Permission is granted to make and distribute verbatim copies of this manual, provided the copyright notice and this permission are preserved on all copies. SSEC makes no warranty of any kind with regard to the software or accompanying documentation, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. SSEC does not indemnify any infringement of copyright, patent, or trademark through use or modification of this software. Mention of any commercial company or product in this document does not constitute an endorsement by SSEC. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this manual, and SSEC was aware of the trademark claim, the designations are printed in caps or initial caps. The information in this manual is subject to change without notice. Considerable effort has been expended to make this document accurate and complete, but SSEC cannot assume responsibility for inaccuracies, omissions, manufacturers' claims or their representations.
# Table of Contents

Welcome to McIDAS Developer/Operating Training ................. iii

Session 1. The McIDAS Programming Environment ............. 1-1

Session 2. Applications Development in the ADDE ............... 2-1

Session 3. Development Environment for McIDAS-X and -OS2 ... 3-1

Session 4. Writing GUIs for McIDAS using Tcl/Tk ............... 4-1

Session 5. McIDAS Navigation and Calibration Subsystems ...... 5-1

Session 6. Designing and Implementing Calibration Modules .... 6-1

Session 7. Designing and Implementing Navigation Modules ...... 7-1

Session 8. Developing Local Decoders in McIDAS-XCD .......... 8-1

Session 9. Moving to a Distributed System ...................... 9-1

Session 10. McIDAS-XSD Operations .......................... 10-1

Session 11. McIDAS-XCD Operations .......................... 11-1

Session 12. McIDAS Operations on a Distributed System ........ 12-1
Table of Contents
Welcome to McIDAS Operator/Developer Training

Welcome to the 1995 McIDAS Developer/Operator Training sessions. SSEC is providing this training to give you the information you need to develop and support locally-created software in your McIDAS environment.

In the next 2½ days, the McIDAS training staff will do the following:

- teach you how to write applications using the new Applications Program Interfaces (APIs) for argument and data fetching
- explain the new McIDAS paradigm of the client/server relationship
- demonstrate how to write client applications and data servers
- give you instructions for setting up a McIDAS development arena
- tell you how to interface McIDAS applications with a Graphical User Interface (GUI)
- show you how to write calibration and navigation modules for local data sources and geographic projections
- introduce you to the processes and subsystems included in the McIDAS-XCD package
- explain some of the processes and procedures associated with the administration of a distributed operational environment
- give you hands-on experience
About this training manual

This training manual provides detailed information about the content of each talk. You shouldn’t need to take notes.

To follow along in the manual as the trainers present their talks, look for the small number printed in the lower-left corner of their slides. This number references a page or group of pages in the manual.

Throughout this manual, references are occasionally made to lines of source code included with a training session. These line number references are shown as bold characters surrounded by brackets; for example: [A101-A110].
Hands-on exercises

The hands-on exercises are designed to give you practical experience with the information discussed during the talks. You will work in pairs on these exercises. Each xterminal in the training room has an account with the password mug95trn1. Each account is set up with a typical McIDAS configuration, as shown below.

Environment variables:
PATH=~/mcidas/bin:~/mcidas/bin, plus necessary compiler directories
MCPATH=$HOME/mcidas/data:~/trainer/mcidas/data:
                ~/mcidas/data:~/mcidas/data:~/mcidas/help

Directories:
$HOME/mcidas
$HOME/mcidas/bin
$HOME/mcidas/data
$HOME/mcidas/help
$HOME/mcidas/lib
$HOME/mcidas/src

You will write sections of code necessary for:

• an ADDE data server
• an ADDE client application
• a graphical user interface for the ADDE client application

The ADDE data server

Your first task will be to write a section of code for an ADDE data server to deliver MRF gridded fields to a client application as image data. The data set provided contains gridded temperature data at 1000 mb from the Medium Range Forecast Model (MRF) in ASCII format. The data is in the directory ~trainer/mcidas/data in the files MRF1000Tnn where nn is the forecast hour of the grids. The first line contains geographic information about the data. The second line contains the valid forecast day and time of the data. The third line contains the filing format used for the data and the units. The remainder of the file contains the actual data oriented in a row-major format, so the first data point occurs at the North Pole and the dateline, and moves eastward around the globe. Subsequent lines move southward, ending at the South Pole. Your task will be to write a portion of the data server that delivers a line of data to the client.
The source code for the server exercise is in $(HOME)/mcidas/src in the files below.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subserv.c</td>
<td>main to all servers</td>
</tr>
<tr>
<td>mugadir.c</td>
<td>directory server</td>
</tr>
<tr>
<td>mugaget.c</td>
<td>data server</td>
</tr>
<tr>
<td>mcmugutil.c</td>
<td>specific functions for training</td>
</tr>
<tr>
<td>mservutil.c</td>
<td>generic functions for training</td>
</tr>
</tbody>
</table>

Due to time constraints, you will only write the function **ReadMugLine**.
This function reads a line of data from the file. The interface for this function is shown below.

```c
int ReadMugLine (char *src_file, READPARM *read, int band, short *buf, char *err)
```

**Input:**
- `src_file`: ascii file containing source data
- `read`: READPARM struct containing read specifications
- `band`: band number to read

**Output:**
- `buf`: buffer containing image data
- `err`: error string returned when a failure occurs

The READPARM structure contains specifications that may be needed in the function.

```c
typedef struct READPARM {
    char src_type[4]; /* source data type: GVAR, MSAT, etc. */
    char des_unit[4]; /* destination units: RAW, BRIT, etc... */
    char src_unit[4]; /* source units: RAW, BRIT, etc... */
    int begele; /* beginning element */
    int beglin; /* beginning line */
    int bufsiz; /* size of the buffer to read */
    int des_len; /* destination byte length of one data point */
    int elem_res; /* resolution in element direction */
    int line_res; /* resolution in line direction */
    int maxele; /* last element in image */
    int maxlin; /* last line in image */
    int minele; /* first element in image */
    int minlin; /* first line in image */
    int numband; /* number of bands in the image */
    int numele; /* number of elements to read */
    int numlin; /* number of lines to read */
    int src_len; /* source byte length of one data point */
    int ul_elem; /* element in upper left corner of image */
    int ul_line; /* line in upper left corner of image */
} READPARM;
```
The ADDE client application

Your second task will be to write portions of a client application, MUGAREA, that takes data from two forecast time periods, performs simple math operations on the data, and files the resulting values in a McIDAS Area file. You can then display the created destination file with the McIDAS ADDE command IMGDISP.

You will add the necessary code to perform the following tasks in mugarea.pgm:

- read from the command line:
  - the destination dataset name and position location and separate the two values
  - the number of lines and elements to request from the server
- append the values for the number of lines and elements to the sort strings
- insert mcaget calls to get the data from the two source files
- get the nav and cal blocks for the source data
- start the server transaction to write the data
- read in the lines of data and write the results to the destination data
- write out comment cards

The user interface for MUGAREA will look like this when complete:

MUGAREA source pos1 operation pos2 destination <Keywords>

Parameters:
- source: source dataset name containing MRF grids
- operation: mathematical operations: ADD, SUB, AVG
- pos2: position of 2nd grid to use (default=second most recent)
- destination: destination dataset and position (no default)

Keywords:
- SIZE=nlines ncolumns size of area to get from server

This completes the exercises scheduled for the first day.
On the second day, you will add the following sections to the GUI code:

- the message text to the beginning of the application
- option widgets for preparing the call to the client application MUGAREA
- the call to the client application MUGAREA
- the online help to the GUI

We hope you find these sessions rewarding and educational. If there is anything we can do for you during your participation in these training sessions, please let us know.
The McIDAS Programming Environment

Presented by
Tom Whittaker
McIDAS Development Team Manager

Session 1
McIDAS Developer/Operator Training
October 23-25, 1995
Table of Contents

Overview .................................................. 1-1
  History .............................................. 1-1
  Our goals ........................................... 1-2

The McIDAS library ...................................... 1-4
  Conventions ........................................ 1-4
  Generic data fetchers ........................... 1-8

Command parameter/argument fetching .................. 1-10

Sample code ............................................. 1-12

mccmd functions ...................................... 1-14
  General argument fetching utilities ............ 1-14
  Character strings ................................ 1-16
  Numeric ............................................. 1-17
  Time ............................................... 1-18
  Date ............................................... 1-19
  Latitude/longitude (or other angles) ........... 1-20

mcarg functions ...................................... 1-21
  Internal use functions .......................... 1-22
  Useful utilities that replace older functions 1-24

Argument fetching status ............................ 1-26
Overview

This section discusses the software components of McIDAS-X and McIDAS-OS2. First, it introduces you to the historical perspective by describing the evolution of the software development. Then it describes our goals for developing applications in McIDAS. Finally, it provides the information you need for developing applications, including McIDAS library conventions and command parameter/argument fetching.

History

McIDAS was originally developed on a Raytheon-440 computer using punch cards, paper tape and magnetic tape. The second phase moved the code to Harris minicomputers in a distributed network, with two data servers and several applications boxes. The system was then centralized onto an IBM mainframe, which became the McIDAS-MVS system.

Shortly thereafter, the first smart workstation was developed on DOS-based personal computers. These machines front-ended display hardware, like the Tower. There were very few local applications, since the multitasking required had to be simulated in the McIDAS software.

The first large-scale port of applications software came when the OS/2 operating system was embraced for PCs. Our efforts to create an environment similar to the mainframe resulted in a relatively easy port of many applications. Soon, however, we learned that modifications were necessary because of special hardware; for example, VGA displays had only 16 color/gray levels. In addition, we needed to write drivers for each display head (VGA, Tower, WIDE WORD, SDA) and make them appear to the applications as the same kind of raster-oriented devices, with some varying characteristics (frame size, number of colors, etc.). At the same time, we needed communications drivers for the common modes (asynchronous, ProNET, TCP/IP) that satisfied all our applications.

The success of McIDAS-OS2 led us to consider migrating to the Unix environment. Sites were requesting support for the applications on these faster, larger hardware platforms. Our first attempt to accomplish this was to take the OS/2 code and write specialized routines for keyboard/mouse, text display, and image/graphics display, plus the system-level interfaces required for communications and disk I/O. McIDAS-X was born. The implementation was done completely using the X Windows system, except for the ASK command and the Graphical User Interface. Early plans to support the WIDE WORD display head from the micro-channel in RS/6000 machines has been dropped.
Moving to support McIDAS-X on more platforms made us increasingly sensitive to industry standards. We have significantly changed the base code to make it more portable and less platform- and vendor-dependent.

Our goals

Last year, we embarked on a project to reconnect McIDAS to Unix. Work on this project began early in 1995 and will be completed by the end of the year. Here are some its goals:

Move the low level code to C
This move will help eliminate infinities, such as command line length, in the code for such things as array dimensions. C is also generally easier to interface with essential operating system functions.

Create more code that is common between OS/2 and Unix
Changes in I/O, for example, let us use the same libibio routines in Unix and OS/2 versions of McIDAS, producing more efficient maintenance.

Modify the implementation of User Common
User Common will remain available only as long as needed; in some cases, only for the duration of a single command execution.

