_STATION_ID_LEN 16
#define BD_WIND_AIRPORT_ID_LEN 16
#define BD_WIND_MAX_STATIONS  16

typedef struct
{
    BDRecordHdr  hdr;
```

78

```
    BDTime      timeStamp;
    float       or_degrees;

    char airport_id[BD_WIND_AIRPORT_ID_LEN]; // eg, "ABQ", "MEM"
    short cfa_direction_degs;            // 0-360 valid, 999 invalid data
    short cfa_speed_knots;               // 0-99 valid, 255 invalid data
    short cfa_gust_knots;                // 0-99 valid, 255 invalid data

    short stationCount;                  // 0, 1, 2 .. Max_LLWAS2_Stations
    BDWindStation  station[BD_WIND_MAX_STATIONS];

} BDWind;

typedef struct
{
  char  id[BD_WIND_STATION_ID_LEN];  // eg, "35A", "21CA"
  float latitude;              // degrees West (NYC < 0.0)
  float longitude;             // degrees North (NYC > 0.0)
  float altitude;              // meters above mean sea level
  short direction_degs;        // 0-360 valid, 999 invalid data
  short speed_knots;           // 0-99 valid, 255 invalid data
  short gust_knots;            // 0-99 valid, 255 invalid data
  short flags;                 // see Flag bit-map values

} BDWindStation;
```

GLOSSARY

AMDA	ASR-9 Microburst Detection Algorithm
AP	Anomalous Propagation
API	Application Programming Interface
ARENA	AREa Noted for Attention
ARINC	Aeronautical Radio INCorporated
ASOS	Automated Surface Observing System
ATC	Air Traffic Control
COHO	Coherent Oscillator
COTS	Commercial Off-The Shelf
CPI	Coherent Processing Interval
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECPI	Extended Coherent Processing Interval
FAA	Federal Aviation Administration
FTC	Functional Template Correlation
GPS	Global Positioning System
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronic Engineers
LAN	Local Area Network
LLWAS	Low Level Windshear Alert System
MDT	Maintenance Display Terminal
MIGFA	Machine Intelligent Gust Front Algorithm
NWS	National Weather Service
PRF	Pulse Repetition Frequency
PSF	Program Support Facility
RAID	Redundant Array of Inexpensive Disks
RDP	Radar Data Processor
RDT	Ribbon Display Terminal
SBC	Single Board Computer
SCSI	Small Computer Systems Interface
SD	Situation Display
SEP	Storm Extrapolated Position
STC	Sensitivity Time Control
TCP/IP	Transmission Control Protocol/Internet Protocol
TDWR	Terminal Doppler Weather Radar
TRACON	Terminal Radar Control ??
TWIP	Terminal Weather Information for Pilots
UDP	User Datagram Protocol
VSP	Variable Site Parameter
WSP	Weather Systems Processor
XDR	eXternal Data Representation

REFERENCES

1. Weber, M.E., Noyes, T.A., "Wind Shear Detection with Airport Surveillance Radars", MIT Lincoln Laboratory, Lincoln Laboratory Journal, Volume 2, no. 3, p. 511-26, 1989

2. Weber, M.E., "Dual-Beam Autocorrelation Based Wind Estimates from Airport Surveillance Radar Signals", MIT Lincoln Laboratory, Project Report ATC-167, FAA/PS-89-5, Jun 1989.

3. Newell, O.J., Cullen, J.A., "ASR-9 Microburst Detection Algorithm", MIT Lincoln Laboratory, Project Report ATC-197, 1993.

4. R.L. Delanoy, S.W. Troxel, "The Machine Intelligent Gust Front Algorithm", MIT Lincoln Laboratory, Project Report ATC-196, 1993

5. Chornoboy, E.S., "Storm Tracking for TDWR: A Correlation Algorithm Design and Evaluation", MIT Lincoln Laboratory, Project Report ATC-182, 1992.

6. F.W.Wilson Jr. R.K Goodrich, K. Brislawn, "Enclosing Shapes for Single Doppler Radar Features", Journal of Atmospheric and Oceanic Technology, Volume 9, No. 2, 97-107, April 1992.

7. Weber, M.E., Stone, M.L., "Low Altitude Wind Shear Detection Using Airport Surveillance Radars", IEEE Aerospace and Electronics Systems Magazine, Volume 10, no. 6, p. 3-9.

8. Weber, M.E., "ASR Weather Systems Processor Signal Processing Algorithms", MIT Lincoln Laboratory, Project Report ATC-, April 18, 1997

9. Saia, J.J., Stone, M.L., Weber, M.E, "A Description of the Interfaces between the Weather Systems Processor (WSP) and the Airport Surveillance Radar (ASR-9)", MIT Lincoln Laboratory, Project Report ATC-259, 1997

10. Lewine, D., "POSIX Programmer's Guide", O'Reilly & Associates, 1994

11. Lippman, S.B, "C++ Primer", Addison-Wesley, 1991

12. Eckel, B., "Thinking in C++", Prentice Hall, 1993

13. Welch, B.B., "Practical Programming in Tcl and Tk", Prentice Hall, 1995

14. Harrison, M., McLennan, M., "Effective Tcl/Tk Programming", Addison-Wesley, 1998