

CR-189392

**EOS Testbed System
NASA Project Number 5555-07**

Final 1994 Report

1N-61

205

202/5

p. 55

**Principle Investigator: William Emery
University of Colorado
Aerospace Engineering
Boulder, Colorado**

September 6, 1994

(NASA-CR-189392) A LAND-SURFACE
TESTBED FOR EOSDIS Final Report,
1994 (Colorado Univ.) 55 p

N95-17327

Unclas

G3/61 0033945

History

The downloading and archival of satellite data is not new. There are several thousand downloading and archive sites throughout the world. The newest of these archival sites is the EOS Testbed system (data system) which allows for an archive site to use low cost storage devices and software to archive and transport data to end users over the Internet.

The data system started in 1991 from a National Aeronautics and Space Administration (NASA) grant to provide satellite images to end users via the Internet. The original proposal was to use Macintosh computers and to deliver preprocessed images of 520 by 520 pixels centered over Colorado. These images were of all five channels (.03 through 12.8mhz bands); channel 1 and 2 being the visible channels, channel 3 the nearly visible nearly infrared, and channels 4 and 5 being strictly infrared. These images were processed from data supplied by the National Oceanography and Atmospheric Administration (NOAA) polar orbiting satellites, NOAA's 9, 11, and 12.

The post processing of the data was conducted by the Colorado Center for Astrodynamic Research (CCAR) using a post launch navigation software developed at the University of Colorado and the National Center for Atmospheric Research (NCAR) (Emery, Baldwin, Rosbrough, 1991). This software would geo-register the data to a prearranged map projection and center latitude and longitude point. The range of the image was set to 5.0 degrees (2.5 degrees from the center point) to set the resolution of the image to its maximum (1km per pixel at nadir).

Images were processed on a daily basis and then were transmitted using the Internet to NCAR and stored on the Mass Storage device (StorageTek 660). The number of images processed ranged from three to four passes per day from 1989 to 1991. Though each individual image was small in storage size, the volume became quite large as new images were added to the existing file directory on a daily basis.

At this time, we had no users of the system nor did we have a system that would deliver the images to the end user. It became apparent that a Macintosh would not be compatible with operations on the StorageTek 660 nor would it be able to work as a gateway server to the outside world. For these reasons it was decided to purchase a Digital Equipment Corp. Dec5000. In 1991, this was one of the fastest workstations and could be used as a gateway server as well as a communication device to the StorageTek 660.

The original system design was very simple. The system would catalog the images in a simple scrolled window. The user then could pick by name the image or images he or she wanted and the Dec5000 would make a call to the StorageTek 660 to retrieve the image. Once an image was returned to the Dec5000, it would place the image in an anonymous File Transfer Protocol (FTP) directory for the end user to pick-up. This setup worked fine when we had only a handful of users, but as end users began to grow the single FTP directory was becoming cluttered with hundreds of images and users were becoming confused about which images belonged to them.

It became apparent that the system would need to track the end users name and the image files selected and place the images into a new ftp directory that would be named after the user. This was not a simple task. The StorageTek 660 uses a software protocol called the MASnet/Internet Gateway Server (Migs). Migs relies on a string of default variables and returns a sequence number for each request whether the request is successful or fails. Since the request comes from a machine and not a user the StorageTek 660 does not know who ordered a request. Once an order has been processed, the StorageTek 660 sends a mail message to the ordering machine on the status of the request . Parsing a mail message becomes quite complex and must work as follows:

The gateway computer allows a user to login. The user must type in his or her name. This information is parsed into a file and stored. The user makes an order or orders and these individual orders are given a sequence number. These numbers are then parsed into the same file as the users name. Now the gateway computer knows the files ordered the name of the file, and the user name. As files are returned to the gateway computer, the sequence number is parsed from the mail message along with the filename. The gateway computer then renames the file to the correct name and places the image file or files into a directory the computer creates for the user under his or her name.

This architecture allowed end users to pick up only the images they had ordered. Users quickly learned they could order several images (in some cases several hundred images) and later pick up what images they felt were useful. This lead to the expansion of the anonymous ftp drive from one gigabyte to 4 gigabytes. After contacting end users we found that the reason they were ordering so many images was because they had no idea of whether the image had cloud or whether it was clear. Subsequently, they would order all the images and sort through them at a later time. What was lacking was a way of seeing exactly what you were ordering before you place the MIGS command.

Browsing by image name was not enough. End users needed to be able to view an image before it was ordered. This request lead to the visual browse of the images. To my knowledge we are the first to create on-line visual browse of AVHRR images. This was an even more complex task than the parsing of the sequence mail messages as described above.

The gateway computer would need to use a language that would be able to transmit an image over the Internet and display that image on the end user's computer. Motif Xwindows was chosen to perform this task. Xwindows would allow for the capture of an image and the placement of it within a window widget

called a bulletin board widget. With the image captured, it could then be transmitted within the widget and displayed on any computer being serviced by Motif or Open windows.

All display widgets use a colortable (colortable being the pixels color values using the Red, Green, Blue matrix and being scaled from 0 to 254) . This colortable must be created each time an image is displayed. Because a colortable can be altered for specific needs or display purposes we found that a colortable had to be created as a default to support the browse images. What this means is, if the users computer did not support a colortable that would display the browse image, the gateway computer would first send a default colortable to allow the end user to view the browse image (Figure 1).

With the feature of on-line browse the amount of non essential mass store retrievals dropped. What developed next was that the area coverage was not large enough for many users. We found that users now wanted areas that were not within our preprocessed image sizes. It was determined that we would start processing larger images to include coverage over the west coast. Increasing the image size increased the volume of data that was transmitted back to the end user. Once an image was delivered to the end user he or she would then crop the section of the image that they needed and discarded the rest. Pressure from end users lead to the final system (navigate) which was designed to allow the end user to geo-register his or her own image in the zone of coverage, and the type of resolution and map projection that would best suit their needs.

Phase 1 Navigate System

It was determined that the navigate system would use our existing navigation program and have Xwindow interface to build the command line that would geo-register the images (Figure 2). Since the navigation code had never

been developed to geo-register several images at once, the data system would need to be modified to spawn off a child to wait until a file was returned from the StorageTek 660 to navigate the image or images. Prior to this time, all of the files on the StorageTek 660 were preprocessed images. Now it would be necessary to store a full AVHRR data pass of 130 megabytes each on the StorageTek 660. This type of data had been archived from late 1989 and required several months to be placed on-line using the StorageTek 660. When an order was placed, the Xwindow display windows would spawn a child to hold the command line in active memory. This process would contain information on how to geo-register the images, the user name and the delivery file directory. The command to return a file from the StorageTek 660 went through the MIGS gateway computer, where the status of the request would be assigned. This first design relied on the StorageTek 660 as the file storage device. Because of access problems and time constraints, the first phase on navigate never worked correctly. The StorageTek 660 would deliver less than 50% of the files on time. Before the navigate system, it did not matter if the files took several minutes or even several hours to return. The intermittent delivery time of the data files caused serious problems in the navigate system. When files were not returned in a reasonable period of time, the navigate system would keep spooling the navigate command in memory. This spooling would continue to build until the computers memory was full and cause the computer to crash. By setting a time-out to the spooling process, meant if a data file did not return on time the process was stopped. Either of the above cases meant users were not getting their images. Since we were using the StorageTek 660 for free we were not a high priority of NCAR to service our needs. We had several meetings with NCAR and it was determined that a test would be run to find out exactly how long it was taking to return a single file. After a week of testing, it was determined that the average file took between 25 and 35 minutes to be returned to the gateway

computer. This time constraint was much too long. Memory processes could not be spooled this length of time and not become corrupt. This led to the navigate system not making any calls to the StorageTek 660 and allowed end users access to only current passes that were updated daily. Complaints about the lack of past data and the data system having no control over the NCAR equipment led to phase 2 of the navigate system.

Phase 2 Navigate II

Navigate II was developed from all the information that had been gathered from the previous years of work and research. The new system would have its own storage device, track its own files and tapes, allow sorts on the data prior to ordering and would have a windowing design that would simplify all of the complex operations that are hidden from the user.

Hardware Design

The first phase of navigate used Digital DEC Station 5000 to be the gateway computer. This system was extremely reliable in its hardware. For this, DEC was again approached to find a machine that would have extensive I/O operations and be able to still allow file transfers while being hammered by internal jobs. From all machines that were proposed, it came to our attention that the new DEC Alphas were going to be the best choice for our needs.

The original machine that was offered was a Digital DEC Station 3000/400 (a mid range Alpha). Our original plans were to use one high speed machine to operate both the data storage system as well as navigate the images. By testing the navigation program on a loaner Alpha, we found that the navigation code would take several minutes and would need to be executed on a separate machine. This

expanded and complicated the system design in that now two machines would need to work as one.

Again, Digital was contacted on our current design and the use of a Turbo Channel connection was advised to allow the two computers to be hardwired to each other and work as one. The Turbo Channel proved to be a dismal disappointment in that it was only two ethernet cards that would allow communications between two machines that were not connected via the Internet. It was later discovered that the Turbo Channel could be made to work as we intended but we would need a special router and the cost of the router was half the price of a new DEC Alpha.

Our previous work with navigate showed us that we would need a disk farm to hold and transfer data. Western Scientific had high speed drives that could accept our data at rates above 5 megabytes per second. This speed would allow a data file to be transferred from tape to drive in less than 5 minutes. The drives selected were two 8 gigabyte drives. The 16 gigabytes would allow us to offer more files and allow more users to access the system without complications.

Now with the driver and storage requirements completed, it was time to define our needs for a storage device. Our AVHRR passes are on average, 130 megabytes in size and we archive three to four passes per day. It became quite apparent that we would need to archive by 1994, one full terabyte of data. This size requirement immediately eliminated a great deal of the storage systems.

Research into storage devices showed that we would need to purchase equipment and software that was ranging from \$50,000 to \$150,000 US. to immitate the StorageTek 660. This cost structure was far beyond the budget requirements.

After an extensive search, the three final choices became: the 4mm Dat drive, a read write CD-ROM and the Exabyte 120i 8mm tape system. All of our data was currently being stored on Exabyte 8mm tape and using a 4mm Dat system

would mean that all data would need to be rewritten, and tapes could not be shared with the current lab drives. At the time of procurement, the 4mm Dats did not have the final storage requirement of a full terabyte. Read write CD-ROM is useful when the files are exactly the same size and all of the data is ready to write at a single time. This system was not viable for our needs in that we would need to store data until we had exactly enough to fill a CD and then write the CD. With current SONY read write CDRoms you can write several times to a CD but the cost again is prohibitive for our project.

The Exabyte 120i was chosen as the medium for our data storage system. The 120i offered many flexibility's we needed. First, the 120i uses four Exabyte drives giving us the ability to [read, read, read, read] or [write, write, write, write] or any combination of the above. This would allow us to serve several users within a short period of time. The 120i, using the Exabyte 8500C compression drives, will hold over one terabyte on its 120 tapes (note 116 slots within the 120i and 4 tapes in the drives).

Unfortunately, the 120i with all of its hardware capabilities did not use any type of driver software to actuate the robotics arm in the fashion that we would need. At this time, Exabyte was approached and manuals were supplied explaining the hardware operations of the 120i and the 8500C drives. From this point, it became apparent that driver software would have to be developed from scratch and implemented onto our DEC Alphas to run the meteorological data archive system (MDAS).

Software Design

Meteorological Data Archive Software

By researching software packages to run our data storage system we found that there was a lot of freeware software that could be used as a building tool to create a data storage system. GNU database software was chosen as the base tool and standard C programming for building the data storage software. GNU is a very powerful package in that it allows for remote procedure calls (RPC) to access files and servers at multiple times. GNU also allows any type of interface to be used as the callback design function. By using GNU, we saved months of programming and made the software easily portable to any type of computer. (assuming that the computer has a C compiler).

Exabyte Driver Software

The driver software became more and more complex and the operations started to take shape. The driver not only needed to operate from the hardware calls, it needed to have multiple error checking returns placed in the code to allow the MDAS system to work correctly. Since the driver software was developed for the 120i running on an DEC Alpha, we are not sure how the system would operate if it were to be recompiled on another type of system running ULTRIX UNIX or Solaris. The driver software handles only the operations of the Exabyte 120i robotic arm . The driver software tracks errors and usage of the hardware. The driver software will return errors if and only if the arm cannot complete it's task. The software does not handle any maintenance of the drives or any type of error dealing with the status of a file or errors from the 8500C drives themselves. The driver software is truly the hardware coding for the driver arm only.

Interface Software

From the first conception of the data system, Motif Xwindows has been the primary window interface (relying on Widget Create Library calls (WCL) to

develop and manage the windowing interface). WCL is freeware distributed through anonymous ftp at Massachusetts Institute of Technology (MIT). The original code was developed for Sun workstations using X11 version R5 for Motif. Using WCL as a GUI builder allows C programming code to be passed into callbacks that intern are keyed through WCL by the use of buttons, pulldowns, etc. WCL allows full access to an applications defaults file (Figure 3). This file controls the colors, design, and placement of windows. This application defaults file allows the windowing design to be flexible for the implementation of the interface.