Rework the image/graphic window
At present, all McIDAS-X routines link with the X Windows library; those that write images and graphics do so by direct calls to the X Windows system. The rework includes the following changes:

- The applications can write display output to a shared memory frame object.
- The display routine reads from the frame object and displays as needed.
- The colorizing algorithms are more flexible.
- Frame objects can have multiple windows.
- Windows can be zoomed, roamed and resized.
- The number of color levels can go beyond the current limit of 128.
Rework the status and command windows

Our survey showed that having these separate windows caused more problems for users than any other single user interface issue. To correct this and also reduce clutter on the desktop, the command and text windows will be combined, much like McIDAS-OS2.

Allow applications to run from the Unix shell

This change will enable automated processing. For example, programs may be started from the Unix crontab; McIDAS commands will no longer require a McIDAS X Windows environment in order to run. These two modes will be available:

- run a single command, making a User Common and McIDAS environment as needed
- create a McIDAS environment, with or without the image/graphics and text/command windows, and allow several commands to run
The McIDAS library

To develop applications in McIDAS, you must understand its library conventions. This section explains those conventions, and provides information about the generic data and argument fetchers.

Conventions

Functions

Although the interface to old functions did not change, some functions were rewritten and some functionality was replaced. For example, reading and writing data is now done with the ADDE functions; command line parameter fetching is done with new argument fetchers.

New functions have a unique prefix:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>C-callable, API level</td>
</tr>
<tr>
<td>m</td>
<td>Fortran-callable, API level</td>
</tr>
<tr>
<td>M0</td>
<td>C-callable, non-API</td>
</tr>
<tr>
<td>m0</td>
<td>Fortran-callable, non-API</td>
</tr>
</tbody>
</table>

Additional functions for reading and writing data using the ADDE model are being written. Image and grids are currently supported; point source and text are in development. All new functions employ the new argument fetchers.

Argument fetchers

The new argument fetchers, released in the June 1995 upgrade, are functional equivalents to all current functions, except mclex, which has no analogy. These argument fetchers do the following:

- remove the 12-character limit on command line arguments
- provide convenient range checking on parameter values
- consolidate functions
- provide more systematic error messages
Data fetching APIs

New functions were created for image and grid point data using new APIs to implement the ADDE (Abstract Data Distribution Environment) client/server approach to reading and writing datasets. We are currently creating new applications to functionally replace existing ones in these areas, and are developing APIs for point source and textual data.

The new APIs treat the data in a more abstract manner. For example, requests to read data use a descriptive list to define the appropriate subset of the dataset that the server will return to the client. The data returned to the client is in a defined, internal format, irrespective of the actual file format of the data. For example, a server written to read grid point data may return the data in the form of an image.

Coding in C

Most new library functions are coded in C, with Fortran interfaces provided as needed. There are two considerations here:

- The ordering of elements in multi-dimensional arrays in Fortran and C are different.
- The strings of characters between Fortran and C are different.

Using C provides the opportunity to eliminate the hard-dimensioned arrays that Fortran requires. Below is an example of cross-language interfacing.

From Fortran, calling a Fortran interface:

```c
CALL SDEST('THIS IS A TEST', 99)
```

From C, calling a C-coded interface looks much the same:

```c
(void) sdest("this is a test", 99);
```

From C, calling a Fortran-coded interface looks different. First, the name of the function must end with an underscore character, for example: `sdest`. Second, the Fortran convention is to pass arguments by address; C passes by value. Finally, Fortran appends an extra hidden argument for each character string passed to a function containing the length of the string.

```c
char message[] = "THIS IS A TEST";
int value = 99;

(void) sdest_(message, &value, strlen(message));
```
New utilities

New library utilities are continually being added to the McIDAS library. With the June 1995 upgrade, functions were added for converting strings to time, date, and angle. A new units conversion function was also released. We plan to expand upon this set with each upgrade.

Interface documentation block

All new software contains a standard, formatted Interface Documentation Block that provides complete interface information with the code. The components of this block include the following:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>name and short description</td>
</tr>
<tr>
<td>Interface</td>
<td>details of the interface/call sequence</td>
</tr>
<tr>
<td>Input</td>
<td>input variables</td>
</tr>
<tr>
<td>Input/Output</td>
<td>variables used for both input and output</td>
</tr>
<tr>
<td>Output</td>
<td>output variables</td>
</tr>
<tr>
<td>Return values</td>
<td>values and a description of what they mean</td>
</tr>
<tr>
<td>Remarks</td>
<td>useful information, algorithms, etc.</td>
</tr>
<tr>
<td>Categories</td>
<td>a list of words from which to choose, for example: grid, image, pt_src, text, system, event, file, etc.</td>
</tr>
</tbody>
</table>

This information is currently in Chapter 5 of the preliminary version of the McIDAS Programmer’s Manual. In the future, cross-reference searching tools, using the Categories values, will be provided to help identify functions of particular classes.

Below is a sample template that SSEC uses for Fortran API functions and subroutines.

C THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED.
C *** McIDAS Revision History ***
C *** McIDAS Revision History ***

* $ Name: 
* $ mcname - short description of purpose/use/etc
* $ Interface:
* $ subroutine
* $ integer function
* $ double precision function
* $ mcname(integer param1, character*(*) param2, integer param3(64))
* $ Input:
* $ none
* $ param1 - description of it
* $ Input and Output:
* $ none
* $ param2 - description of it
SUBROUTINE MCNAME (.....)
IMPLICIT NONE

C --- symbolic constants & shared data
C --- external functions
C --- local variables
C --- initialized variables

** Compatibility library **

During the transition from old functions to new, SSEC will support the old interfaces. Once a function is no longer called by any McIDAS application, it will move to the compatibility library. No further testing of that function will occur. Each site is responsible for maintaining any code that uses these functions. After a period of one year, SSEC will cease issuing the function with McIDAS software upgrades.
Generic data fetchers

McIDAS is moving toward a client/server paradigm, where the applications do not have direct knowledge of the data's location or format. To accomplish this, both communications and an abstract model of the data formats must be created; the job of the server is then to map the actual file format into this abstraction and deliver it to the client. Client software can then be written to the definition of the abstraction to work with particular data formats, such as: image, grid, point source and text.

The image and grid data formats are defined and included in the suite of applications in core McIDAS. Work continues on point source and text.

Note that some of the terminology has changed.

**image** refers to data that may be viewed as a picture, but may also contain information about geo-referencing and calibration; this is to distinguish the McIDAS Area file format from the form in which data arrive at the client, namely *image format*.

**grid** hasn't changed meaning; however, the abstraction of the grid point data is somewhat different than the McIDAS grid file format.

**point source** refers to single point data; for example, the McIDAS MD file format.

**text** refers to plain text information that is undecoded, line-oriented, and intended to be read.

In all cases, the client makes a request of the server using an abstract description that includes a dataset name and other selection criteria. In many cases, choices of returned units may also be specified. After the request is made, the client then asks for *records* to be returned. The format of each record follows:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Record format</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>one-dimensional array of values of a single quantity</td>
</tr>
<tr>
<td>grid</td>
<td>two-dimensional array of values of a single quantity</td>
</tr>
<tr>
<td>point source</td>
<td>array of values at a single point of several quantities</td>
</tr>
<tr>
<td>text</td>
<td>lines of text</td>
</tr>
</tbody>
</table>
**Image data functions**

Functions are available to initiate transactions for reading and writing image data, as well as the prefix, comment card, calibration, navigation, directory and image data.

These are described in Chapter 5 of the preliminary version of the *McIDAS Programmer's Manual (10/95)*.

**Grid point data functions**

Functions are available for requesting one or more grids from a server, reading and writing grids, etc. These also are described in Chapter 5 of the preliminary version of the *McIDAS Programmer's Manual (10/95).*
Command parameter/argument fetching

The McIDAS environment provides interfaces for applications programs to pick up values of command-line arguments. Interfacing functions also provide a mechanism for parsing arbitrary strings from a source other than the command line.

All applications-level interfaces are prefixed with **mccmd** (Fortran) or **Mccmd** (C). The *cmd* means that diagnostic messages are displayed by the argument fetching subsystem for any syntax or format errors that occur. If an error does occur, the functions return a status of less than zero.

Note these significant changes:

- The string limit is no longer limited to 12-characters; it can be whatever the calling program declares.

- The syntax for keywords is now *aaa.bbb*, where *aaa* is required, and any of *bbb* may be used.

- The keyword " " (blank) signifies the command line for getting positional parameters; thus the previous functions **ikwp** and **ipp** are combined into one routine, **mccmdint**.

The functions for command line parameter fetching are listed below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mccmd</td>
<td>fetches the current McIDAS command line</td>
</tr>
<tr>
<td>Mccmdkey</td>
<td>validates the defined and command line keywords</td>
</tr>
<tr>
<td>Mccmdnam</td>
<td>fetches all keyword names occurring in the command line</td>
</tr>
<tr>
<td>Mccmdnum</td>
<td>returns the number of values associated with a command line keyword</td>
</tr>
<tr>
<td>Mccmdquo</td>
<td>fetches the quote field string command line argument</td>
</tr>
<tr>
<td>Mccmdstr</td>
<td>fetches a program command line argument in character format</td>
</tr>
<tr>
<td>Mccmdint</td>
<td>fetches a program command line argument in integer format</td>
</tr>
<tr>
<td>Mccmddbl</td>
<td>fetches a program command line argument in double format</td>
</tr>
<tr>
<td>Mccmdiyd</td>
<td>fetches a program argument in integer date format, yyyyddd</td>
</tr>
<tr>
<td>Mccmdihr</td>
<td>fetches a program argument in integer time format, hhmmss</td>
</tr>
<tr>
<td>Mccmddhhr</td>
<td>fetches a program argument in fractional hours, hh.mmmm</td>
</tr>
</tbody>
</table>
Name | Description
--- | ---
Mccmdll | fetches a program argument in integer latitude/longitude format, ddmmss
Mccmddl | fetches a program argument in latitude/longitude format, dd.ffff

The functions below are general-purpose utilities.

Name | Description
--- | ---
mcsrtoint | converts the numeric token to integer type format
mcsrtodbl | converts the numeric token to double type format
mcsrtolhex | converts the hexadecimal token to integer type format
mcsrtoid | converts the date token to integer date format, yyyddd
mcsrtodhr | converts the time token to double fractional hours, hh.ffff
mcsrtolll | converts the latitude/longitude token to integer type format, ddmss
mcsrtodll | converts the token to double fractional latitude/longitude, dddpppp
mcuvtr | converts physical units; real values
mcuvtd | converts physical units; double precision values

**Example**

```
INTEGER MCCMDINT
INTEGER FRAME
INTEGER VALUE

Replacement for IPP and IKWP

MCCMDINT(keyword, position, message, def, min, max, value)

Get first parameter on keyword "FRAME"; use LUC(51) as the default, and limit the range from 1 to LUC(13)

    if (MCCMDINT('FRA.ME', 1, 'Frame number', LUC(51), & 1, LUC(13), FRAME)) .lt. 0) return

Get first positional parameter on command; use -1 as the default and do NOT check the range (allow any integer value)

    if (MCCMDINT('', 1, 'The first value', -1, & 1, 0, VALUE)) .lt. 0) return
```
Sample code

C THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED.