Implementation of System

Hardware

Our hardware did not arrive until the beginning of 1994 ,putting us eight months behind schedule. The first piece of equipment to arrive was the Exabyte 120 along with two Exabyte 8500C drives. The 120 was originally thought to operate exactly like a test Exabyte 10i unit we had based the driver software operations on. We found that there were a few changes that needed to be implemented to make the 120i work correctly. This meant using a self diagnostic software system to track the calls from the Alpha (a lab loaner machine) to the 120. Through the use of this tracking software, we could see the error messages and the calls that were being passed to and from the 120 allowing us to correct the driver software bugs. The task now became one of tweaking the CAM calls for the 120i to allow it to operate correctly.

The original DEC 3000/400 machine was backordered for over 6 months and Digital decided to give us their newest machine, a DEC 3000/600. The 600ds is even faster than the 400ds (a 600 runs at a clock speed of 150) allowing us to operate more efficiently. As we started hooking up all of the hardware, we found

complications in that the Turbo Channel did not work (as previously mentioned). In addition, our two new alphas were delivered without an operating system and we needed to find a version of Open Software Foundation (OSF).

Software

Operating System Software

OSF is the latest type of operating system to be used by the Alphas. It allows for complex calls with the versatility of other operating systems (at least, this is what we were originally hoping for). What we found was that OSF should not have been released. Throughout the past five months, each machine has been loaded with a completely new operating system five times. The first version to be loaded was version 1.2, followed by version 1.3, then version 1.3B, then version 2.0, and finally version 2.0 revision 250.

Version 1.2 would not allow for any SCSI cam operations. Each time a drive was attached to the Alpha, the system would crash. There was to be a fix for this in version 1.3, but again, we could not get any CAM interfaces to operate. Version 1.3B finally fixed the CAM errors and allowed the driver software as well as the Exabyte drives to operate correctly. All of the interface software uses Motif X libraries being called from WCL. WCL would not compile on any version prior to version 2.0. Digital first claimed it was our software and not a problem on their part and that other groups were using Motif on the Alphas without any trouble. At this point, these groups were contacted and none of them had gotten Motif working correctly.

Through DBX tracking, we found that Motif was crashing before it ever got to main. This meant that it was a memory problem. We sent the error message to Digital and after a week of review they reported that there was a malloc problem and that it would be corrected in version 2.0. We now purchased version 2.0 and

with minor changes to the WCL code we were able to get the software working correctly and the interface brought up it's first windows.

The other side of the coin was that what problems Digital had solved in version 2.0, they created new problems in the CAM area. Again, our drives became useless, and the driver arm would crash each time the machine was rebooted. Digital was again approached and we found that all of our complaints would be corrected in version 3.0. At this point, we could not wait. The data system was off-line and needed to be operational by mid summer. It was then decided to run one Alpha with version 2.0, to run the windows interface, and the other with version 1.3b, to run the Exabyte 120. This is not the best scenario but time constraints make for drastic decisions.

Driver Software

The driver software was developed at the University of Colorado (CCAR) to actuate and monitor the robotic arm in the Exabyte 120. The 120 did not come with any type of software to drive the arm or intrinsic functions to use the 120 as a store and retrieve data system. The driver needed to be able to position the arm in the correct position to pick out a tape and place the tape within the 8500C tape drive. Coding for this operation meant getting into the HEX stream of the machine. The robotic arm is belt driven and counts rotations on a spindle gear to determine the location. The base system knows of a home position and uses an optic sensor to locate positions within the 120. Knowing these positions, the robotic arm is able to continuously know it's position. If the driver arm loses it's position, the base coding will reset the arm and return it to home and put the arm into a configure mode to optically find the set location points and the drives. The 120 will also reset itself automatically every 80 calls. This reset caused difficulty for the MDAS system which interpreted this as an error. This reset is set by Exabyte and is not

able to be overridden or the duration changed. During this process, the 120 is off-line and no calls can be made to the unit. Once the arm is through finding all internal positions, it will continue to finish jobs placed through MDAS.

The coding to run the arm is written in C and relies on SCSI/CAM commands and returns. Being Common Access Method (CAM) specific, the calls are simplified and but since CAM is not an industry standard each call is specific to OSF 1.3B, making the driver only operational on an Alpha running OSF 1.3.B. Each command in SCSI must return a 0 or a 1 for completion or error. If an error is sensed, the driver code will fill out a data structure to interpret the error and place that error in the user error file (uerf) located in the root directory on the Alpha.

At this point, the driver software has failed to be operational on OSF version 2.0 or version 2.0 revision 250. Researching through Digital support has found that these versions do not understand what type of SCSI drives are attached to the computer. Because the CAM commands to Sleep, SleepLock, SleepUnlock all come from one library call (PRDVIS.MPL), version 2.0 or higher causes a reboot each time this type of call is placed to a tape drive. As of this time, it is a mystery as to the cause of the system crash, but Digital has informed us that with version 3.0 this will be corrected.

Interface Software

To contend the time constraints on the MDAS system, the entire user interface was developed using a DEC 5000 personal computer with WCL and the Motif X11 R5 libraries installed. The interface windows were operational in this environment and were ready to be ported to the Alphas as soon as they arrived. What we were not ready for was the problems that developed trying to port WCL to the Alphas. The first time WCL was installed it would core dump and cause the full operating system to be rebooted.

After a hit and miss approach, we found that we could compile WCL by compiling the sub directories MRI and ARI separately. After review of the WCL source code, we found that ARI was not being used for any of our type of windowing and was not needed. At this point we removed all of the links to ARI and compiled WCL using MRI only. With WCL compiled, we started to port the interface software onto the Alphas.

At this point, we started to get operation errors in trying to manage the interface. As discussed before, the windows were crashing before main could be initialized. This meant that the memory was not being allocated correctly. Malloc.h is a file that controls the size requirements for memory. This file is contained in the include files for the operating system (OSF). As mentioned above, we dealt with Digital on this problem and found that our operating system would need to be reinstalled to version 2.0 revision 250 for our memory problem to be corrected. This meant reinstalling yet another operating system and reinstalling WCL.

After the new operating system was reinstalled, we found that the interface windows were able to get past main but would crash before the parent window could be initialized. We started from scratch and reviewed all of WCL. Since WCL is freeware and we were installing on a machine that was new, there was absolutely no help from MIT, Digital, or other users of WCL. After a lengthy review we found that in WCL the operations rely heavily on the passing of integers. As WCL passes integers it expects to have integers returned. By returning an integer, this could mean returning an integer pointer or a long. We found that we needed to cast the integers to a long and this corrected WCL errors and allowed the WCL to operate correctly.

It should be noted that OSF is extremely picky on the types of pointers, character pointers, integer structures and castings that are in code not originally written on or for the Alphas. This selectivity caused hours of delay in trying to

compile code for the Alphas, especially those codes that are integer dependent. With WCL running correctly, the window interface was able to operate on the Alpha running OSF version 2.0 revision 250. It is our conclusion that WCL will not run on any other version of OSF with the possible exception of version 3.0 which is due to be released next month.

System Operation

Interface Windowing System

The interface for the MDAS system was developed to operate with simple window commands. Through the experience gained with the previous navigate systems, users required windows that were easily understood and easily operated. The initialization of the system requires a user to set up an xhost relationship between his or her computer and the gateway Alpha. This is a simple UNIX command

(>>xhost <gateway>).

Once the users machine returns the accept message

(<gateway> added to access control list)

the user can then telnet to the gateway computer with again a simple telnet command

(telnet <gateway>).

At this point, the gateway computer will ask for the login name of the user. The gateway computer is set up to use a single user and password for all users. This allows for multiple users without having to manage each single login. The user types in the login name and gives the password to the gateway computer. Once a user has been given access, all control options are defaulted and the user will not be able to break into the system. The gateway computer will ask the user for the name of his or her computer. The response can be by Internet protocol

number or by host table name. This request serves two purposes. First, the gateway computer initializes the windows to be transferred to the users computer. Second, the security file is getting the name of the machine that is being served.

Once the security file has been updated by the login response, it looks at the user name and machine from the register file and the xhost request. If there is a match then the user is allowed to continue. If there is not a match the user is not allowed access. Also at this point in time, a user is only allowed 5 navigation requests per day. The security file tracks this and will not allow a user back on the system for 24 hours. This security system should keep users from passing their login information out to others and will force users to register with the system administrator to gain access to the Navigate II system .

Motif Xwindows work in a parent child hierarchy format. The first window to appear is the main window known as the information dialog window (exp xxx). In the beginning of the navigate system users were asked to fill in their name, phone, and email address. Complaints were that this was too much for them to do each time they logged into the system. Navigate II has been altered to accept only an email address. All other information is parsed from the registration file for each user.

The information dialog window has two buttons one to clear the screen and allow you to retype your email and the other to gain access to the system. The accept callback actuates the security sequence to check the parameters and to allow the user to gain access. First the name in the email is verified as an actual user. This check is done by scanning the register file for the email sequence. If found the name and machine are placed into the active security file. Next, the callback checks the xhost display that has been typed in prior to the information dialog window being activated. If the xhost machine matches the users information then, the user

is accepted and a small dialog window will appear telling the user that they have been accepted and they can continue (Figure 4).

Once in the Navigate II system, the user can activate a menu that gives a popup window with four choices: Search Browse, Goes Images, Programs and Quit. The search browse button actuates the search window that must be activated in order to continue with the AVHRR navigation process. The user must first decide what latitude and longitude to center the search. Other options for the search include: starting and stopping time interval and which satellites data should be used. If the user does not know the latitude or longitude of the area but of a major city near the area of coverage, a pulldown window can be activated from the cities button to reveal some of the major cities within the coverage area. By selecting a city the latitude and longitude dialog text widgets are filled with the correct values (Figure 5).

Once a search is activated, the DoSearchCB will go out and evaluate each browse image and create a file that will contain the name of each browse image that has the search coordinates, time frame, and the satellite(s) selected (Figure 12). Once the system is finished, the SearchCatalogCB is activated to reveal the file names through a popup scrolled window (note: this also puts the list of the files in the navigate fileList popup window) (Figure 6).

When the user clicks on the file name, the ViewCB is activated to allow the browse image for that name to be displayed (Figure 13). This callback requires the use of the colors.c and colors.h files to allow the image to be placed within the window widget and to be viewed on any type of computer (Figure 1). The images are preprocessed, low resolution images and are viewed by use of a greyscale colortable. At the bottom of the viewing window are three buttons: one button returns you to the browse list that was generated by DoSearchCB, the next button activates the navigate window, and the third is a quit button.

The navigate window is the most complex window in the Navigate II system due to the operations that may be activated. The window can spawn four process commands. The first grabs the parameters in the window and used this information to create the navigation command line (filename, latitude, longitude, range, channels, and zenith angle). The second is a command line created to access the MDAS system (filename, and return path). Along with the navigate and MDAS commands, there are two other spooled processes that must be held in memory until the navigation process is complete. These processes produce an overlay map and scale a two byte image to an eight bit image. This one navigate window has a total of 7 callbacks associated with its operation (Figure 7).

Once a navigation process has been started, there is no way for the user to append or quit the external process. This feature was designed into the system to not allow files to be deleted from the queue table in MDAS until the MDAS system clears its files in the correct sequence (to be discussed further in the MDAS section). The time an order is placed to the time a finished image is delivered to the FTP directory is less than 10 minutes -- three to four times faster than the original Navigate system.

To retrieve a file from the 120 and place the AVHRR data file onto a designated hard drive takes less than 5 minutes (time is based on retrieving a 130 megabyte file located at the end of a 40 file tape). The longest period of time involves moving the data file to the navigation drive for processing which takes 4 minutes. The current navigation code that is running on the Alpha has been modified for the MDAS system and will navigate the AVHRR data file in less than 60 seconds.

Users may only request 5 navigations per day. Since we have not been able to test the full load scenario on the Alphas, the number of navigations may increase or decrease depending on the usage load placed on the computers. Once a

user has finished his or her requests, the system will automatically build an FTP directory under the users name and place the ordered images from their requests in that directory for the user to pick-up (Figure 8).

The GOES callbacks only allow the viewing of a preprocessed full disk image. The GoesViewCB is similar to the ViewCB used in the AVHRR browse image in that a scrolled list of GOES images are set active by the double clicking of the mouse button and they are viewed in a manner similar to browse (Figure 9). The actual ordering of the GOES images is drastically simplified in that no processes must be actuated and the file is simply moved into the users FTP directory. All of the GOES images are placed on-line on a rotating basis where one weeks worth of data is placed on a single hard drive. The oldest file is replaced by a new file. In this fashion we can keep a full week period on-line at one time.

The programs callbacks again only place a selected file from a scrolled list window into the appropriate FTP directory (Figure 10). Both the GOES and the program's callbacks must track the users name and ordering sequence.

MDAS System

The Meteorological Data Archiving System (MDAS) was developed to track and retrieve the AVHRR files for the navigation process. MDAS is solely written in GNU database software and C coding. GNU simplifies the calls and allows a flexibility in porting the code to other computer platforms. GNU is freeware and is distributed to Internet users for the development of software products.

MDAS relies solely on remote procedure calls (RPC's) where the client Alpha makes calls to the MDAS server Alpha which in turn stores and retrieves files within the 120. The MDAS server, and storage devices may all reside on different machines. The RPC mechanism takes care of the system calls used in network communications and the details associated with the data format

conversions between the Alphas. The RPC calls, for storage and retrieval, are sent from the interface windows application process to the MDAS server where they are queued. The MDAS server then makes its own RPC calls to a device daemon, which runs the driver software to retrieve or place a tape for the file transfer.

Setting up the MDAS system requires starting the MDAS daemon as well as a daemon for the device driver. Both of these daemons run on the server Alpha. A MDAS server / client relationship must be executed on each client computer to have access to the MDAS system. This is formatted by each client knowing the Internet Protocol number of the server and the server identifying which computer is making a call. The MDAS system uses the RPC calls to: `load_new_tape`, `unload_tape`, `load_tape`, `lock_loader`, `unlock_loader`, and `lock_unlock_database`.

If for any reason an error occurs during any process of the MDAS in either a device, or internal error, the MDAS system locks (through the data base) whatever resources it was using (a drive or robotic arm), essentially freezing that part of the system. MDAS then mails the system administrator a message explaining the error and it is then the system administrator 's responsibility to correct the problem and restart the system. Until we gain more experience on the MDAS system, it is better to freeze the system than to try and recover from an unknown error and risk corrupting the database or contents of the tapes.

Both the server and the device daemons also write time-tagged error messages to `stderr` or `stdout`, which is redirected to a file when starting the server and device daemons. Currently , MDAS handles the following errors:

dd errors on a read or write to tape; which could be from a bad tape, a disk being full, a hard disk failure or a drive failure. Second, if a tape is not found in the correct slot. For this to occur, an error was made on the operators part. The data system knows of each tape from the tape table and the only way a tape could not be in the correct slot is for it to be moved outside the control of MDAS. Third, if a

wrong tape header detected. Again this is a human error where the operator either put the right tape in the wrong slot, or the wrong tape in right slot. Fourth, is a loader arm failure. Though MDAS is capable of capturing this error, the 120's robotic arm does not return any type of error message. The MDAS system will continue on the assumption the loader moved the tape as directed, until another error occurs, which will either be the file set error returned by the tape device driver or a dd error. Fifth, the drive fails to rewind and eject a tape. This has not been physically tested but the drive arm should sense not being able to pick or place a tape in the drive. Or finally, an internal database error. This is a very serious error. The scenarios for this case to happen are that the computer has crashed during a database file write or someone accessed a database file while it was being accessed by the MDAS system, causing a read / write conflict. This error is caused by a bug in the GNU database software.

The MDAS system uses the RPC calls to: *load_new_tape*, *unload_tape*, *load_tape*, *lock_loader*, *unlock_loader*, and *lock_unlock_database*. To load a new tape, the call, *load_new_tape*, calls a device daemon procedure which will physically load the tape into the 120, then adds a new tape record to the *tape_table* database. The system manager must first lock the loader before this call can be initiated. There must also be a free slot in the 120 and the tape should be physically placed in the slot before continuing. The new tape call will ask the administrator questions about the tape, such as: What is the tape name. Will it be a read, write, or both. Does the tape contain files? How many files? What is the free space on the tape? And finally, which slot will the tape reside? After the tape has been added the tape table will contain the status of the tape, times mounted, last file position, and tape header.

Load tape is similar to load new tape except that the MDAS system already knows the information about the tape, it just needs to know the new position of the

tape. *Unload_tape* calls a device daemon procedure which will unload a tape from the 120. It will then update the tape record in the tape table database. Unload-tape changes the status of the tape table record from LOADED to NOTLOADED making the MDAS system aware that the tape is no longer loaded in the 120. If calls are made which required the unloaded tape, the systems administrator will receive a message from MDAS telling them to load the unloaded tape so processes can continue.

Lock_loader calls into the MDAS server daemon, and acquires sole use of the 120 robotic arm. It waits to acquire the robotic arm from any requests ahead of this command. Once *Lock_loader* takes control it will create it's own queue table record. The *Unlock_loader* calls back into the MDAS server daemon and remove the queue table record created by *Lock_loader* and signals that the robotic arm is ready for processing. Finally, the *Lock_Unlock_Data_Base* is a function to enable the operator to run the load and dump database executables safely while the MDAS server is running and servicing requests. This locking ensures that only one process is accessing a table at once, so that the situation where one process is writing while the other is reading is prevented. The *Lock_Unlock_Data_Base* calls to the MDAS server daemon and locks or unlocks a database table. *Lock_Unlock_Data_Base* with lock type equal to 1 will block other processes from accessing the table if it is already locked by a server process (or by an earlier call to *Lock_Unlock_Data_Base*). It will unblock once the lock is released, lock the table itself and return. *Lock_Unlock_Data_Base* with lock type will equal 0, will unlock a table regardless of whether it was locked by a previous *Lock_Unlock_Data_Base* or by a server process. Therefore, it is very important never to unlock if you have not locked beforehand (for the same reasons that *Lock_Unlock_Data_Base* works by setting semaphores located in memory which is shared between the server and it's child processes).

The MDAS system is largely a database-driven system. Six database tables (each a binary file) are used for storing static information about the files, tapes, devices and owners, as well as serving as the dynamic request queue, and the synchronization point for processing of a device resources and log history of the steps taken by each request. The first of these data files is the queue table. The queue table deals with only three types of requests: store a file, retrieve a file or lock the robotic arm. The queue table tracks the driver by using two commands: *Using_Drive* and *Using Loader*. Both of these records track the current process and status of the queue table.

The queue table has four waiting records: *Waiting _For_Free_Drive*, *Waiting _For_Free_Loader* (if more than one 120), *Waiting_For_Operator_To_Load_Tape*, and *Waiting_For_Tape*. The "Waiting records" have status's of next requests and are held in the queue table until they are moved to the "Using records". The queue table work with the file and tape tables to decide where a tape should be loaded and, if more than one 120, which arm should be accessed.

The final records are *Copy_File* and *Duplicate_Request*. Both of these are important in that the *Copy_File* works to transfer data from one tape to another. This may happen when copying old files to a new tape and works in the backup dump mode. *Duplicate_Request* saves the MDAS system from going and getting a file from tape more than one time, if it currently exists in the cache. If two users request the same file within or about the same period of time, the MDAS system will flag the *Duplicate_Request* record and will know not to retrieve the file again, but to leave it in the cache directory so it can be processed for the second user.

The *tape_table* deals strictly with the name of the tape, owner, permissions, default device, location of the tape, status, times mounted, amount of free space on the tape, and the last file position. Each tape is listed by a name. It can be a user

name or any type of designated UNIX name. An owner can be assigned to a tape (in our case the data system tapes are owned by the system). As with any file, permissions can be set for a full tape or files on that tape. The default device parameter works by selecting a drive specified for use and the MDAS system will not process that tape until that specific drive becomes free. This work well in the down loading of new AVHRR passes to the MDAS system, since one drive is designated at certain time intervals to upload data. The location of the tape within the 120 is important to the MDAS system for the previously mentioned reasons. Status has also been discussed earlier in whether a tape is LOADED or NOTLOAED. The Times_mounted parameter is important for several reasons. If a tape has been accessed an excessive amount of time, it would be best to copy the data to a new tape. If a tape has not been accessed at all or very seldom it might be best to not store that particular tape in the 120. Both of these functions can be easily accessed through the use of this parameter. The amount of free space on a tape becomes a crucial piece of information for the MDAS system. This tracks how many files are on a tape and how much space is available for further storing capability. For these reasons the last_file_postion parameter is also needed to maximize our storage capability.

The file_table tracks the filename, owner, file size, tape name, permissions, file position, total access, and last access. The file name is a key field. This is what the returned file is named. The owner of a file may or may not be the owner of the tape (example a system tape may contain files from several users). The file size is used to check the dd process as well as capacity of the tape. The tape name parameter contains what tape holds this file and where this tape is located in the 120. Permissions can be set for any file to be read, write, execute or any combination of the above for a variety of users. File position is the number of end of file (EOF) markers to pass on the tape b before reaching the beginning of the

file. The total access time is the total number of times the file has been read, plus one for the original write. And finally, the last access time is the last time the file was read, or the last time it was written to tape if it has not yet been read.

The `owner_table` tracks various administration records such as: the owner, hosts, groups, number of tapes allowed and mailing address. All of the above are set by the system administrator. The owner of the particular file may or may not own the tape that the file resides on. Each user of the MDAS system may be limited to the number of tapes they have access to and the number of tapes they can write to. The mailing address parameter is used for system calls like: the users tapes are full etc.

The `device_table` is used solely for the MDAS system to check for devices such as the number of storage boxes (120) , number of tape drives, set paths. cache path and quota and number of tape slots. This record file will allow for the addition of multiple device drivers, multiple cache directories and allocations of tape slots. The `device_table` is the first record initialized when MDAS starts up.

The `history_table` is a log of all processes, forks, and requests to the MDAS system. It is a running log of all commands and successful or failed requests to the system. It does not control any actions of the MDAS system and is strictly an administrative file. It should be noted that all tables are operational in the binary mode. To edit the files by hand, they must be converted to ASCII and returned to binary format for the MDAS system to operate correctly.

These above database tables are the backbone of the MDAS system. With this information that the tables provide, the MDAS system can retrieve and place files within the 120 on a continuous and correct fashion.

The MDAS system, as previously mentioned, relies on RPC calls to operate the GNU database software. A user interface incorporating all of these procedures is set in the systems manager driver. This interface software will allow the system

manager to read and write to the MDAS system without using the Xwindows interface. (Figure 11).

The MDAS system has the flexibility to add users. Each user can be added in much the same way as adding a new user to the Navigate system. An ID., group, host, and home can all be set. Each user can run his or her own interface to the MDAS system or the system manager driver can be modified to the individual user's needs.

Conclusion

Having completed the Alpha testing and through the start of the Beta testing, Navigate II has shown to work exactly as designed. There will always be critics that site the shortcomings of the original Navigate system. But to date, the storage system has been able to read files off and navigate them in a continuous and timely manner.

The only bugs found in the system throughout the testing period have been minor fixes such as the deselection of a text field which has caused crashes in the user interface. This has been corrected by not allowing a user to deselect any file until they have selected another to take its place. Our largest problem, to date ,is that the OSF operating system will not allow us to use our 8500C drives to write files to tape. We are able to read without error but writing causes core dumps. Digital has been contacted and they are sending out a fix to the SCSI / CAM procedure calls to allow the Alpha to identify the 8500C drives and operate the calls correctly. As soon as this fix arrives the MDAS system will be fully operational. At the time of this paper, Navigate II system can be brought on-line without the 8500 drives being able to write (since all of our data is processed to tape on a separate system and tapes are then loaded into the 120). Users of the Navigate II

system can only acquire processed images. They do not have access to place data on the system.

As proprietary software goes, we are not close to the average 10,000 man hours that go into a commercial software package. However, we have developed a low cost storage system that can be modified to store any type of data in any type of format. Interfaces can be modified or quickly rewritten to drive the MDAS system under any type of scenario. For these reasons it is hard to beat the MDAS system running under Navigate II.