C *** McIDAS Revision History ***
C *** McIDAS Revision History ***

C ? EXAM -- Describe the purpose of this command
C ? EXAM color <keywords> "quote
C ? Parameters:
C ? color | the color level to write the text in (def=1)
C ? Keywords:
C ? GRAPHIC= | graphics frame number (def=current)
C ? WIDTH= | line width (def=1)

C ? -------

    subroutine main0
    implicit none

C --- symbolic constants & shared data

C --- external functions
    integer mcmdkey
    integer mcmdint
    integer mcmdstr
    integer mcmdquo

C --- local variables
    character*80 quote
    character*10 option
    character*255 quotientstring
    integer colorlevel
    integer graphicframe
    integer width

C --- initialized variables
    character*10 keynames(2)
    data keynames/'GRA.PHIC', 'W.IDTH'/

C Verify the keywords...
    if (mcmdkey(2,keynames).lt.0) goto 999

C Get the color level
    if (mcmdint(',1,'Text color',1,
    & itrmch('MIN_COLOR_LEV',-1), itrmch('MAX_COLOR_LEV',-1),
    & colorlevel) .lt. 0) goto 999

C Get the graphic frame number
    if (mcmdint('GRA.PHIC',1,'Graphics frame',luc(56),1,
    & luc(14), graphicframe) .lt. 0) goto 999

C Get the line width
    if (mcmdint('W.IDTH',1,'Line width',1,1,10,width)
    & .lt. 0) goto 999

C Finally, get the quote string
    if (mcmdquo(quotestring).lt.0) goto 999
C Start the graphics with the desired frame and width
    call initpl( graphicframe, width)
C Write the line of text
    call wrtext( luc(63), luc(64), 10, quotestring,
                & len_trim(quotestring), colorlevel)
    call endplt

999  return
   end

EXAM 1 GRA=10 "TEST

EXAM:
EXAM: Invalid Graphics frame.
EXAM: first GRA= argument is too big --> 10
EXAM: Must be valid 'Graphics frame' integer value within range 1 thru 4.
EXAM:

EXAM 10 GRA=1 "TEST
EXAM:
EXAM: Invalid Text color.
EXAM: first positional argument is too big --> 10
EXAM: Must be valid 'Text color' integer value within range 1 thru 8.
EXAM:

EXAM 1 GRX=1 "TEST
EXAM:
EXAM: Invalid command keywords: GRX,
EXAM:
mccmd functions

The sections below give detailed information and examples of the most commonly used functions.

General argument fetching utilities

The utilities mccmdkey, mccmdnam, mccmdnum and mccmd are further defined below.

mccmdkey

mccmdkey - Validate command line keywords, printing errors to edest.
Replaces function: keychk

integer function
mccmdkey(integer numkey, character(*) valid_keywords(numkey))

Example:

   INTEGER NUMKWS
   INTEGER MCCMDKEY
   PARAMETER (NUMKWS=3)
   CHARACTER*10 KEYWORDS (NUMKWS)
   ...
   DATA KEYWORDS/'FRA.ME','COL.OR','LIN.ES'/
   C--- this means that for the keyword FRAME, at
C--- least 'FRA' must be specified; however the
C--- values 'FRA', 'FRAM' or 'FRAME' may be used.
   ...
   IF (MCCMDKEY(NUMKWS, KEYWORDS).LT.0) RETURN
   ...

mccmdnam

mccmdnam - Get all keyword names occurring in the command line.
Replaces function: kwnames

integer function
mccmdnam(integer maxkey, character(*) keywords(maxkey))

Example:

   INTEGER MAXKEY
   INTEGER MCCMDNAM
   INTEGER NUMKEYS
   PARAMETER (MAXKEY=25)
   CHARACTER*255 KEYWORDS(MAXKEY)
   ...
   NUMKEYS = MCCMDNAM(MAXKEY, KEYWORDS)
   IF (NUMKEYS.LT.0) RETURN
   ...
mccmdnum

mccmdnum - Return # values associated with given command line keyword.
    Replaces function: nkwp

    integer function
    mccmdnum(character(*)(*) keyword)

    Example:
      INTEGER MCCMDNUM
      INTEGER NUMKEYS
      ...
      NUMKEYS = MCCMDNUM('FRA.ME')

mccmd

mccmd - Build and return the current McIDAS command line.

    character(*)(*) function
    mccmd()

    Example:
      CHARACTER*255,MCCMD,COMMAND
      ...
      COMMAND = MCCMD()
The character string functions `mccmdquo` and `mccmdstr` are defined below.

**mccmdquo**

mccmdquo - Get the quote field string command line argument. Replaces function: cgfld

Note: there is NO leading " character on the string returned by this function

integer function mccmdquo(character(*) value)

Example:

```plaintext
INTEGER MCCMDQUO
INTEGER STATUS
CHARACTER*255 QUOTESTRING
...
STATUS = MCCMDQUO(QUOTESTRING)
```

C--- if the status is < 0 then an error happened

```plaintext
IF (STATUS.LT.0) RETURN
```

```plaintext
IF (LEN_TRIM(QUOTESTRING).EQ.0) THEN
C--- there is no quote string
```

**mccmdstr**

mccmdstr - Get a program command line argument in character form. Replaces functions: ckwp, cpp

integer function mccmdstr(character(*) keyword, integer position, character(*) default, character(*) value)

Example:

```plaintext
INTEGER MCCMDSTR
CHARACTER*255 VALUE
...
```

C--- get the 2nd positional parameter as a string

```plaintext
IF (MCCMDSTR(' ', 2, 'NONE', VALUE).LT.0) RETURN
```

C--- now get the first keyword parameter for TYPE=

```plaintext
IF (MCCMDSTR('TYP.E', 1, 'NONE', VALUE).LT.0) RETURN
IF (VALUE.EQ.'NONE') THEN
-- default returned or user typed "NONE" --
```
The numeric functions `mccmdint` and `mccmddbl` are described below.

**mccmdint**

`mccmdint` - Get a program command line argument in integer type format.
Replaces functions: `ikwp`, `ipp`

integer function
`mccmdint(character(*) keyword, integer position, character(*) printmsg, integer def, integer min, integer max, integer value)`

Example:

```
INTEGER MCCMDINT
INTEGER FRAME
...
C--- if the first parameter of the keyword FRAME has
C--- a syntax error, let the system print out an
C--- appropriate message and just return

IF (MCCMDINT('FRA.ME', 1, 'Frame number', LUC(51),
& 1, LUC(13), FRAME)) .LT.0) RETURN
```

**mccmddbl**

`mccmddbl` - Get a program command line argument in double type format.
Replaces functions: `dkwp`, `dpp`

integer function
`mccmddbl(character(*) keyword, integer position, character(*) printmsg, double precision def, double precision min, double precision max, double precision value)`

Example:

```
INTEGER MCCMDDBL
DOUBLE PRECISION TEMP
...
C--- Note that the range and default values must be
C--- DOUBLE PRECISION values

IF (MCCMDDBL(' ', 1, 'Temperature', 27316.D-2,
& -300.D0, 400.D0, TEMP).LT.0) RETURN
```
The time functions **mccmdihr** and **mccmddhr** are described below.

### mccmdihr

**mccmdihr** - Get a program argument in integer type time format hhmmss.
Replaces functions: ikwphr, ipphr, mkwp, mpp

Integer function
mccmdihr(character*(), keyword, integer position,
character*(), printmsg, integer def, integer min,
integer max, integer value)

Example:
INTEGRER MCCMDIHR
INTEGER BEGTIME
INTEGER ENDTIME

C--- Note that for the INTEGER functions, the form
C--- of the returned value is HHHMMSS; also, the
C--- range and default values must be specified the same way

IF (MCCMDIHR('TIM.E', 1, 'Starting time', 0, 0,
& 235959, BEGTIME).LT.0) RETURN

IF (MCCMDIHR('TIM.E', 2, 'Ending time', BEGTIME, 0,
& 235959, ENDTIME).LT.0) RETURN

### mccmddhr

**mccmddhr** - Get a program argument in fractional hours format hh.fffff.
Replaces functions: dkwpfhr, dpphhr

Integer function
mccmddhr(character*(), keyword, integer position,
character*(), printmsg, double precision def,
double precision min, double precision max,
double precision value)

Example:
INTEGRER MCCMDDHR
DOUBLE PRECISION BEGTIME
DOUBLE PRECISION ENDTIME

C--- Note that the returned value is DOUBLE PRECISION
C--- in the form: hh.nnnnn; the range and default values
C--- must be specified in the same way

IF (MCCMDDHR('TIM.E', 1, 'Starting time', 0.D0,
& 0.D0, 24.D0, BEGTIME).LT.0) RETURN

IF (MCCMDDHR('TIM.E', 2, 'Ending time', BEGTIME,
& 0.D0, 24D0, ENDTIME).LT.0) RETURN
The date routine `mccmdiyd` is described below.

### mccmdiyd

`mccmdiyd` - Get a program argument in integer type date format yyyyddd.
Replaces functions: ikwpyd, ippyd

integer function
mccmdiyd(character(*) keyword, integer position, 
       character(*) printmsg, integer def, integer min, 
       integer max, integer value)

Example:
INTEGER MCCMDIYD

C--- Note that the value returned is in the form: YYYYDDD
C--- and that the range and default values must be
C--- specified in the same manner

IF (MCCMDIYD('DAT.E', 1, 'Date', 0, 1970001, 
& 2035001).LT.0) RETURN
Latitude/longitude (or other angles)

The lat/lon functions **mccmdill** and **mccmddll** are described below.

**mccmdill**

mccmdill - Get a program argument in integer type lat/lon form ddmms.
Replaces functions: ikwpil, ippl

integer function
mccmdill(character(*) keyword, integer position,
character(*) printmsg, integer def, integer min,
integer max, integer value

Example:
```
INTEGER MCCMDILL
INTEGER LATNW
INTEGER LATSE
...
C--- Note that the value returned is in the form: DDDMMSS
C--- and that the range and default values must be
C--- specified in the same way
IF (MCCMDILL('LATITUDE', 1, 'Upper left latitude',
& 0, -900000, 900000, LATNW).LT.0) RETURN
IF (MCCMDILL('LATITUDE', 2, 'Lower right latitude',
& LATNW, -900000, 900000, LATSE).LT.0) RETURN
```

**mccmddll**

mccmddll - Get a program argument in fractional lat/lon form dd.dddddd.
Replaces functions: dkwpil, dppl

integer function
mccmddll(character(*) keyword, integer position,
character(*) printmsg, double precision def,
double precision min, double precision max,
double precision value)

Example:
```
INTEGER MCCMDDLL
DOUBLE PRECISION LATNW
DOUBLE PRECISION LATSE
...
C--- Note that the value returned is a DOUBLE PRECISION value
C--- and is in the form: ddd.mmm; the range and default
C--- values must be specified in the same way
IF (MCCMDDLL('LATITUDE', 1, 'Upper left latitude',
& 0.0, -90.00, 90.00, LATNW).LT.0) RETURN
IF (MCCMDDLL('LATITUDE', 2, 'Lower right latitude',
& LATNW, -90.00, 90.00, LATSE).LT.0) RETURN
```
mcarg functions

The mccmd functions call another series of functions whose names begin with mcarg. While the mccmd functions provide for error message output and operate only on the command line, while the mcarg functions use a handle to identify the string being parsed and translated, and you can configure the parsing of tokens.

To use these functions, first call mcargparse to place the string of characters in the internal structure. The return from mcargparse is a unique handle used in subsequent calls to fetch values. At the end, call mcargfree to free the handle and the internal space.