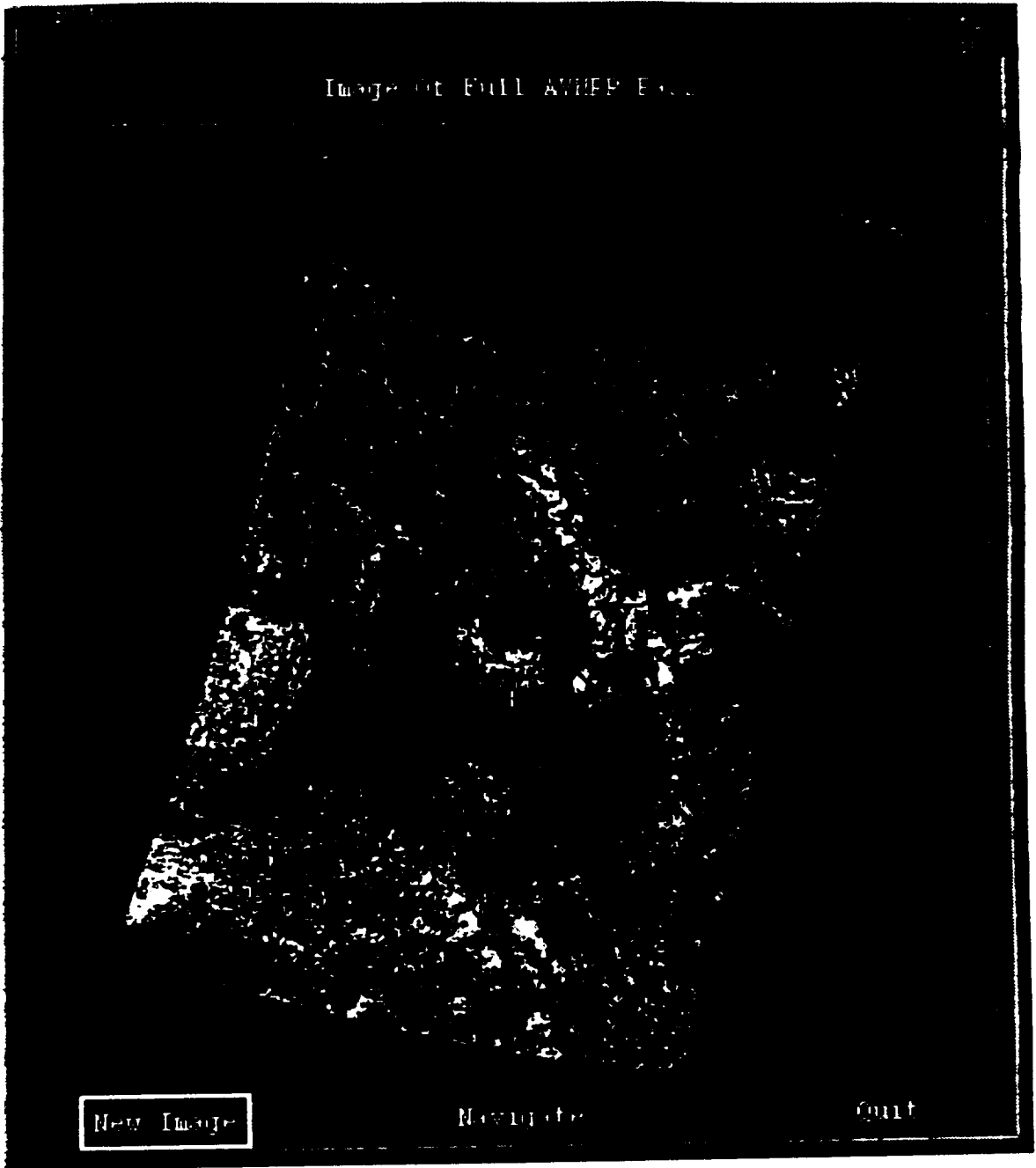


Figure 1. AVHRR Browse Image



Figure 2. Navigated Image Centered Over Colorado

ORIGINAL PAGE IS
OF POOR QUALITY

```

Applications Defaults file for Navorder
Using Wcl library to create the windows

(c) 1994 Tim Kelley

Default usage is often DECstation.
*defaultVirtualBindings: \
  csfBackSpace:      <Key>Delete
Setting the Fonts for the Windows
*FontList:courier

Navorder is the shell widget
Navorder.vcPopups:      acceptDialog, brsList, fileList, goesList, \
                        progList, workingDialog, cityList
Navorder.vcChildren:   mainWin, navWin, goesWin, browseWin, \
                        searchWin

Navorder.title:       Navigate Prototype System

Navorder*foreground:   yellow
Navorder*background:   blue

|
| Main Window For E-mail
|
Navorder*mainWin.vcClassName:  XmMainWindow
Navorder*mainWin.vcChildren:  menuBar, table
Navorder*mainWin.width:      400
Navorder*mainWin.height:     200

|
| Menu Bar
|
Navorder*menuBar.vcConstructor:  XmCreateMenuBar
Navorder*menuBar.vcChildren:     fileMenu, file, help

|
| File Menu Members
|
Navorder*fileMenu.browse.vcClassName:  XmPushButton
Navorder*fileMenu.browse.labelString:  Sat Brovse Pass
Navorder*fileMenu.browse.mnemonic:     B
Navorder*fileMenu.browse.activateCallback:  WcManageCB(*searchWin)
Navorder*fileMenu.browse.sensitive:     False

Navorder*fileMenu.sep2.vcConstructor:  XmCreateSeparator

Navorder*fileMenu.goes.vcClassName:    XmPushButton
Navorder*fileMenu.goes.labelString:    GOES Image
...

|
| Main Table
|
Navorder*table.vcClassName:  XpTable
Navorder*table.vcChildren:  title, acceptButton, clearButton, \
                             email, ebox\
Navorder*table.Layout:     title 0 0 10 1 tE:\
                             acceptButton 1 12 2 1 tE:\
                             clearButton 7 12 2 1 tE:\
                             email 1 6 1 1 hI:\
                             ebox 2 6 7 1 h:
...

```

Figure 3. Applications Defaults File

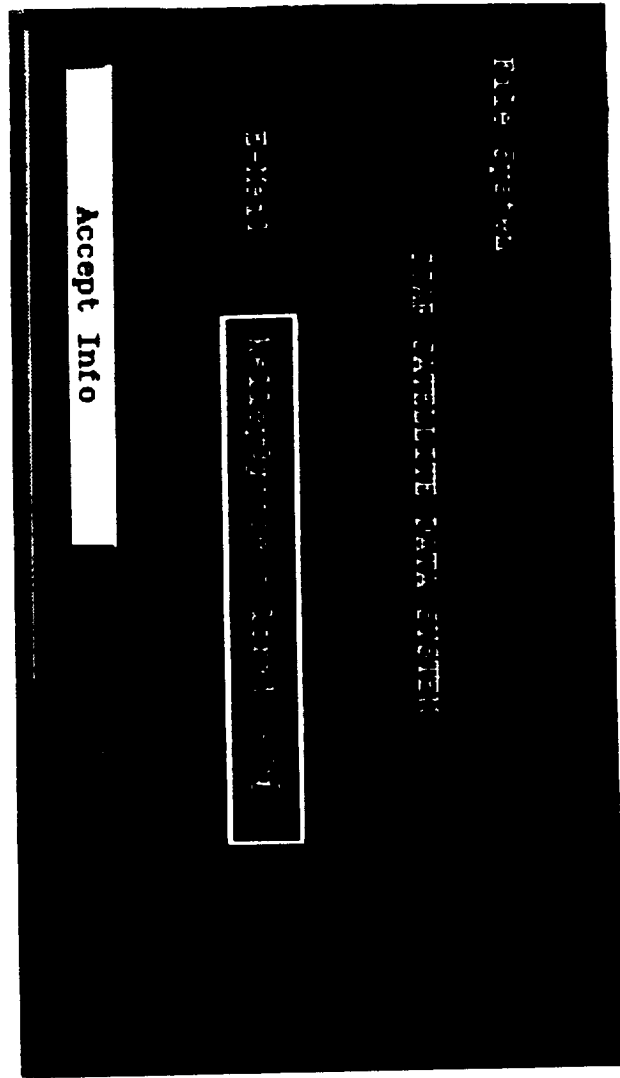


Figure 4. Information Dialog Window

ORIGINAL PAGE IS
OF POOR QUALITY

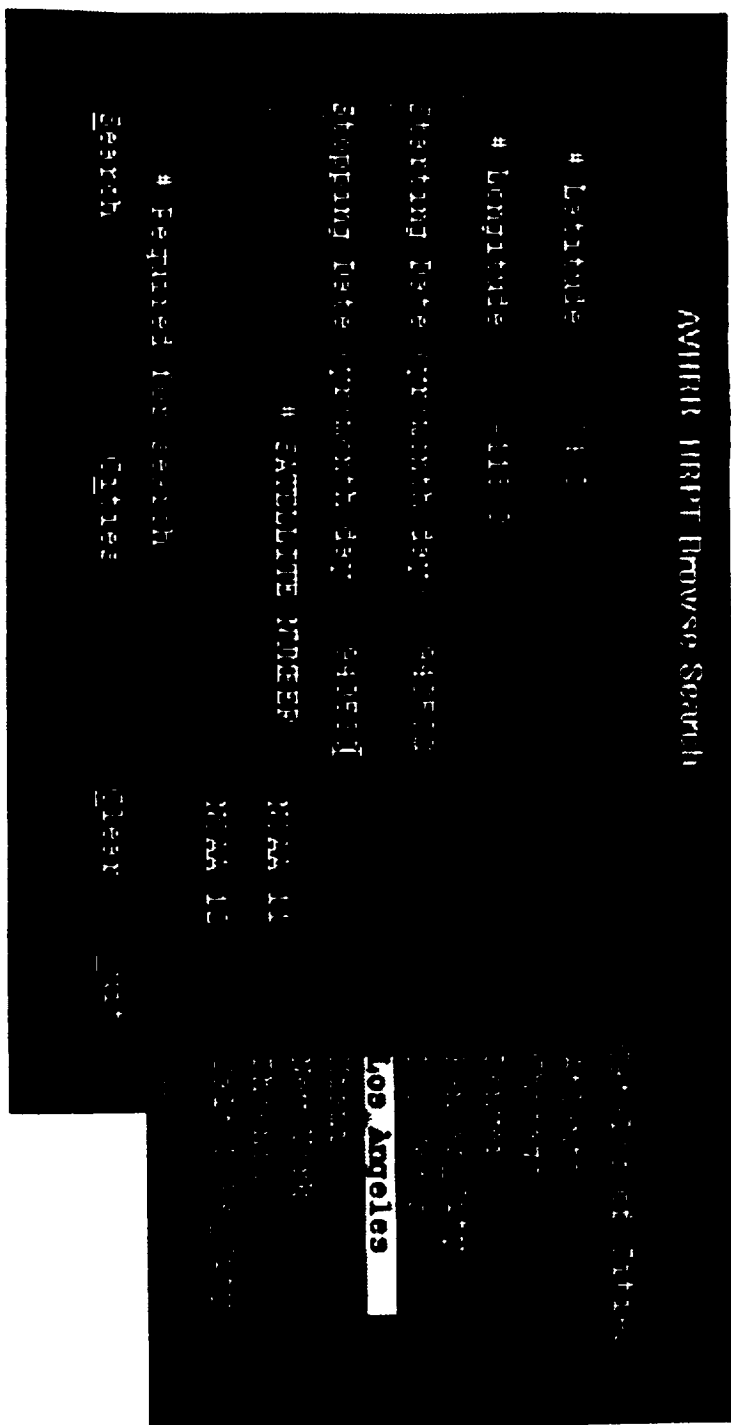


Figure 5. Browse Search Window with Cities List Popup

ORIGINAL PAGE IS OF POOR QUALITY

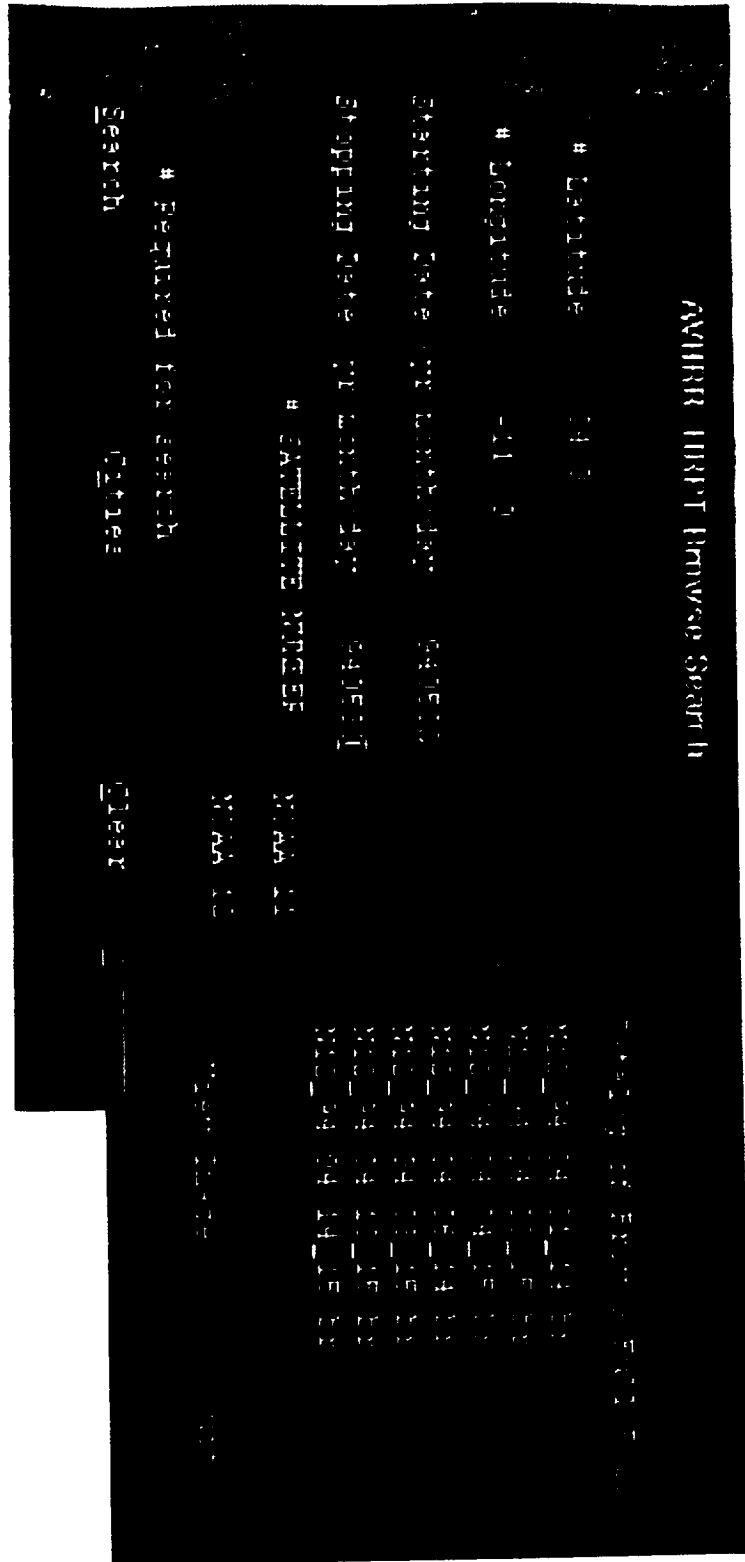


Figure 6. Browse Search Window with Brs List Popup

Navigation by NOAA	
<u>AVHRR Data</u>	MIC_94_04_03_15
Latitude Center Point	34.0
Longitude Center Point	-113.0
Range In Degrees	
Channel	Channel 1 Channel 2 Channel 3 Channel 4 Channel 5
Resolution Value	1 km
Finished Image Size	256 x 256
Projection Type	Cont overlay Map
Options	Smith Angles Spot/Line File Header Off Elevation Map U S 2 Byte Image
Place <u>Order</u>	<u>B</u> rowse <u>R</u> eturn <u>C</u> lear <u>Q</u> uit

Figure 7. Navigate Window

```

from navigate.c
...
/* Data-System additions */
...
if (command.dashs[0] == 'y')
{
/* put together destination dir */
destdir = (char *)calloc(strlen(command.dashd) +
                        strlen(command.dashu) + 5, sizeof(char));
sprintf(destdir, "%s/%s", command.dashd, command.dashu);

/* Make sure destdir exists */
if (stat(destdir, &dirStat) == 0)
{
/* It exists, make sure it's a directory */
if (!(dirStat.st_mode & S_IFDIR))
{
return;
}
}
else
{
if (errno == ENOENT)
{
/* It doesn't exist, create it */
if (mkdir(destdir, 0755) != 0)
{
return;
}
}
else
{
return;
}
}

/* if we made it through all of this, destdir is a directory */
/* copy files to the FTP area */
cmd = (char *)calloc(strlen(command.inageten) +
                    strlen(destdir) + 10, sizeof(char));

if((strlen(command.dashs)) == 0)
    sprintf(cmd, "mv %s.%s %s", command.inageten, destdir);
else
    sprintf(cmd, "mv %s.%s", command.inageten, destdir);

system(cmd);
...