For example:

```
ihand = mcargparse('PGM first second 1 2 3',0,length)
istat = mcargstr(ihand, ' ', i, ' ', istat)
istat = mcargint(ihand, ' ', 3, 0, 1, 0, ival, global)
istat = mcargfree(ihand)
```

Note that support layer functions (those below the API) are used by the argument fetchers; you should avoid using them in applications programs since their interfaces or function may change. One possible exception during this transition phase is if your applications call mclext, which has no direct, API-level replacement. If you need an arbitrary string to be treated as the command line, you must make these calls:

```
istat = mcargfree(0)
istat = mccmdput(mccmdparse(cline, lenlin))
```

The mcarg functions are listed below.
Internal use functions

**General**

mcargparse - Parse the given text into arg-fetching structure.
   (return the handle)
   
   integer function
   mcargparse(character*(*), character*(*), given_syntax(10),
   integer parsed_len)

mcargfree - Free parsed arg-fetching structure for the given handle.
   
   integer function
   mcargfree(integer arg_handle)

mcargdump - Display parsed arg-fetching to McIDAS debug destination.
   
   subroutine
   mcargdump(integer arg_handle)

mcargcmd - Build and return a McIDAS command line for the given handle.
   
   character*(*) function
   mcargcmd(integer arg_handle)

mcargkey - Validate arg-fetching keywords, optionally printing errors.
   
   integer function
   mcargkey(integer arg_handle, integer numkey,
   character*(*), valid_keywords(numkey), integer printflag)

mcargnam - Fetch all keyword names within parsed arg-fetching text.
   
   integer function
   mcargnam(integer arg_handle, integer maxkey,
   character*(*), keywords(maxkey))

mcargnum - Return # args for given keyword in parsed arg-fetching text.
   
   integer function
   mcargnum(integer arg_handle, character*(*), keyword)

**Character strings**

mcargquo - Fetch the quote field string argument.
   
   integer function
   mcargquo(integer arg_handle, character*(*), value)

mcargstr - Fetch an argument in character form.
   
   integer function
   mcargstr(integer arg_handle, character*(*), keyword,
   integer position, character*(*), def, character*(*), value)
Numeric

mcargint - Fetch an argument in integer type format.

integer function
mcargint(integer arg_handle, character(*) keyword,
integer position, integer def, integer min,
integer max, integer value, character(*) arg)

mcargdbl - Fetch an argument in double type format.

integer function
mcargdbl(integer arg_handle, character(*) keyword,
integer position,
double precision def, double precision min,
double precision max, double precision value,
character(*) arg)

Time

mcargihr - Fetch an argument in integer type time format hhmmss.

integer function
mcargihr(integer arg_handle, character(*) keyword,
integer position, integer def, integer min,
integer max, integer value, character(*) arg)

mcargdhr - Fetch an argument in double fractional hours format hh.fffff.

integer function
mcargdhr(integer arg_handle, character(*) keyword,
integer position,
double precision def, double precision min,
double precision max, double precision value,
character(*) arg)

Date

mcargiyd - Fetch an argument in integer type date format yyyyddd.

integer function
mcargiyd(integer arg_handle, character(*) keyword,
integer position, integer def, integer min,
integer max, integer value, character(*) arg)
**Latitude/longitude (or other angles)**

`mcargill` - Fetch an argument in integer type lat/lon format ddddmmss.

```fortran
integer function
mcargill(integer arg_handle, character(*) keyword,
   integer position, integer def, integer min,
   integer max, integer value, character(*) arg)
```

`mcargdll` - Fetch argument in double fractional lat/lon format ddd.fffff.

```fortran
integer function
mcargdll(integer arg_handle, character(*) keyword,
   integer position,
   double precision def, double precision min,
   double precision max, double precision value,
   character(*) arg)
```

---

**Useful utilities that replace older functions**

**Numeric**

`mcstrtoint` - Convert given numeric token to integer type format.

Replaces function: `iftok`

```fortran
integer function
mcstrtoint(character(*) token, integer value)
```

`mcstrtodbl` - Convert given numeric token to double type format.

Replaces function: `dftok`

```fortran
integer function
mcstrtodbl(character(*) token, double precision value)
```

`mcstrtohex` - Convert given hexadecimal token to integer type format.

Replaces function: `iftok`

```fortran
integer function
mcstrtohex(character(*) token, integer value)
```
Time

mcstrtohms - Convert given time to integer hours, minutes and seconds.
  
  integer function
  mcstrtohms(character*(*)) token, integer hour,
  integer min, integer sec)

mcstrtoihr - Convert given time token to integer time format hhmmss.
  Replaces function: itokhr, iftok
  
  integer function
  mcstrtoihr(character*(*)) token, integer ihr)

mcstrtodhr - Convert given time token to double fractional hours hh.ffff
  Replaces function: dtokhr, dftok
  
  integer function
  mcstrtodhr(character*(*)) token, double precision dhr)

Date

mcstrtoiyd - Convert given date token to integer date format yyyyddd.
  Replaces function: itokyd, iftok
  
  integer function
  mcstrtoiyd(character*(*)) token, integer iyd)

Latitude/longitude (or other angles)

mcstrtoill - Convert given lat/lon token to integer type format dddmmss.
  Replaces function: itokll, ifto
  
  integer function
  mcstrtoill(character*(*)) token, integer ill)

mcstrtodll - Convert given token to double fractional lat/lon ddd.ffff.
  Replaces function: dtokll, dfto
  
  integer function
  mcstrtodll(character*(*)) token, double precision dll)

mcucvtr - Convert an array of real values from one unit to another.
  
  integer function
  mcucvtr(integer num, character*(*)) unitin, real bufin(*),
  character*(*)) unitout, real bufout(*), integer idif)

mcucvtd - Convert an array of double precision values from one unit to another.
  
  integer function
  mcucvtd(integer num, character*(*)) unitin, double precision bufin(*),
  character*(*)) unitout, double precision bufout(*), integer idif)
Argument fetching status

The argument fetching status codes and their descriptions are listed below.

<table>
<thead>
<tr>
<th>Status code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-] 0nnn</td>
<td>argument comes from the default</td>
</tr>
<tr>
<td>[-] 1nnn</td>
<td>argument comes from the command line</td>
</tr>
<tr>
<td>[-] 2nnn</td>
<td>argument comes from the system string table</td>
</tr>
<tr>
<td>[-] n00n</td>
<td>character string argument</td>
</tr>
<tr>
<td>[-] n01n</td>
<td>quote field string argument</td>
</tr>
<tr>
<td>[-] n10n</td>
<td>integer argument</td>
</tr>
<tr>
<td>[-] n11n</td>
<td>integer hexadecimal argument</td>
</tr>
<tr>
<td>[-] n20n</td>
<td>decimal argument (double)</td>
</tr>
<tr>
<td>[-] n21n</td>
<td>double hexadecimal argument</td>
</tr>
<tr>
<td>[-] n30n</td>
<td>date argument</td>
</tr>
<tr>
<td>[-] n31n</td>
<td>current date argument</td>
</tr>
<tr>
<td>- n32n</td>
<td>year within date argument is invalid</td>
</tr>
<tr>
<td>- n33n</td>
<td>mon month within date argument is invalid</td>
</tr>
<tr>
<td>- n34n</td>
<td>mm  month within date argument is invalid</td>
</tr>
<tr>
<td>- n35n</td>
<td>day of month (dd) within date argument is invalid</td>
</tr>
<tr>
<td>- n36n</td>
<td>day of year (ddd) within date argument is invalid</td>
</tr>
<tr>
<td>[-] n40n</td>
<td>integer time argument</td>
</tr>
<tr>
<td>[-] n41n</td>
<td>current integer time argument</td>
</tr>
<tr>
<td>- n42n</td>
<td>hours within integer time argument are invalid</td>
</tr>
<tr>
<td>- n43n</td>
<td>minutes within integer time argument are invalid</td>
</tr>
<tr>
<td>- n44n</td>
<td>seconds within integer time argument are invalid</td>
</tr>
<tr>
<td>[-] n45n</td>
<td>double time argument</td>
</tr>
<tr>
<td>[-] n46n</td>
<td>current double time argument</td>
</tr>
<tr>
<td>- n47n</td>
<td>hours within double time argument are invalid</td>
</tr>
<tr>
<td>- n48n</td>
<td>minutes within double time argument are invalid</td>
</tr>
<tr>
<td>- n49n</td>
<td>seconds within double time argument are invalid</td>
</tr>
<tr>
<td>[-] n50n</td>
<td>integer lat/lon argument</td>
</tr>
<tr>
<td>- n52n</td>
<td>degrees within integer lat/lon argument are invalid</td>
</tr>
<tr>
<td>- n53n</td>
<td>minutes within integer lat/lon argument are invalid</td>
</tr>
<tr>
<td>- n54n</td>
<td>seconds within integer lat/lon argument are invalid</td>
</tr>
<tr>
<td>[-] n55n</td>
<td>double lat/lon argument</td>
</tr>
<tr>
<td>- n57n</td>
<td>degrees within double lat/lon argument are invalid</td>
</tr>
<tr>
<td>- n58n</td>
<td>minutes within double lat/lon argument are invalid</td>
</tr>
<tr>
<td>- n59n</td>
<td>seconds within double lat/lon argument are invalid</td>
</tr>
<tr>
<td>[-] 90n</td>
<td>keyword status</td>
</tr>
<tr>
<td>nnn0</td>
<td>argument is ok</td>
</tr>
<tr>
<td>- nnn1</td>
<td>argument is invalidly formatted (invalid char)</td>
</tr>
<tr>
<td>- nnn2</td>
<td>integer argument can't contain a fraction</td>
</tr>
<tr>
<td>- nnn3</td>
<td>argument exceeds system limits for desired format</td>
</tr>
<tr>
<td>- nnn4</td>
<td>out-of-range argument &lt; given min</td>
</tr>
<tr>
<td>- nnn5</td>
<td>out-of-range argument &gt; given max</td>
</tr>
</tbody>
</table>
Applications
Development in the
ADDE

Presented by
Dave Santek
McIDAS Applications Project Leader
# Table of Contents

Overview ................................................. 2-1
   General terminology ................................ 2-2
   Client/server concepts ............................... 2-3
   ADDE name constructs ............................ 2-4

Image data .............................................. 2-5
   Terminology ........................................ 2-5
   Image data API .................................... 2-7
   Reading the image directory block ............ 2-8
   Reading the data block ........................... 2-11
   Reading the navigation block ..................... 2-15
   Reading the calibration block .................... 2-16
   Reading the comment block ....................... 2-17
   Writing image data ................................. 2-18

Gridded data .......................................... 2-22
   Gridded data API .................................. 2-23
   mgsort .............................................. 2-24
   Reading grids ..................................... 2-25

ADDE servers .......................................... 2-27
   Sending data ....................................... 2-27
   Performance ....................................... 2-27
   Error reporting .................................... 2-28
   Debugging ......................................... 2-30
   'Under the hood' ................................ 2-30

Exercise .................................................. 2-32

Sample code ........................................... 2-35
   mugaget.c ......................................... 2-35
   mugadir.c .......................................... 2-46
   mcmugutil.c ...................................... 2-50
   mug.h .............................................. 2-72
   subserv.c ......................................... 2-75
   servutil.h ......................................... 2-77
   mugarea.pgm ...................................... 2-79
Overview

The ADDE (Abstract Data Distribution Environment) provides the following enhancements to McIDAS.