```

Figure 8. FTP Directory Coding

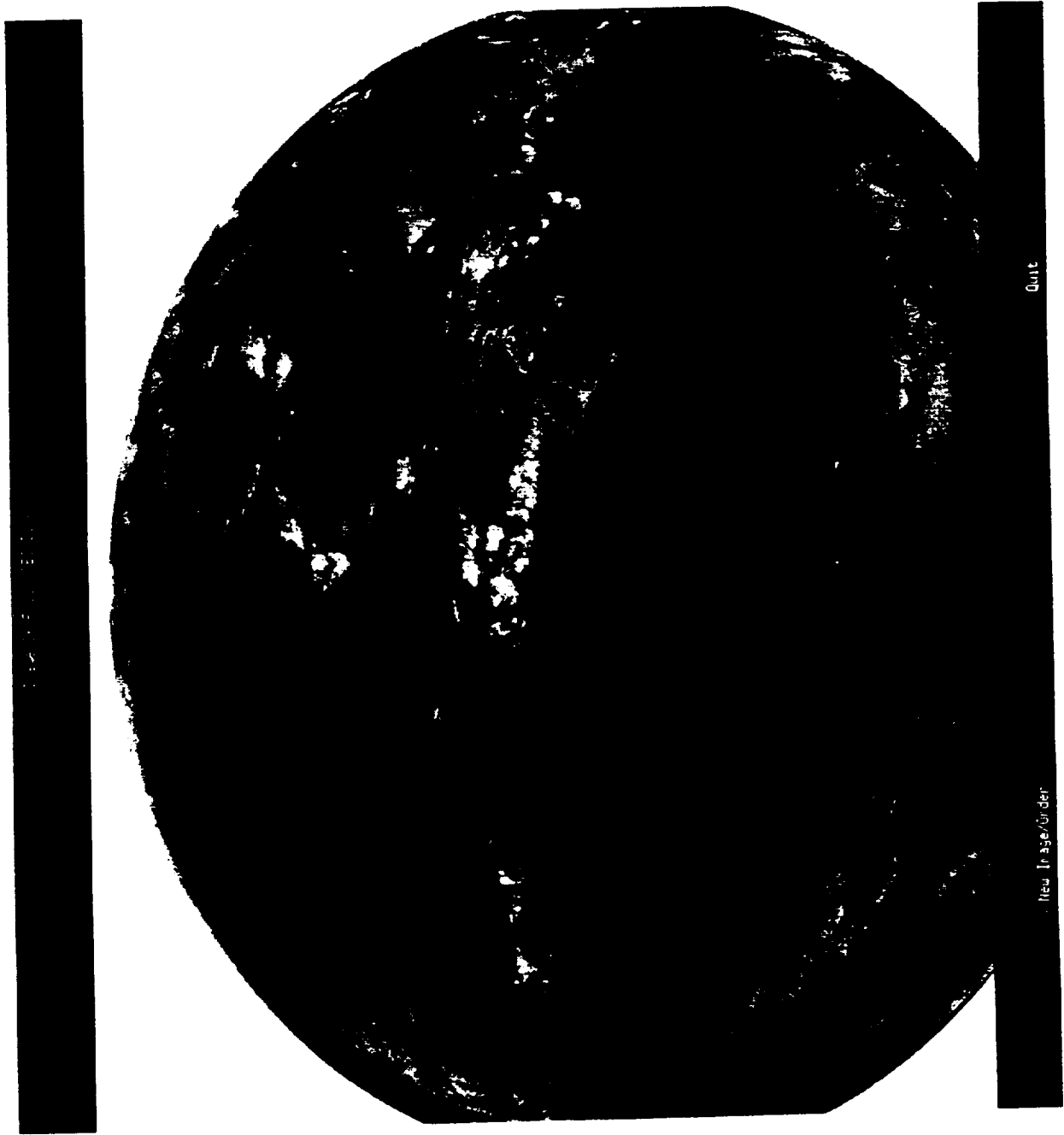


Figure 9. GOES Browse Image

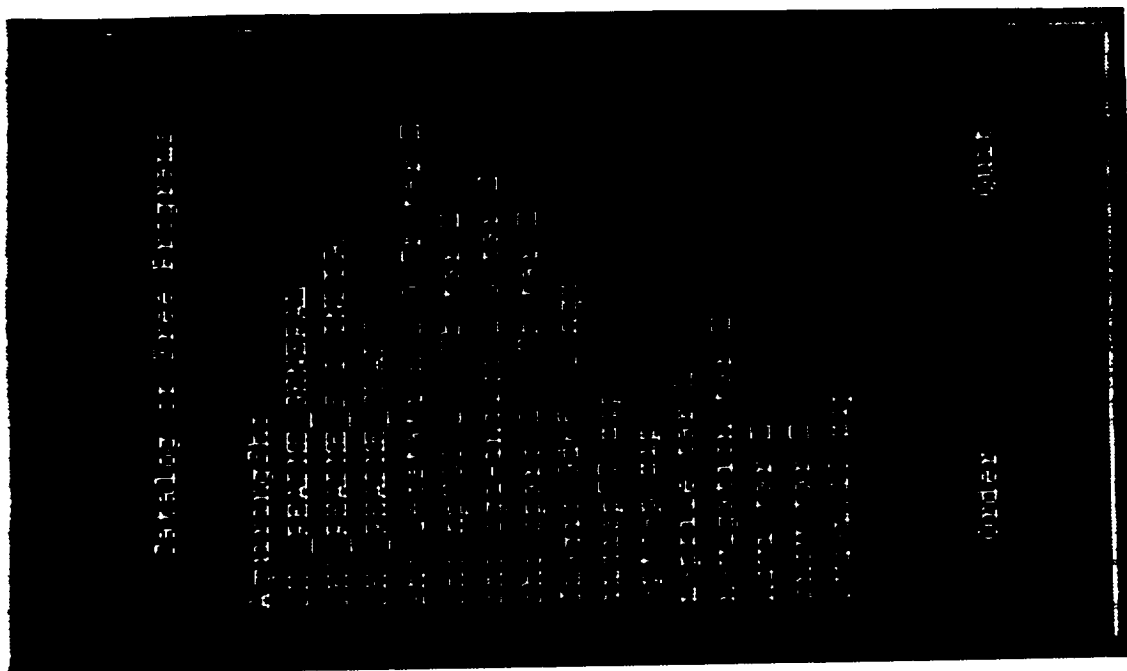


Figure 10. Programs Scrolled List

ORIGINAL PAGE IS
OF POOR QUALITY

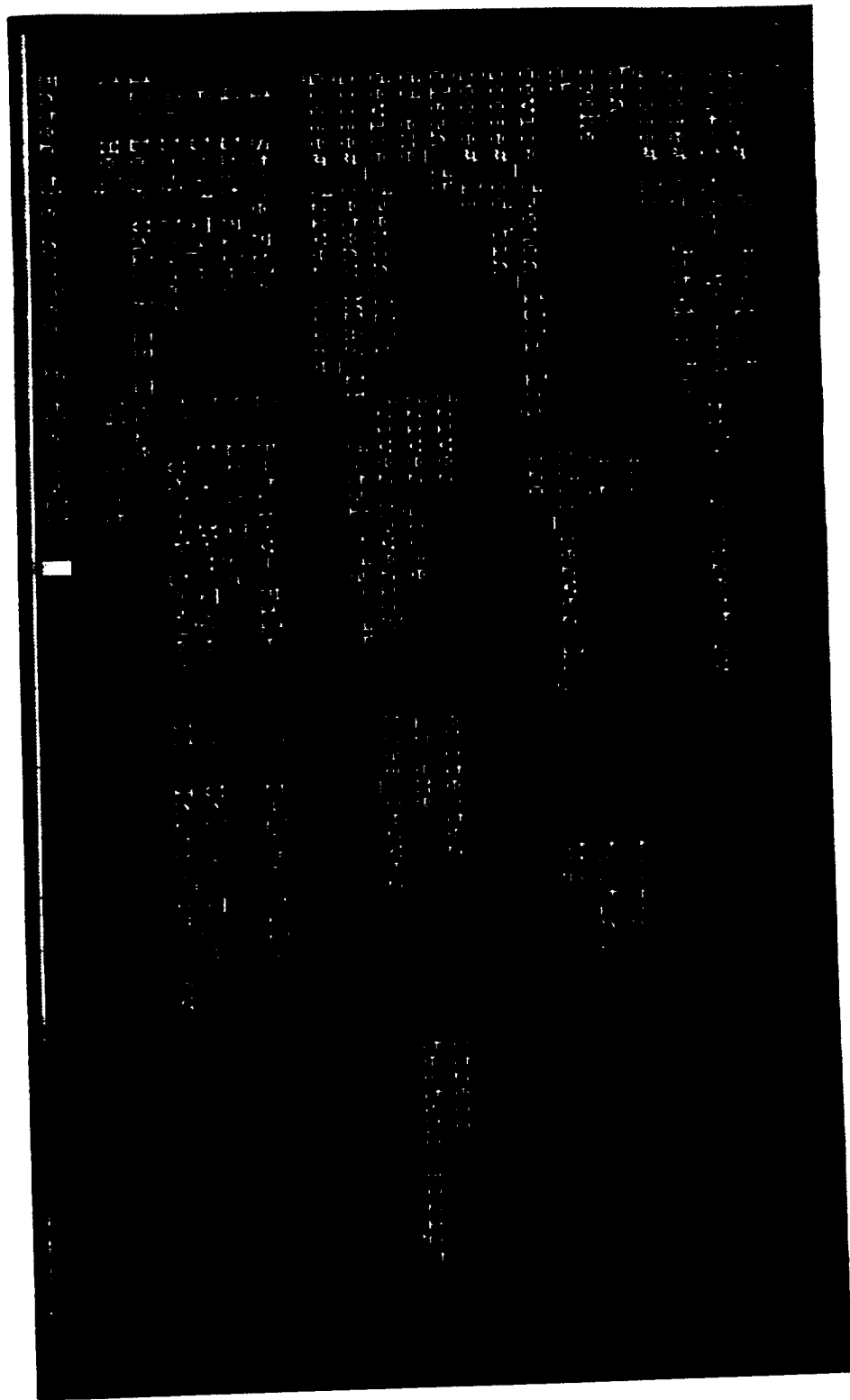


Figure 11. MDAS Interface

ORIGINAL PAGE IS
OF POOR QUALITY

```

void DoSearchCB(Widget w, XtPointer client, XtPointer call)
{
...

/* get the search information */

/* latitude */
buf = XtTextFieldGetString(WcFullNameToWidget(w, "searchWin.table.latText"))
/* convert to floating point */
params->lat = atof(buf);
/* Put latitude text field in navigation window */
XtTextFieldSetString(WcFullNameToWidget(w, "navWin.table.latText"), buf);
/* free buffer space */
XtFree(buf);

...

/* Verify inputted search information */

...

/* Read in the Browse Catalog file */

...

if(file_good)
{
file_cnt = file_cnt + 1;
browseFile = (char *)XtCalloc(strlen(navorderResources.browseDir) +
                             strlen(name2) + 3, sizeof(char));
sprintf(browseFile, "%s/%s", navorderResources.browseDir, name2);

/* check to see if lat/lon coordinate is in this file */
if(coord_check(browseFile, params->lat, params->lon))
{
/* file is good -- output filename to item list */
fileitems[numItemsWrite] = (char *)malloc(SUPPERSIZE * sizeof(char));

strcpy(fileitems[numItemsWrite], name2);
numItemsWrite = numItemsWrite + 1;
}
}

...

/* bring up browse list window */
fprintf(stderr, "bringing up brslist window\n");
XtManageChild(WcFullNameToWidget(w, "brsList"));
searchCatalogCB(WcFullNameToWidget(w, "brsList.scrollList"), client, call);

...

```

Figure 12. DoSearchCB Coding


```

void ViewCB(Widget w, XtPointer client, XtPointer call)
{
...

/* Get name of file selected */
XtVaGetValues(WcFullNameToWidget(w, ""brcList*scrollist"),
              InNselectedItems, &items, NULL);
InStringGetLtor(items[0], InSTRING_DEFAULT_CHARSET, &brsfilename);
path = (char *)XtCalloc(strlen(BROWSEPATH) + strlen(brsfilename) + 2
                      sizeof(char));
sprintf(path, "%s/%s", BROWSEPATH, brsfilename)

/* Open the file */

...

/* Read in the Data */

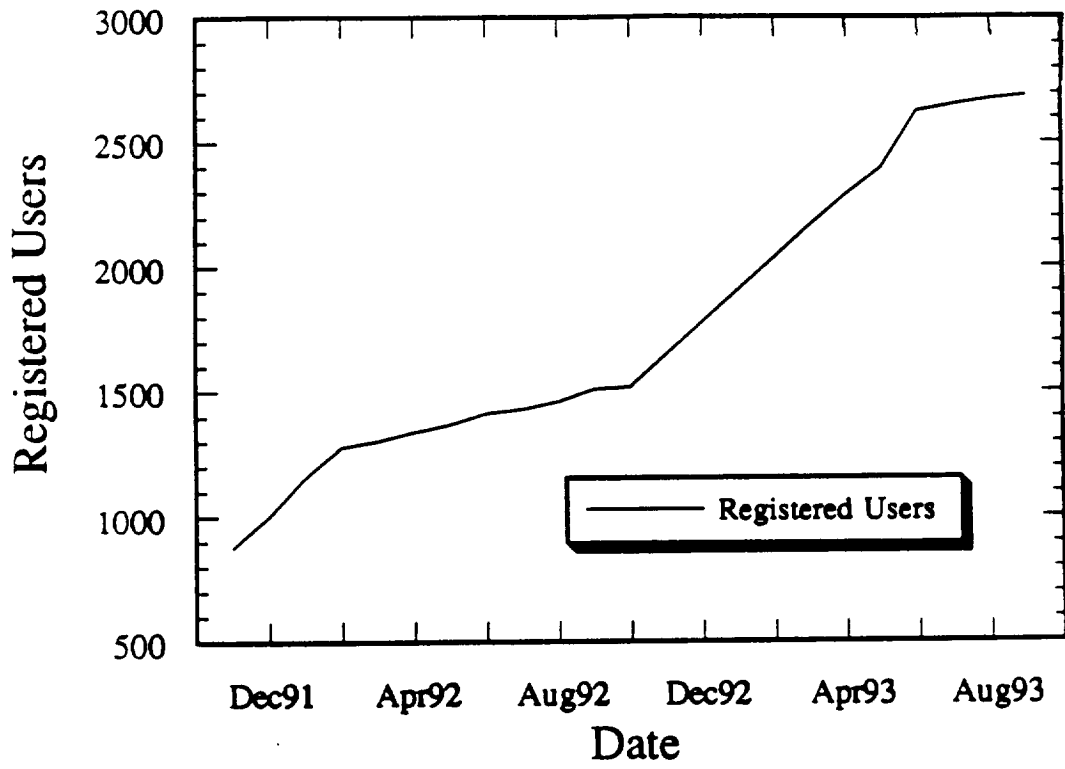
...

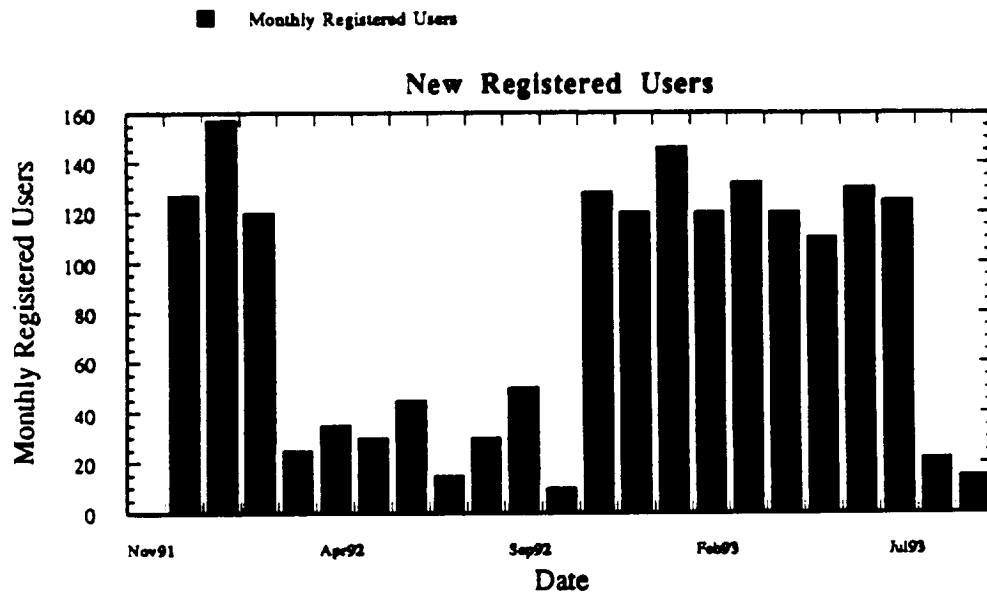
image = WcFullNameToWidget(w, ""browseWin*image");
if(image == NULL) {
/* Create the image widget */
XtVaCreateManagedWidget("image", shoImageWidgetClass,
                          WcFullNameToWidget(w, ""browseWin*brsimageFrame"),
                          XtNimageData, data,
                          XtNimageWidth, width,
                          XtNimageHeight, height,
                          XtNvisualInfo, basicInfo->visualInfo,
                          XtNresPixels, basicInfo->resPixels,
                          XtNnumResPixels, basicInfo->numResPixels,
                          NULL);
XtManageChild(WcFullNameToWidget(w, ""browseWin"));
ViewCB(w, client, call);
} else {
/* Use XtVaSetValues */
XtVaSetValues(image, XtNimageData, data, XtNimageWidth, width,
              XtNimageHeight, height, NULL, 0);
}
...

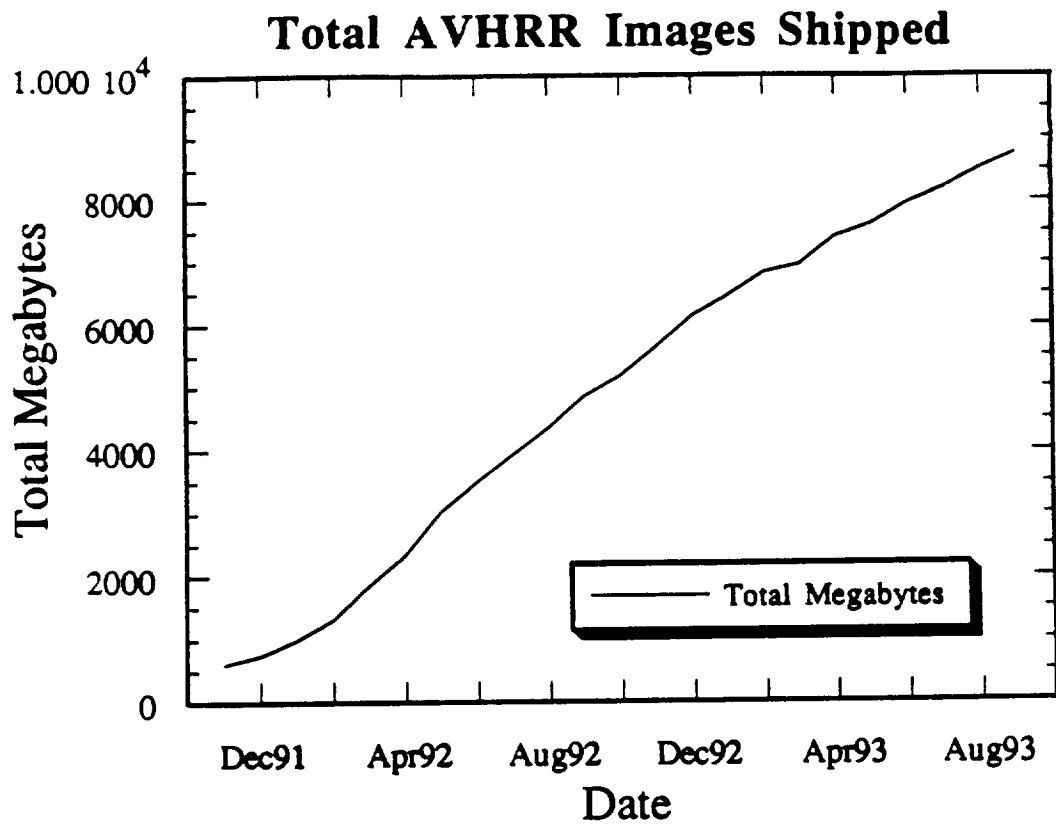
```