**Easier access to data on remote machines**

As satellite and weather data are acquired on a distribution of workstations rather than just the mainframe, our current mechanism for identifying data sets (using numbers for Area, Grids, and MDs) becomes very limiting. A use of NFS (Network File System) with the McIDAS File Redirection would be very difficult to manage. ADDE uses names to denote datasets; file numbers are irrelevant from the users’ perspective, but are still used on the workstation serving the McIDAS data. For example, RT may point to a machine that has Real-time data on it.

**Improved performance**

The servers on MVS are always running and the data is not spooled up before sending it. The user’s application will make use of the data as soon as it starts being sent. For Unix data servers, the performance is much better than NFS.

**Better data management capabilities**

The use of names, instead of numbers, is a more intuitive way to manage data. Rather than remembering that the GOES-7 full resolution visible data is stored in Area files 101 to 104, in ADDE they may be stored in WEST/VIS-CONUS. And if the operational west satellite is switched to GOES-9, the ADDE name remains the same, though the file numbers may be different. The user doesn’t need to know, only the administrator of the server.

**More flexibility in data handling**

We’ve identified three basic data types for use in ADDE: Image, Grid and Point. These data types have one thing in common: the ability to earth locate an individual data element. Also, Image and Grid data are both 2-D arrays of data elements. These similarities make it possible to serve data to applications in a format other than its native format. In this training session, the data source is 2-D grids but the server will serve it to the application as Image data.
**Transparent incorporation of non-McIDAS data formats**

You can write servers that allow McIDAS ADDE applications to read and write in non-McIDAS data formats. We have prototyped this capability with SSM/I Pathfinder HDF files and GI (Global Imaging) satellite image data format. In today’s exercise, we will work with grids stored in text files.

This training session will provide the information you need to write your own image data and gridded data applications and servers in the ADDE. If you have not used ADDE, you should read the section titled Introduction to the ADDE, in either the *McIDAS-X* or *McIDAS-OS2 Users Guide*.

**General terminology**

The terms below are used throughout this section.

- **connection**: the initialization that occurs when a client determines the location of the dataset server and then issues a request for a data exchange. The server examines the request and determines its validity; if the request is valid, the connection is opened and the client is authorized to begin its transaction.

- **inetd**: Unix system daemon that listens to various network ports.

- **position**: absolute position number; corresponds to an Area or Grid file in a range of files.

- **sort clause**: a text string that specifies the spacial, temporal and spectral limits of a transaction. The server defines the number and format of the sort clauses for defining a request.

- **transaction**: any ADDE exchange; it implies a transfer between an ADDE client and server.
The following is an analogy to ADDE: The server is the cook in a restaurant. The patron (ADDE application) examines the menu (named datasets), places an order (request) with the waiter (client) who walks through the door (pipe) into the kitchen (server machine) and gives the order to the cook (server application). The cook prepares the food (data) and makes it available for pickup (in the pipe). The waiter brings it to your table (workstation). You, as the application, get only one chance to order and cannot make any changes. Also, you promise to eat everything on your plate (read all the data); no more, no less.

ADDE distributes data using networked servers and clients. Servers store the data and send it to a client. Clients request and receive data, and run applications on the data such as displaying imagery or contouring. When you run McIDAS commands that manipulate data, such as IMGDISP, these are client applications. When you run the DSSERVE command to manage local datasets, that is an example of running a server application.

Each McIDAS session acts as both a client and a local server. The client can request data from either its local server or from a remote server. The remote server can either be a McIDAS-MVS mainframe or McIDAS-X workstation configured as a server of data.

Clients and servers communicate via a TCP/IP communications protocol.

**Views of the world**

We’ve identified two distinct domains: the client and server. Because of the intentional separation, their view of the other’s domain is abstract.

The applications that run on the client are instructed by the user to operate on a dataset along with some selection criteria (sort clauses). The application makes a request to have the data returned in a particular format; Image, for example. By virtue of this abstraction, the application is assured of having the data available as requested or is notified that this is not possible.

On the other hand, the server promises to send back the data as requested or notify the application if there is a problem. Each server operates on a specific data format; there are individual servers for each of the McIDAS data formats. Additionally, the way the server sends the data back will necessitate more servers. For instance, we have a text file containing grid point data. We may want to serve it up as an image, to display it as a grayscale picture (IMGDISP), or as a grid to contour it (GRDDISP).
In non-ADDE commands, all image, grid and point source files are referenced by file numbers. If you don’t know the file numbers, finding data can be difficult. The ADDE commands use dataset names composed of three parts: type, group and descriptor. Type indicates the type of data: image, grid or point. McIDAS Area files are of type image; McIDAS Grid files are of type grid; McIDAS MD files are of type point.

To incorporate a non-McIDAS file format, the type of data must be identified; it is related to the organization of the data and how it will be used. For example, 2-D arrays of data can be thought of as image or grid data in our model. To classify the data as image or grid, you should consider the size, source of data, and display format (grayscale picture vs. contour). With ADDE, it can be both.

The group and descriptor point to a particular dataset. On the client, a routing table determines which server to route the request to based on the group. On the server, the group and descriptor resolve to a dataset, such as a range of McIDAS Area files.
Image data

Although accessing image data in ADDE is different than the traditional methods, there is enough similarity to make the porting of applications a surmountable task. This phase in the implementation of ADDE provides a transition to the new data access procedures without extensive changes to existing code.

You must adhere to some new rules in ADDE. The previous method allowed for random access and more freedom in taking alternative actions based on the data being read. In ADDE, the data access is sequential, and once requested, must all be read in. There are now some global keywords related to image file data. Most of these specifications are related to sectorizing the data (time and center point, for example) so that each application does not have to retrieve and validate the keywords independently. Also, the limitation of three open McIDAS Area files was lifted; the limiting factor now is resources on the client (memory) and the server (number of server processes running concurrently).

A general structural difference in the logic is also apparent. Previously, any kind of searching or sectorizing was done in the application and it was scattered throughout. Now, search conditions and sector specifications are defined at request time and are processed by the server. By the time the application is reading the data, it should be what is precisely needed.

Terminology

The terms below are used throughout this section on image data.

- **calibration block**: the block that holds the information to transform the image element’s sensor units to common physical units, such as IR temperature or visible albedo.

- **comment block**: a collection of 80-character text fields documenting any processing that may have altered the image elements, types of calibrations available for this image, or the latitude/longitude of the image’s center element.

- **data block**: a 2-dimensional matrix of image elements; the dimensions of the data block and size of each element are in the directory block.
**directory block**
an image object block containing a description of the physical characteristics of the image and the location of all ancillary blocks in the object.

**image element**
the individual data value produced by a sensor.

**image line**
the row dimension of a data block; image elements are ordered from left to right in the image line.

**image line prefix**
the prefix section of an image line that holds ancillary data defined by some image types.

**image object**
a rectangular array of elements that collectively represents an image and its collateral information.

**image object blocks**
a collection of image objects; each block contains either image elements or collateral information.

**navigation block**
the block that holds the information for determining the location of image elements in physical space; it normally includes the precise timing and attitude information of the sensor platform used to determine the earth latitude/longitude of an image element.
The ADDE image data API routines are listed below by function, along with equivalent routines for the old API.

A cursory look at this list gives the appearance of a one-to-one correspondence. There are similarities which make the transition of existing code to ADDE plausible; but logic restructuring is also part of this process, as the new routines will not just drop in. The next few sections will contrast the previous method for accessing McIDAS Area files with the ADDE method for accessing image data.

The complete description of each routine can be found in Chapter 5 of the preliminary version of the McIDAS Programmer’s Manual (10/95).

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
<th>Old API</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcaget</td>
<td>opens a connection to read the data block from an image file</td>
<td>opnara</td>
</tr>
<tr>
<td>mcaput</td>
<td>opens a connection to write image data to an image file</td>
<td>makara</td>
</tr>
<tr>
<td>mcadir</td>
<td>opens a connection to read the directory block from an image file</td>
<td></td>
</tr>
<tr>
<td>mcadrd</td>
<td>reads the directory block from an image file</td>
<td>readd</td>
</tr>
<tr>
<td>mcalin</td>
<td>reads the data portion of the current image line</td>
<td>redara</td>
</tr>
<tr>
<td>mcapfx</td>
<td>reads the prefix portion of the current image line</td>
<td>redpfx</td>
</tr>
<tr>
<td>mcanav</td>
<td>reads the navigation block from an image file</td>
<td>araget</td>
</tr>
<tr>
<td>mcacal</td>
<td>reads the calibration block from an image file</td>
<td>araget</td>
</tr>
<tr>
<td>mcacrd</td>
<td>reads the comment block from an image file</td>
<td>icget</td>
</tr>
<tr>
<td>mcaout</td>
<td>writes the prefix and data portions of an image line</td>
<td>wrtara</td>
</tr>
<tr>
<td>mcacou</td>
<td>writes the comment block to an image file</td>
<td>icput</td>
</tr>
<tr>
<td>mcadel</td>
<td>deletes an image file</td>
<td></td>
</tr>
</tbody>
</table>
Traditionally, McIDAS applications that access image objects (Areas) read the directory block before accessing the image elements. `readd` takes an Area number as input and returns the directory block associated with that number. If the application is searching for files that meet certain criteria, the application will do that selection.

The following code segment is an old Area directory read example.

**Code segment showing the area directory read loop.**

```c
  c --- I want an Area with the following
        SS = 32   ! GOES-7 Visible
        DAY = 95001 ! Jan 1 1995
        TIME = 100000 ! 10 Z

  c --- search the following range of Areas
        beg_area = 100
        end_area = 199

  c --- read Area directory for Area range beg_area to end_area
        do 100 area=beg_area,end_area
            call readd(area,directory)
        enddo

  c --- validate area existence
        if( directory(1).eq.0 ) then

  c --- check area parameters
        if( directory(3).ne.SS ) goto 100
        if( directory(4).ne.DAY ) goto 100
        if( directory(5).ne.TIME ) goto 100

  c --- do something using the values in the area directory
        ...

        endif

  100 continue
```
**mcadir and mcadrd**

The ADDE interface to the image directory is through **mcadir** and **mcadrd**. **mcadir** opens a connection based on a set of sort clauses for a given dataset name; **mcadrd** returns the directories and comments blocks.

Sort clauses restrict the search based on the image day, in the format YYDDDD; the image start time, in the format HHMMSS; and the SSEC sensor source number, 1 to 99. You must specify these sort clauses as a range of values. Below is a list of the valid sort clauses that you can use with **mcadir**.

<table>
<thead>
<tr>
<th>Sort clause format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUX YES or AUX NO</td>
<td>appends the center lat/lon, resolution and calibration types to the comment block (default=YES)</td>
</tr>
<tr>
<td>DAY bday eday</td>
<td>image Julian day range</td>
</tr>
<tr>
<td>SS ss1 ss2</td>
<td>SSEC sensor source number range</td>
</tr>
<tr>
<td>SUBSET bpos epos</td>
<td>ADDE position range, or SUBSET ALL to retrieve all directories</td>
</tr>
<tr>
<td>TIME btime etime</td>
<td>image time range</td>
</tr>
</tbody>
</table>

The sort clause AUX (Auxiliary) provides an enhanced comment block.

When the sort clause AUX YES is in the sort condition list, the image object directory server will append comment entries describing the latitude and longitude of the center element of the image, the earth area (in kilometers) covered by the center element of the image, and the valid calibration types for the image, as shown below.

```
Center latitude = latitude
Center longitude = longitude
Latitude resolution (km) = resolution
Longitude resolution (km) = resolution
Valid calibration for band band = unit
```

Once the connection is opened with **mcadir**, the application makes repeated calls to the **mcadrd** function. As long as the function status returns zero, **mcadrd** has returned an image object directory block and the associated comment block. A function status of one means all directory blocks in the specified dataset matching the sort conditions were returned. Note that a **mcadir** call must precede a call to **mcadrd** as shown in the code segment below.
Code segment showing the ADDE image object directory read loop.

c --- set sort conditions
   sorts(1) = 'SS 32 32'
sorts(2) = 'DAY 95001 95001'
sorts(3) = 'TIME 100000 100000'
sorts(4) = 'SUBSET ALL'
nsort = 4

c --- dataset name
dataset = 'RT/GOES-7'

c --- turn error reporting on
   error_flag = 1

c --- open a connection for the specified dataset
   if( mcdair(dataset,nsort,sorts,error_flag).lt.0 ) return

100 continue

c --- read an image directory block meeting the search conditions
   status = mcadir(directory,comment_cards)

c --- read failed
   if( status.lt.0 ) then
      call edest('Failed during directory read of '/dataset,0)
      return
   else if( status.eq.0 ) then

c --- found one, do something using the values in the area directory
   goto 100

endif
Reading the data block

The example code segment below is typical of the code required to read an Area. The application opens the Area (opnara), reads the data lines (redara) and then closes the Area (clsara). The application performs all image sectorization and resolution manipulation.

Code segment showing the area data read loop.

```c
  c --- set line bounds
      beg_line = 1
      end_line = area_directory(9)

  c --- set element bounds
      beg_elem = 1
      end_elem = area_directory(10)
      nelems = end_elem - beg_elem + 1

  c --- band number
      band = 1kwp('BAND',1,8)

  c --- open the Area
      call opnara(area)

  c --- declare output units (IR temperature)
      call araopt(area, 1, 'UNIT', lit('TEMP'))

  c --- declare output precision (4 bytes/element)
      call araopt(area, 1, 'SPAC', 4)

  c --- read area lines
      do 100 line = beg_line,end_line
          call redara(area, line-1, beg_elem, nelems, band, data_buffer)

  c --- scan the data elements
      do 200 element = beg_elem, end_elem
          ....
          200 continue
          100 continue

  c --- close the area
      call clsara(area)
```

**mcaget, mcalin and mcafree**

The ADDE version of the image object data block read is based on defining a request for an image sector, which may be a fragment of an image object contained in a dataset. The process is initiated by specifying a set of conditions describing the desired data segment. These sort conditions form the basis of the client’s request to the server. If an image in the dataset satisfies the sort conditions, a connection is opened and the transaction proceeds. The transaction is done on a line-by-line basis until the entire request segment is transferred.

**mcafree** frees the memory allocated by **mcaget.** It should be called after **mcalin** exhausts the transaction.
**mcaget** passes the request from the client to the server. The return status shows if the connection is opened. If it is, **mcalin** is called repeatedly to retrieve an image line. As long as the return status is zero, an array of image data is present. A return status of one means the entire sector was transferred, the transaction is complete and the connection is closed.

Applications use sort clauses to communicate with the data block server. The number and format of sort clauses are strictly regulated. These clauses allow the application to specify spacial, temporal and spectral limits of the transaction, eliminating the need for the application to scan the dataset for a particular image object. Below is a list of valid sort clauses for the **mcaget** interface.

<table>
<thead>
<tr>
<th>Sort clause format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUX YES or AUX NO</td>
<td>appends the unit and scale to the directory block</td>
</tr>
<tr>
<td>BAND band</td>
<td>spectral band, if the image has multiple bands</td>
</tr>
<tr>
<td>CAL QTIR</td>
<td>quick calibration switch for POES images</td>
</tr>
<tr>
<td>DAY day</td>
<td>image Julian day</td>
</tr>
<tr>
<td>LOCATE cor ycor xcor</td>
<td>sets the coordinate type and the coordinate positions relative to the coordinate type line and element magnification factor; positive values for blowup, negative values for blowdown</td>
</tr>
<tr>
<td>MAG imag emag</td>
<td>number of image lines and data elements</td>
</tr>
<tr>
<td>SIZE lines elems</td>
<td>stretching table name (default= no stretch)</td>
</tr>
<tr>
<td>SU name</td>
<td>absolute position in the dataset</td>
</tr>
<tr>
<td>POS pos</td>
<td>image time range</td>
</tr>
<tr>
<td>TIME btime etime</td>
<td></td>
</tr>
</tbody>
</table>

Below is a code segment illustrating the necessary steps for reading the data block of an ADDE dataset image.
Code segment showing the ADDE data block read loop.

```c
/* get the "standard" sort conditions */
if ( mcsort (nsorts, sorts, parm_pos).lt.0 ) then
   call edest('Failed to return standard sort parms',0)
   return
endif

/* get remaining (non-standard) sort conditions */
sorts(nsорт+1) = 'SIZE 100 100'
nsорт = nsорт+1

/* set the format of the returned data buffer */
format = 'I4'

/* set the units of the returned data */
unit = 'TEMP'

/* open a connection */
status = mcaget (dataset, nsорт, sorts, unit, format, &
   max_byte, msg_flag, directory, handle)
if ( status.lt.0 ) return

continue

/* read the data block */
status = mcalin (handle, data_buffer)
if ( status.lt.0 ) then
   call edest('Read failed',0)
   return

/* got a line of data */
else if ( status.eq.0 ) then
   ... goto 100
endif

/* free the handle */
status = mcafri (handle)
```

`mcaget` allows the application to specify the units and format of the data elements. Since these parameters are necessary to any data transaction, they are specified as separate parameters to `mcaget`. Units may be any physical quantity valid for the image object type; for example, TEMP or RAD. A list of valid unit identifiers is available to an application through `mca_dir` by specifying the AUX YES sort clause. Use the format parameter to specify the bytes per data element in the return array. Valid formats are I1 (1 byte/element), I2 (2 bytes/element), and I4 (4 bytes/element).
**mcasort**

The call to `mcasort` in the above code segment provides general translation of command line keyword parameters into equivalent `mcaget` sort clauses. Any image application level program may call `mcasort` to retrieve the command line keywords and return them as `mcaget` sort clauses. Because the command line keywords that `mcasort` translates were identified as applicable to the majority of image applications, they can be thought of as global keywords. Users can be assured that using the same keyword in different applications will give the same result.

Below is a list of the ADDE image object access keywords and their equivalent sort clauses.

<table>
<thead>
<tr>
<th>Command line keyword</th>
<th>MCASORT translated sort clause</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUX YES</td>
<td>aux yes</td>
<td></td>
</tr>
<tr>
<td>BAND=band</td>
<td>band</td>
<td>only one spectral band in the clause</td>
</tr>
<tr>
<td>DAY=day</td>
<td>day</td>
<td></td>
</tr>
<tr>
<td>LATLON= lat lon</td>
<td>locate lat lon</td>
<td>if keyword PLACE is not specified, loc defaults to center (EC)</td>
</tr>
<tr>
<td>LINELE= line ele</td>
<td>locate line ele</td>
<td>if keyword PLACE is not specified, loc defaults to center (IC)</td>
</tr>
<tr>
<td>MAG=imag mag</td>
<td>imag</td>
<td></td>
</tr>
<tr>
<td>PLACE=loc</td>
<td>none</td>
<td>sets loc for the LATLON, LINELE and STATION keywords</td>
</tr>
<tr>
<td>RTIME= bmin emin</td>
<td>time btime etime</td>
<td>RTIME overrides TIME</td>
</tr>
<tr>
<td>STATION= stn</td>
<td>locate lat lon</td>
<td>if keyword PLACE is not specified, loc defaults to center (EC)</td>
</tr>
<tr>
<td>TIME=btime etime</td>
<td>time btime etime</td>
<td></td>
</tr>
</tbody>
</table>
Applications rarely access the navigation block directly, except when copying it to another file. In most cases, navigational operations are performed through a dedicated API. For those instances where the navigation block is read by the application, the access interface is the `araget` subroutine. `araget` is a generalized input routine that reads application-defined blocks from an area. Applications must define the location and length of the desired information in the area. The example below uses `araget` to read the navigation block from an area.

**Code segment showing the area navigation block read.**

```c
--- get the position of the Nav block from the area directory
pos = aradir(35)
    if( aradir(63).eq.0 ) then
        length = pos - aradir(34)
    else
        length = pos - aradir(63)
    endif
--- read the Nav block
    if( length.gt.0 ) call araget(area,pos,length,nav_buffer)
```

**mcanav**

The ADDE interface to the navigation block is through `mcanav`. At any point after the connection is opened by `mcaget`, the application may retrieve the navigation block using the handle returned by the preceding `mcaget` call. The code segment below illustrates the use of `mcanav`.

**Code segment showing the ADDE image navigation block read.**

```c
--- open a connection
    status = mcaget(dataset, nsort, sorts, unit, format,
                   max_byte, msg_flag, directory, HANDLE)
    if( status.lt.0 ) return

--- read the navigation block
    status = mcanav(HANDLE, nav_buffer)
    if( status.lt.0 ) then
        call edest('Navigation Block Read failed',0)
```
Reading the calibration block

Applications use *araget* to read the calibration block from an area. *araget* takes the position and length of the calibration block as arguments and returns the block in an array. Below is a sample code segment of a typical calibration block read.

**Code segment showing the area calibration block read.**

```c

---
get the position of the Cal block from the area directory
pos = aradir(63)
if( pos.ne.0 ) then
    length = aradir(34) - pos

---
read the Cal block
call araget(area,pos,length,cal_buffer)
...
endif
...

mcacal

*mcacal* reads the calibration block of an ADDE image dataset. *mcacal* can be called any time after the connection is opened by *mcaget*. The handle returned by *mcaget* is passed to *mcacal*, which returns the associated calibration block.

**Code segment showing the ADDE image calibration block read.**

```c

---
open a connection
status = mcaget(dataset, nsort, sorts, unit, format, 
& max_byte, msg_flag, directory, HANDLE)
if( status.lt.0 ) return
...