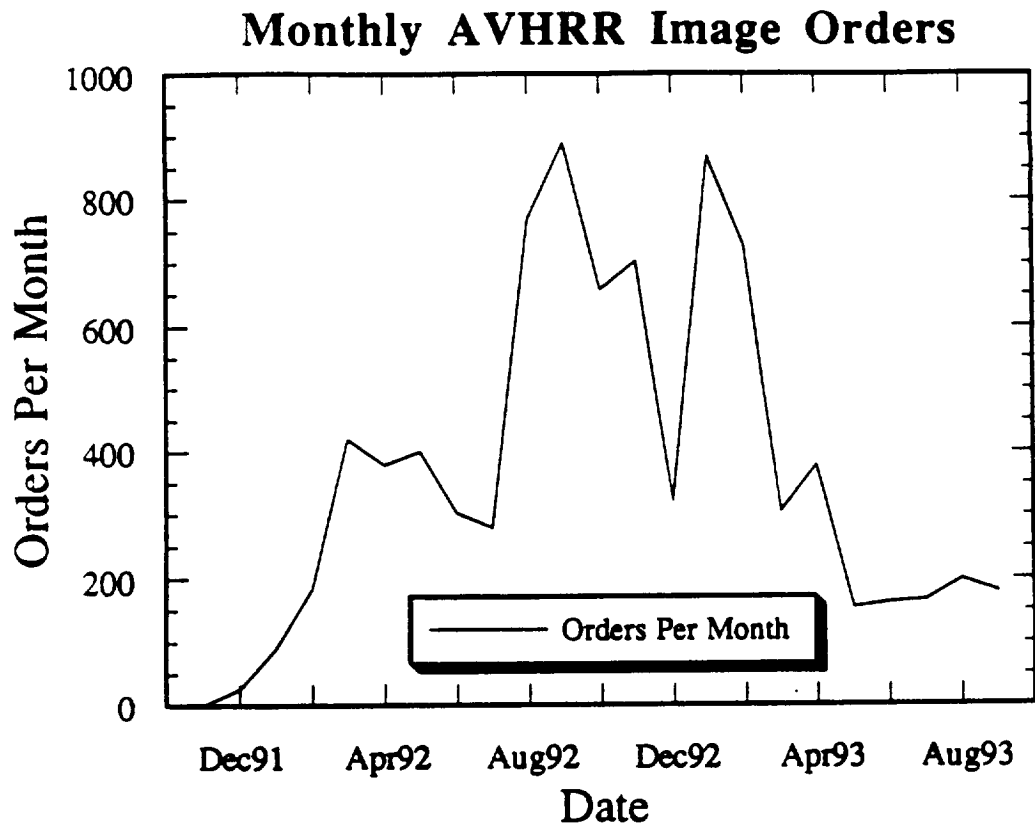
Figure 13. ViewCB Coding

Growth Of Registered Users









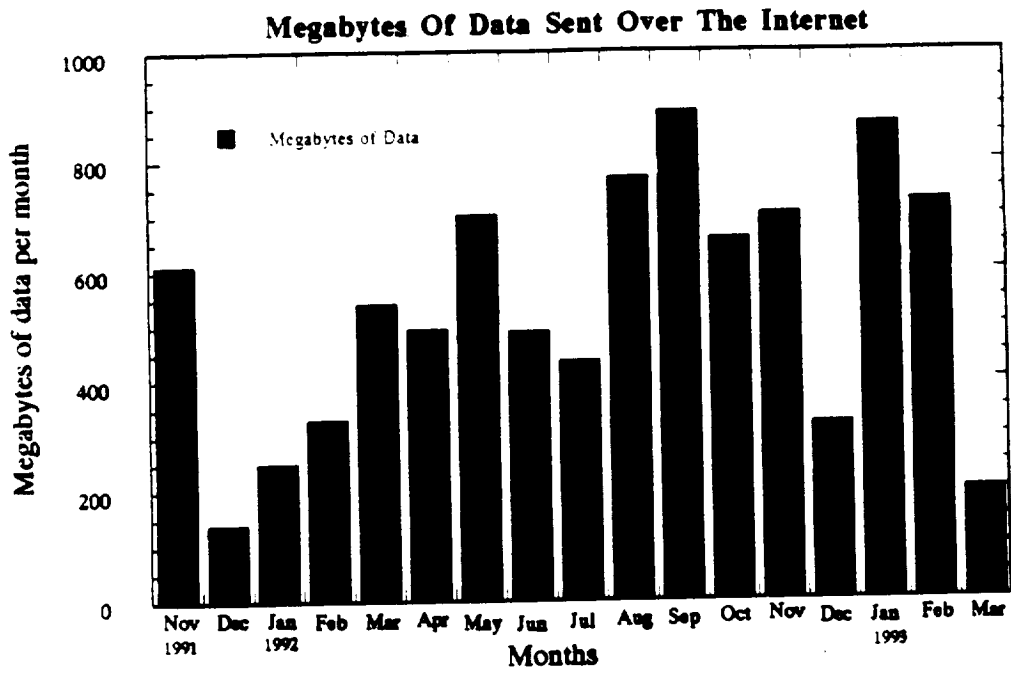


Figure. 1

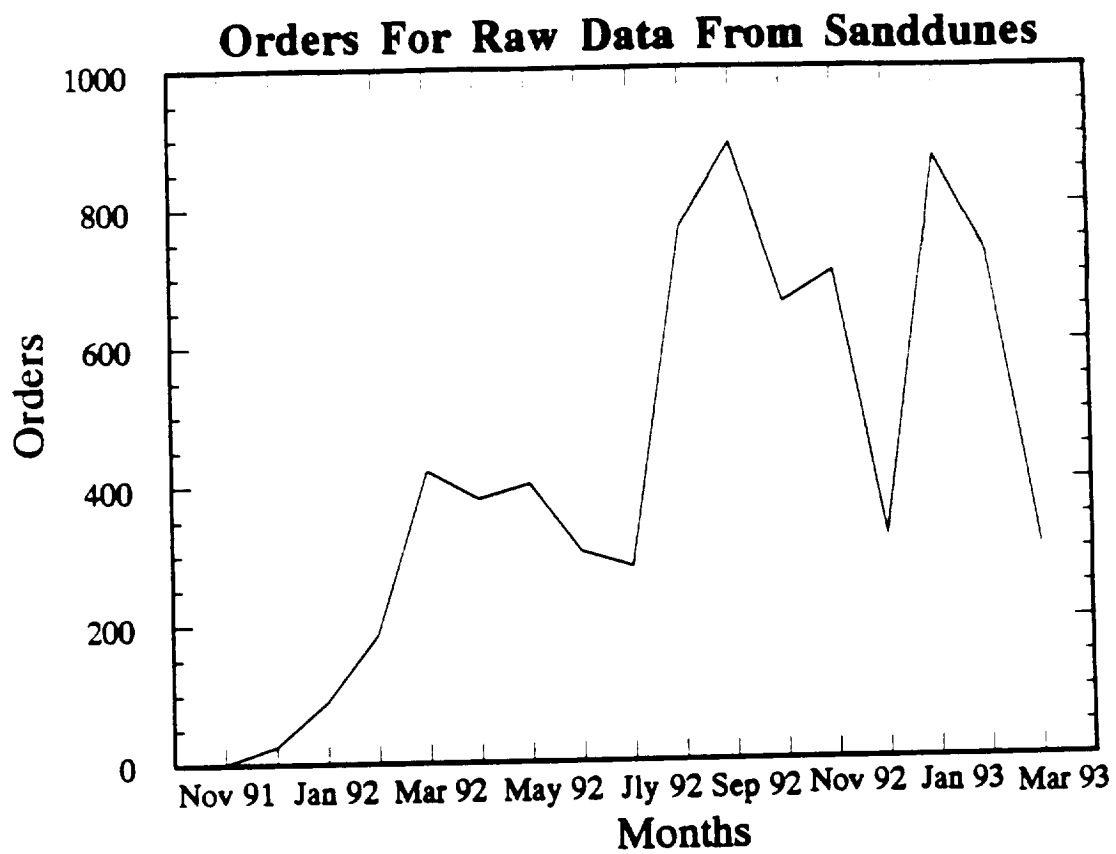
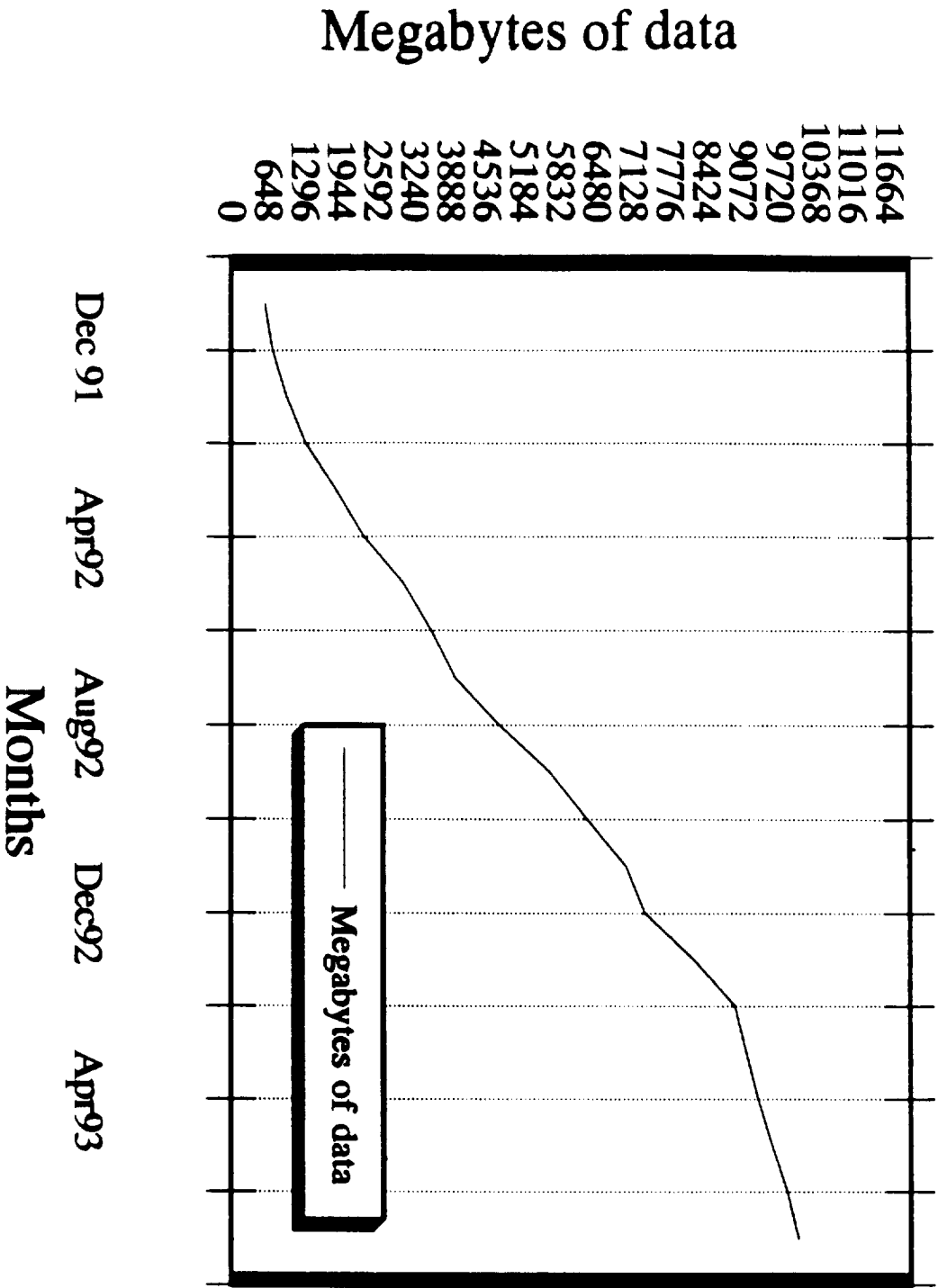
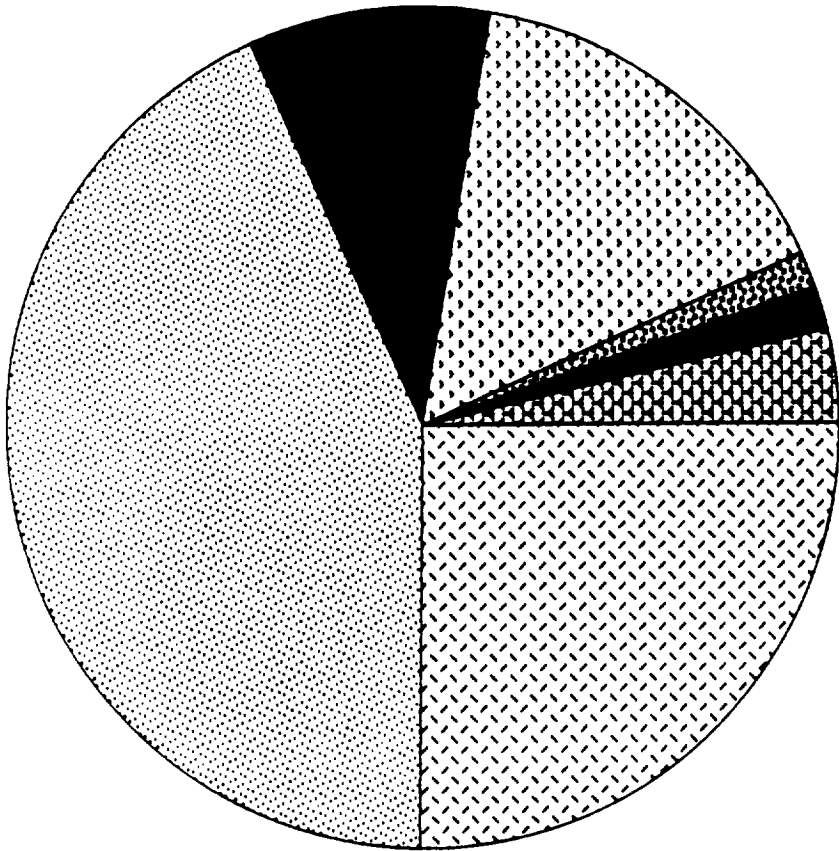