---
read the calibration block
status = mcacal(HANDLE, cal_buffer)
if( status.lt.0 ) then
call edest('Calibration Block Read failed',0)
return
endif
...
```
icget reads the comment block entries from an area. It requires no prior function calls to set up its environment. The application repeatedly calls icget until the return status indicates all entries are accessed. Entries are returned in an 80-byte integer array, which can be moved to a character array for output.

**Code segment showing the area comment block read.**

c --- read and print the comment block entries
100 continue
     if( icget(area, comment_card).eq.0 ) then
         call movwc(comment_card, line_out)
         call spout( line_out )
     goto 100
     endif
...

**mcadrd and mcacrd**

The ADDE image API has two interfaces for reading the comment block. The first is available through the mcadir/mcadrd interface. When a connection is opened with mcadir, all mcadrd transactions return the comment block for an image object. The second interface is available through connections opened by mcaget. After all the data is read, mcacrd is used to read the comment block. Unlike icget, mcacrd returns all 80-character per record entries with a single call.

**Code segment showing the ADDE image comment block read.**

c --- open a connection
     status = mcaget(dataset, nsort, sorts, unit, format,
                    & max_byte, msg_flag, directory, HANDLE)
     if( status.lt.0 ) return

100 continue

c --- read the data block
     status = mcalin(handle, data_buffer)
     if( status.lt.0 ) then
         call edest('Read failed',0)
         return

c --- get a line of data
     else if( status.eq.0 ) then
         ...
         goto 100
     endif

c --- read the comment block
     if( mcacrd(handle, comment_buffer).ne.0 ) then
         call edest('Read of Comment Block failed',0)
         return
     endif
...
The code segment below illustrates the traditional sequence for writing a McIDAS Area. The initial step is to define the essential entries of the directory block and write the block to the Area with makara. The location of the navigation, calibration and data blocks are defined in the directory block in entries 35, 63 and 34 respectively. Once the directory block is written, the navigation and calibration blocks are filled and written to the Area with araput. Next, the Area is opened with opnara and the data is written to the Area on a line-by-line basis with wrtara. When the data block is completed, the buffers are flushed with clsara and the comment block is created and filed with ieput.

**Code segment showing the write.**

```c
! c --- initialize the directory block
! call zeros(directory_block, 64)

c --- fill the essential directory block entries
! directory_block(2) = 4        ! area version
! directory_block(3) = sss       ! satellite number
! directory_block(4) = jday       ! Julian day of image
! directory_block(5) = time       ! nominal start time of image
! directory_block(6) = start_line  ! starting image line number
! directory_block(7) = start_elem  ! starting image element number
! directory_block(9) = num_lines   ! number of lines of image data
! directory_block(10) = num elems  ! number of data elements per line
! directory_block(11) = num bytes  ! number of bytes per data element
! directory_block(12) = line_res   ! line resolution
! directory_block(13) = elem_res   ! element resolution
! directory_block(14) = num bands  ! number of bands
! directory_block(19) = 2**(band-1) ! band map
! directory_block(34) = data_offset ! byte offset to the data block
! directory_block(35) = nav_offset  ! byte offset to the nav block
! directory_block(49) = doc_length  ! length of prefix doc section
! directory_block(50) = cal_length  ! length of prefix cal section
! directory_block(51) = lev_length  ! length of prefix lev section
! directory_block(52) = lit(stype)  ! sensor source type
! directory_block(53) = lit(ctype)  ! calibration type

c --- write the directory block
! call makara(area, directory_block)

c --- initialize the navigation block
! call zeros(nav_block, nav_size)

c --- fill the navigation block entries
! NOTE: "navigation_params" is an array of navigation parameters
! which describes the geo-location of the elements of the area.
do 10 i = 1, nav_size
! nav_block(i) = navigation_params(i)
10 continue

c --- write the navigation block to the area
! call araput(area, nav_offset, nav_size*4, nav_array)

c --- initialize the calibration block
! call zeros(cal_block, cal_size)

c --- fill the calibration block entries
! NOTE: "calibration_params" is an array of calibration parameters which transform the data elements to physical units.
do 20 i = 1, cal_size
! cal_block(i) = calibration_params(i)
20 continue
```

*Applications Development in the ADDE*  
2-18  

*McIDAS Developer/Operator Training*  
October 23, 1995
20 continue

c --- write the calibration block to the area
    call araput(area, cal_offset, cal_size*4, cal_array)

c --- open the area
    call opnara(area)

c --- loop to write image lines to the area
    do 100 line = 1,num_lines

c --- pack the data array
    c NOTE: This assumes a 4 byte to 1 byte compression of the data
    call pack(num elems, data array, data array)

    c --- write a line of data to the area
    c "data array" is a (num lines) by (num elems) array of data
    c elements each of which is (num bytes) long. The elements
    c represent data for (band) from the sensor numbered (sss) on
    c (jday) at (time).
    call wrtara(area, line-1, data array(line))

100 continue

c --- close the area
    call clsara(area)

c --- write the comment block
    call getday(day)
    call gettim(time)
    cday = cfu(day)
    ctime = cfu(time)
    comment = cday(1:5)/' /ctime(1:6)/' This is a comment
    call icput(area, comment)

*mcaput and mcaout*

The process of writing an ADDE image object follows the same format as
reading an ADDE image object. The application identifies a dataset,
defines the sort condition list, and opens a connection with the server. The
connection defines the transactions to perform before the transfer is
successfully completed.

The request to open a connection is performed by *mcaput*, which requires
a valid dataset name, image object position, directory block navigation
block and calibration block. *mcaput* does not return an object handle;
therefore, only one ADDE image object can be written at a time.

The only valid sort clause defined for the ADDE image write interface is
POS, which defines the location of the image object in the dataset. You
must specify this clause or the request to open a connection will fail.

Once the connection is open, the server expects a specific number of bytes
to be transferred. The number of bytes is defined by the entries in the
directory block. Transferring too few or too many bytes results in an error.
All data block write transactions are performed by the *mcaout* function.
*mcaout* has only one argument, which is an array of image elements to
write to the data block. *mcaout* is called as many times as necessary to
transfer the desired number of bytes. *mcaput* must be called prior to
*mcaout.*
**mcacou**

The comment block can only be written after the last byte of the data block is transferred. The number of comment entries is defined during the connection phase of the transaction. Word 64 of the directory block holds the number of comment block entries for the image object. If this entry is nonzero, the number of 80-byte entries is transferred. The **mcacou** function transfers the comment block. It has only one argument, which is an array holding the entire comment block. The code segment below is a copy of the previous example re-coded to use the ADDE image object write API.

**Code segment showing the image object write.**

```c
cc --- initialize the directory block
  call zeros(directory_block, 64)
cc --- create a comment card
  call getday( day )
cday = cfu( day )
cctime = cfu( time )
comment = cday(1:5)"/"'"/"ctime(1:6)"/"'"/ This is a comment '
ncard = ( len_trim(comment) / 80 ) + 1
cc --- fill the essential directory block entries
  directory_block(2) = 4 ! version
directory_block(3) = sss ! satellite number
directory_block(4) = jday ! Julian day of image
directory_block(5) = time ! nominal start time of image
directory_block(6) = start_line ! starting image line number
directory_block(7) = start_elem ! starting image element number
directory_block(9) = num_lines ! number of lines of image data
directory_block(10) = num elems ! number of data elements per line
directory_block(11) = num bytes ! number of bytes per data element
directory_block(12) = line res ! line resolution
directory_block(13) = elem res ! element resolution
directory_block(14) = num bands ! number of bands
directory_block(19) = 2**(band-1) ! band map
call movew(memo, directory_block(25)) ! memo field
directory_block(34) = data offset ! byte offset to the data block
directory_block(35) = nav Offset ! byte offset to the nav block
directory_block(49) = doc length ! length of prefix doc section
directory_block(50) = cal length ! length of prefix cal section
directory_block(51) = lev length ! length of prefix lev section
directory_block(52) = lit( stype ) ! sensor source type
directory_block(53) = lit( ctype ) ! calibration type
directory_block(63) = cal offset ! byte offset to the cal block
directory_block(64) = ncard ! number of comment cards
cc --- initialize the navigation block
  call zeros(nav_block, nav_size)
cc --- fill the navigation block entries
cc NOTE: "navigation_params" is an array of navigation parameters
cc which describes the geo-location of the elements of the
cc image object.
do 10 i = 1, nav_size
  nav_block(i) = navigation_params(i)
10 continue
cc --- initialize the calibration block
  call zeros(cal_block, cal_size)
cc --- fill the calibration block entries
cc NOTE: "calibration parms" is an array of calibration
cc parameters which transform the data elements to physical
cc units.
```
do 20 i = 1, cal_size
  cal_block(i) = calibration_params(i)
20         continue

C --- fill the sorts array
nsorts = 1
sorts(nsorts) = 'POS' //cfu(position)

C --- open a connection to write the image object
if( mcacput( image, nsorts, sorts, directory_block, nav_block,
               & cal_block).ne.0 ) then
  call edest('Unable to initialize image ='//image,0)
  return
endif

C --- loop to write image lines to the image object
do 100 line = 1,num_lines

C --- pack the data array
C NOTE: This assumes a 4 byte to 1 byte compression of the data
C call pack( num elems, data array, data array)

C --- write a line of data to the image object
C "data array" is a (num_lines) by (num elems) array of data
C elements each of which is (num_bytes) long. The elements
C represent data for (band) from the sensor numbered (sss)
C on (jday) at (time).
  if( mcacout( data array ).ne.0 ) then
    call edest('Failed to write image line=',line)
    return
  endif
100         continue

C --- write the comment block
if( mcacou( comment ).ne.0 ) then
  call edest('Failed to write comment block',0)
  return
endif
Gridded data

The access to gridded data in ADDE is different from the traditional method. There is an option that minimizes the differences to allow using the new procedures without extensive changes to existing code, but this is not a recommended long-term solution.

Software written in the past that used information within grids was required to know the following:

- the grid file number, which contained the grid
- the grid number, which contained the grid of data

The ADDE grid software allows a user to access grids by the data contained within them. This data is described, for example, by the parameter (T, Z, etc.), level (SFC, 850, 500, etc.), and day and time. Knowledge of the actual number of the grid and/or grid file where the data resides is not necessary. However, accessing data using the number of the grid or grid file is, of course, still possible.

McIDAS grid data is still stored in grid files; however, the applications are not cognizant of these data structures. It is sufficient to know there is a grid in a dataset that can be retrieved. Datasets can have many positions, or just one position. The former is the case if a dataset spans many grid files; the latter if the dataset consists of just one grid file. The position number of the dataset defines which grid file in that dataset is being accessed.
Gridded data API

The grid data API routines are listed below by function, along with the equivalent routines for the old API. The complete description of each routine can be found in Chapter 4 of the preliminary version of the McIDAS Programmer's Manual (10/95).

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
<th>Old API</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcgdir</td>
<td>opens a connection to read grid headers and grid file headers</td>
<td>Igget</td>
</tr>
<tr>
<td>mcgget</td>
<td>opens a connection to read grids</td>
<td>Igget</td>
</tr>
<tr>
<td>mcgput</td>
<td>opens a connection to write grids</td>
<td></td>
</tr>
<tr>
<td>mcgfdrd</td>
<td>reads the grid file headers</td>
<td>Igget</td>
</tr>
<tr>
<td>mcgdrd</td>
<td>reads grid headers</td>
<td>Igget</td>
</tr>
<tr>
<td>mcgridc</td>
<td>receives grids (C 2-D array ordered)</td>
<td>Igget</td>
</tr>
<tr>
<td>mcgridf</td>
<td>receives grids (Fortran 2-D array ordered)</td>
<td>Igget</td>
</tr>
<tr>
<td>mcgoutc</td>
<td>writes grids (C 2-D array ordered)</td>
<td>Iput</td>
</tr>
<tr>
<td>mcgoutf</td>
<td>writes grids (Fortran 2-D array ordered)</td>
<td>Iput</td>
</tr>
</tbody>
</table>

Both mcgget and mcgdir search local or remote machines for grids that match given search conditions. These conditions are called sorts and they are passed to the server in a sort clause. As parameters are added to the sort clause, the search is refined and, presumably, fewer grids are matched.

If the sort clause is passed to mcgget, then the grids matching the sort conditions and the corresponding grid headers are sent to the application. If the sort clause is passed to mcgdir, the grid file header and the grid headers of the grids, but not the grids themselves, are sent back. For both mcgget and mcgdir, entries are provided to read the data sent (mcgridf and mcgridc for mcgget; mcgfdrd and mcgdrd for mcgdir).

The table below enumerates the valid sort clauses. The mcgsort function translates most of the user-entered keywords into sort clauses.
### Sort clause format

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEV lev1 .. levn</td>
<td>limits the search to particular levels</td>
</tr>
<tr>
<td>PARM parml .. parmn</td>
<td>limits the search to particular parameters</td>
</tr>
<tr>
<td>DAY day1 .. dayn</td>
<td>limits the search to specified days,</td>
</tr>
<tr>
<td></td>
<td>YYYYDDD or YYDDD</td>
</tr>
<tr>
<td>TIME timel .. timen</td>
<td>specifies the search to specific times,</td>
</tr>
<tr>
<td>DRANGE bday eday inc</td>
<td>HHMMSS</td>
</tr>
<tr>
<td>TRANGE btim etim inc</td>
<td>specifies a range of days (with increment) to search</td>
</tr>
<tr>
<td>SRC src1 .. srcn</td>
<td>specifies the range of times (with increment) to search</td>
</tr>
<tr>
<td>FHOUR hour1 .. hourn</td>
<td>specifies a range of times (with increment) to search</td>
</tr>
<tr>
<td>FDAY day1 .. dayn</td>
<td>specifies the source of the data: MRF,</td>
</tr>
<tr>
<td>FTIME time1 .. timen</td>
<td>NGM, etc.</td>
</tr>
<tr>
<td>FRANGE bhour ehour inc</td>
<td>specifies the valid forecast hours for a model run, HHMMSS</td>
</tr>
<tr>
<td>GRID bgrid egrid</td>
<td>specifies the days on which a forecast is valid</td>
</tr>
<tr>
<td>POS pos</td>
<td>specifies the times at which a forecast is valid</td>
</tr>
<tr>
<td>NUM num</td>
<td>specifies a range of forecast hours, with</td>
</tr>
<tr>
<td></td>
<td>increment</td>
</tr>
<tr>
<td></td>
<td>range of grid numbers; supersedes all the sort</td>
</tr>
<tr>
<td></td>
<td>conditions above</td>
</tr>
<tr>
<td></td>
<td>position number in the dataset</td>
</tr>
<tr>
<td></td>
<td>retrieves num grids, or ALL (default=first match)</td>
</tr>
</tbody>
</table>

### mcgsort

The utility subroutine, `mcgsort`, retrieves sort conditions from the command line related to information in the grid header. All but the POS and NUM sort clauses are processed by `mcgsort`. When `mcgsort` picks up the keyword GRID, the grid number itself, all other sort conditions are bypassed.

For some applications, you may want to limit the number of grids returned from a search to one. Therefore, the function `mcgsort` contains a flag to denote when multiple finds of a search condition are allowed. If rep_flag is greater than zero, more than one LEV, PARM, DAY, TIME, SRC, FDAY, FTIME, GRID may be specified. In the case of GRID, though, a range is specified instead of a list. The rep_flag must be greater than zero to use TRANGE and DRANGE. Note that FHOUR and FRANGE can always retrieve more than one grid.

Any application level program may call `mcgsort` to retrieve the command line keywords and return them as `mcgget` sort clauses. Note that the syntax is similar to what the user enters on the command line. For example, the user might enter the keyword TIME=12, `mcgsort` changes that to TIME 120000.
Reading grids

Suppose an application was written to compute a parameter based on the temperature at 850 mb and 500 mb from a 12-hour forecast for the MRF model 00 Z run. The application would have to read the grid headers and determine which grids matched the specification, analogous to the image directory example presented earlier.

The old API for reading grids is a call to `igget` with the grid file and grid number as input. This is used by the majority of McIDAS grid applications and is shown in the code segment below.

**Code segment for searching for particular grids.**

```c
C----- Determine grid file to search: gridfile
C----- Determine maximum number of grids in gridfile: maxgrid
C
   do 100 gridnum = 1, maxgrid
   status = igget(gridfile, gridnum, maxsize, grid, nrows, ncols, header)
    if( header(parm_index) .ne. lit('T ')) go to 100
    if( header(src_index) .ne. lit('MRF ')) go to 100
    if( header(vt_index) .ne. 120000) go to 100
    if( header(time_index) .ne. 0) go to 100
    if( header(lev_index) .ne. 850 .or. header(lev_index) .ne. 500) go to 100
C
C----- Found a match, do something with the grid
C
       ............

100 continue
```

On McIDAS-MVS there is a function `findgrd`, also used by the grid software in the DDE Demo package, which uses selection criteria similar to the sort clause for retrieving grids. This was used as the basis for retrieving grids in ADDE. A sample code segment is shown below.
Code segment for searching for particular grids in ADDE

c--- Determine the name of the dataset where MRF grids are stored
c
   name = 'RT/MRF'

c--- Set up sorts array for grids wanted.
c
   sorts(1) = 'PARM T'
sorts(2) = 'LEV 850 500'
sorts(3) = 'FHOURL 120000'
sorts(4) = 'SRC MRF'

c--- The default is to send only the first grid that matches unless
   NUM ALL is specified

c
   sorts(5) = 'NUM ALL'
   nsorts = 5

   status = mcgget(name, nsorts, sorts, ',', 'I4', maxsiz, msgflg, numgrids, numbytes)

c--- A status is set to indicate success or failure

c--- The number of grids that match (numgrids) is returned

c--- If numgrids is more or less than expected, change the sort clause or

c--- dispose of unneeded grids when they are returned

c--- Now read the grids

c
   do 100 I = 1, numgrids

      status = mcgridf(grid, header)

   c--- Do something with the grid

   .................

100 continue
ADDE servers

This section contains information specific to the ADDE servers.

Sending data

It is the server's responsibility to send only valid data back through the pipe to the client. Calls to `m0sxxsend` send the data. The server must never write to stdout since the data is passed between processes via stdin and stdout. For image data, the code provided for the exercise includes a template that can be used when writing any new image data server. The data server is `mugaget.c` [A1-A731] and the directory server is `mugadir.c` [B1-B238]. The additional code and include files support these modules. As you will see in the exercise, these templates provide a framework for the applications programmer to add the data specific code. A template for grids is not available at this time.

Performance

The client can request that data be sent back as 1, 2, or 4 bytes/data value through the `form` parameter in the `mcaget` call. The server picks up this value from the SPAC keyword [A138]. Because the server has knowledge of the data, it should recognize that a smaller size can be used and adjust the directory entry, word 12, accordingly. By optimally packing the data, the amount of bytes transferred across the network is reduced. For example, the unit BRIT (screen brightness) is requested when an application wants the grayscale value for image data. This is always a 1 byte quantity. If the value from SPAC is not 1, the server should set directory word 12 to a value of 1 before sending the directory, and pack the data to 1 byte. The client interface (`mcaget, mcalin`) will recognize that what was requested was not returned, and will unpack the data to the application's specification.
Error reporting

To report an error back to the application, the server sets an error flag in the request block, fills an error string, and calls m0sxdone. In Fortran, set word 43 in the request block to the error code, and words 44 through 63 to an error message. In C, fill in the returncode and errmsg fields in the servacct structure (in servacct.h).

The following are the currently defined error codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Source</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>various</td>
<td>Successful return</td>
</tr>
<tr>
<td>+1</td>
<td>various</td>
<td>End-of-data condition; no data was returned</td>
</tr>
<tr>
<td>-1</td>
<td>serviceman</td>
<td>Accounting data not acceptable to this host</td>
</tr>
<tr>
<td>-2</td>
<td>server_main</td>
<td>Transaction type not supported by this host</td>
</tr>
<tr>
<td>-5</td>
<td>lwpr</td>
<td>Can't find requested data</td>
</tr>
<tr>
<td>-10</td>
<td>gget</td>
<td>Client MAXWDS is too large for server</td>
</tr>
<tr>
<td>-11</td>
<td>gget</td>
<td>fnmgrd could not find the grid file</td>
</tr>
<tr>
<td>-12</td>
<td>gget</td>
<td>fnmgrd could not find the grid file</td>
</tr>
<tr>
<td>-13</td>
<td>gget</td>
<td>igget call failed</td>
</tr>
<tr>
<td>-17</td>
<td>gget</td>
<td>A unit conversion requested couldn't be done</td>
</tr>
<tr>
<td>-18</td>
<td>gget</td>
<td>Unknown sort condition was specified</td>
</tr>
<tr>
<td>-19</td>
<td>gget</td>
<td>GRID and PARM can't both be specified on sort</td>
</tr>
<tr>
<td>-20</td>
<td>mdks</td>
<td>Can't open MD file</td>
</tr>
<tr>
<td>-21</td>
<td>mdks</td>
<td>One or more requested return keys is not available</td>
</tr>
<tr>
<td>-22</td>
<td>mdks</td>
<td>There was an error while calling mdsin</td>
</tr>
<tr>
<td>-23</td>
<td>mdks</td>
<td>An invalid unit was on a sort specification</td>
</tr>
<tr>
<td>-24</td>
<td>mdks</td>
<td>An invalid key was specified on a sort condition</td>
</tr>
<tr>
<td>-25</td>
<td>mdks</td>
<td>A unit conversion was requested that could not be done</td>
</tr>
<tr>
<td>-26</td>
<td>mdks</td>
<td>Error specifying a LIST as a sort condition</td>
</tr>
<tr>
<td>-27</td>
<td>mdks</td>
<td>An invalid format specifier was given: not I, F, or C</td>
</tr>
<tr>
<td>-28</td>
<td>mdks</td>
<td>Too many sort clauses were specified; the current max is 20</td>
</tr>
<tr>
<td>-29</td>
<td>mdks</td>
<td>The SIZE parameter is smaller than the return vector</td>
</tr>
<tr>
<td>-30</td>
<td>aget</td>
<td>No area found to fit search criteria</td>
</tr>
<tr>
<td>-31</td>
<td>aget</td>
<td>Navigation error</td>
</tr>
<tr>
<td>-32</td>
<td>aget</td>
<td>Requested data not in area</td>
</tr>
<tr>
<td>-33</td>
<td>aget</td>
<td>Bad size requested</td>
</tr>
<tr>
<td>-34</td>
<td>aget</td>
<td>Band not present</td>
</tr>
<tr>
<td>-35</td>
<td>aget</td>
<td>Bad descriptor in a SORT clause</td>
</tr>
<tr>
<td>-36</td>
<td>aget</td>
<td>MAXBYT too small for returned line</td>
</tr>
<tr>
<td>-37</td>
<td>aget</td>
<td>Illegal format requested, not 11, 12, 14</td>
</tr>
<tr>
<td>Code</td>
<td>Source</td>
<td>Condition</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>-38</td>
<td>aget</td>
<td>An entry point of mcaget was called with a bad handle</td>
</tr>
<tr>
<td>-39</td>
<td>aget</td>
<td>Too many Areas open at a