Figure. 3









Total Navigated Images In Megabytes

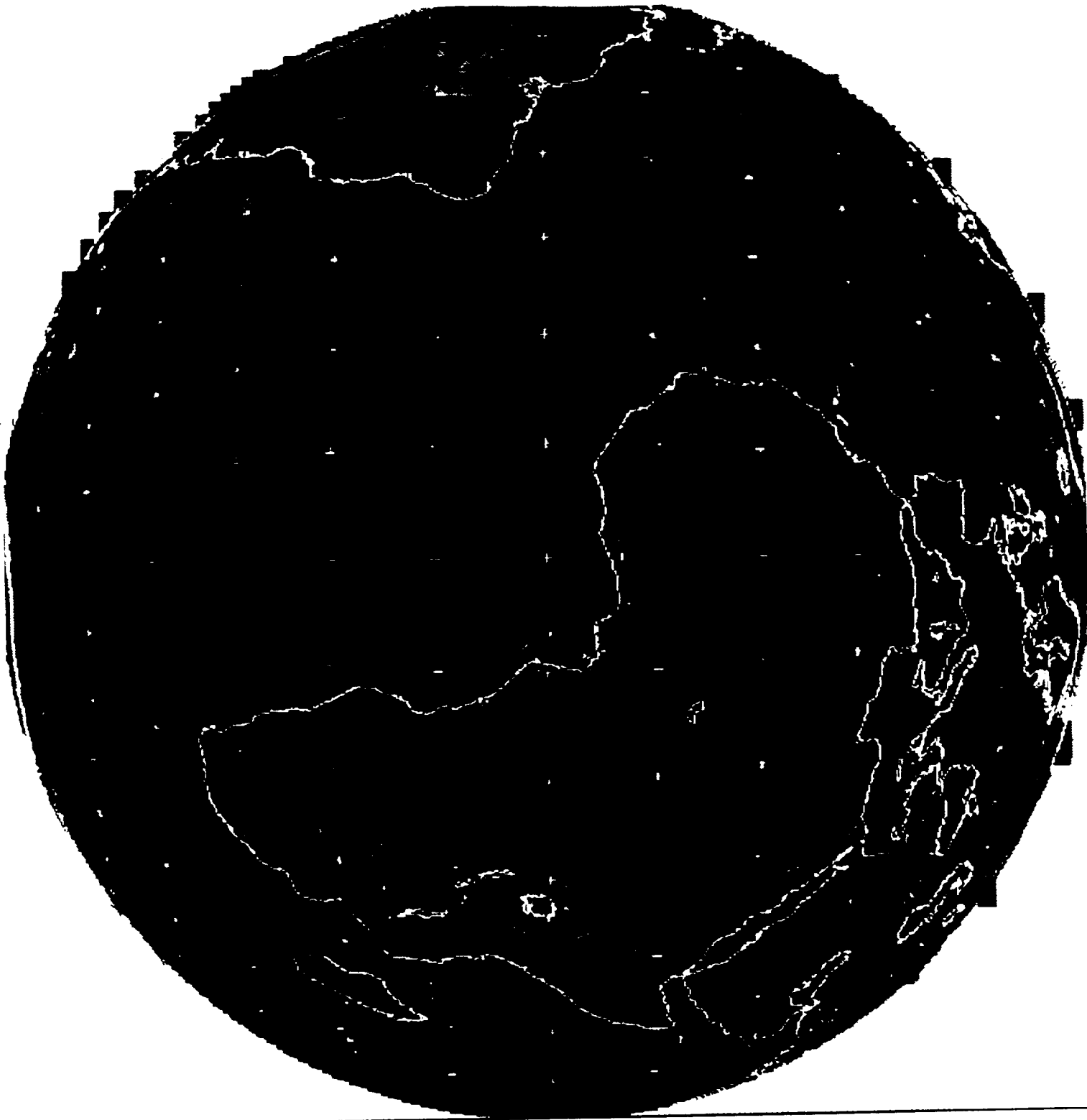


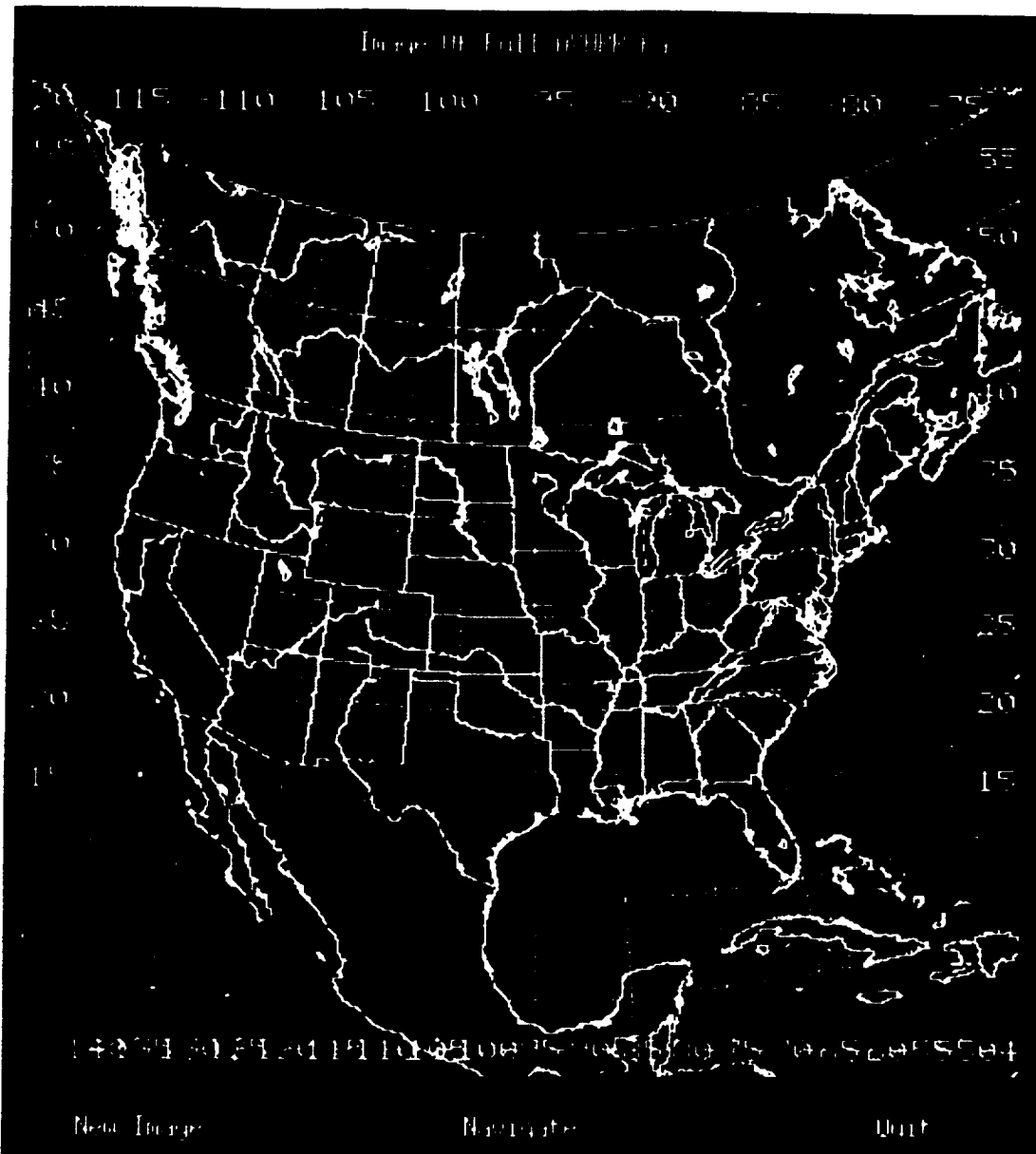
Type Of Users By Category



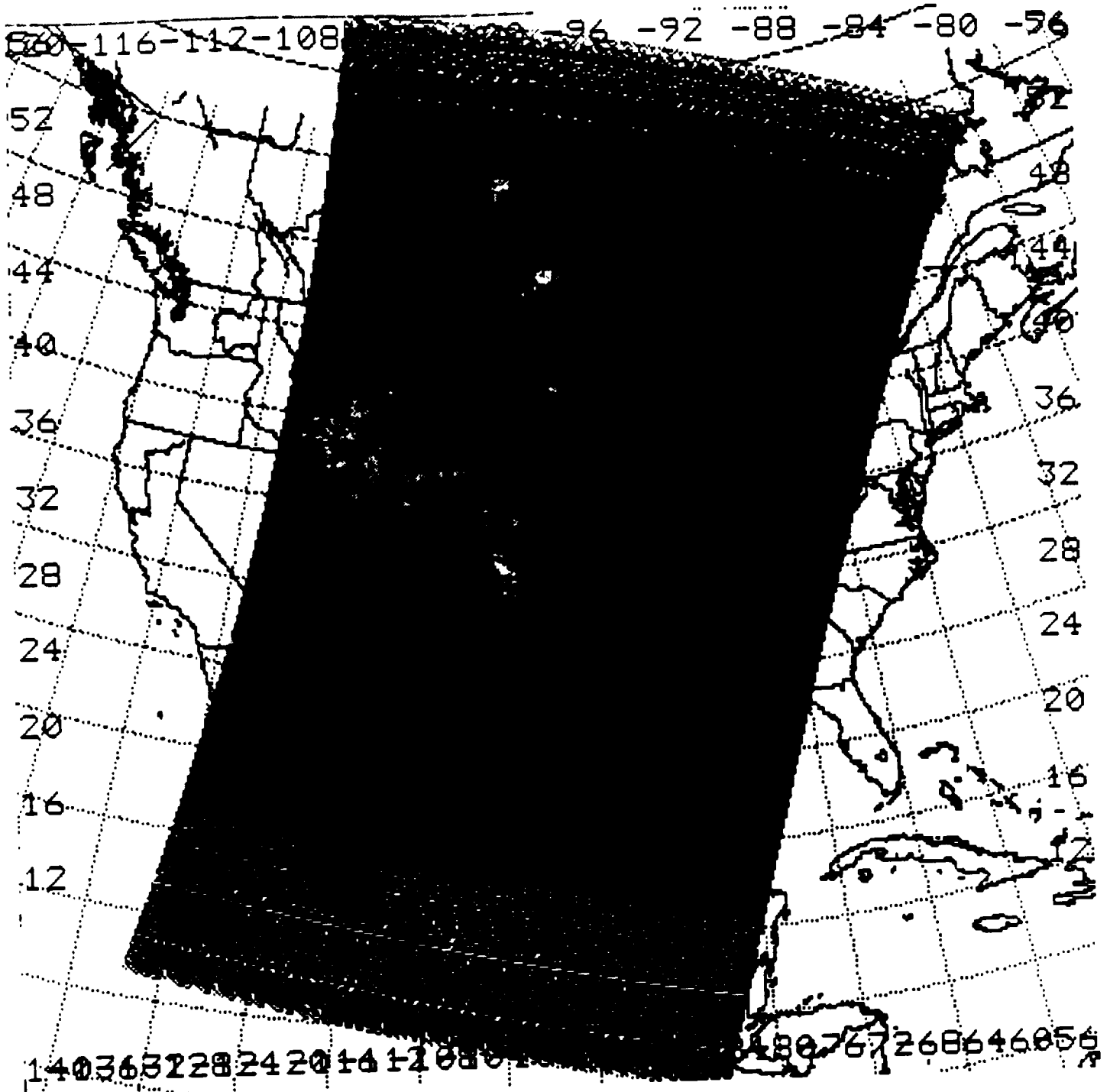
Individual Users

-  Government
-  College
-  Highschool
-  JrHighschool
-  Company
-  Foreign Gov
-  Foreign School
-  NonListed





ORIGINAL PAGE IS
OF POOR QUALITY



ORIGINAL PAGE IS
OF POOR QUALITY





REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 6, 1994	3. REPORT TYPE AND DATES COVERED Contractor Report	
4. TITLE AND SUBTITLE A Land-Surface Testbed for EOSDIS			5. FUNDING NUMBERS 930	
6. AUTHOR(S) William Emery and Tim Kelley				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Colorado CCAR, CB 431 Boulder, CO 80309			8. PERFORMING ORGANIZATION REPORT NUMBER NAS5-32337 5555-07	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration - HQ/ OSSA Washington, D.C. 20546-0001 Universities Space Research Association 10227 Wincopin Circle, Suite 212 Columbia, MD 21044			10. SPONSORING/MONITORING AGENCY REPORT NUMBER CR-189392	
11. SUPPLEMENTARY NOTES Technical Monitor: J. Hollis, Code 930				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified-Unlimited Subject Category 82 Report is available from the NASA Center for AeroSpace Information, 800 Elkridge Landing Road, Linthicum Heights, MD 21090; (301) 621-0390.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The main objective of the Testbed project was to deliver satellite images via the Internet to scientific and educational users free of charge. The main method of operations was to store satellite images on a low cost tape library system. Visually browse the raw satellite data. Access the raw data filed, navigate the imagery--through "C" programming and X-Windows interface software--, and deliver the finished image to the end user over the Internet by means of file transfer protocol methods. The conclusion of this project is that the distribution of satellite imagery by means of the Internet, is feasible, and the archiving of large data sets can be accomplished with low cost storage systems allowing multiple users.				
14. SUBJECT TERMS Computerized Archiving, AVHRR, Navigation			15. NUMBER OF PAGES 53	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited	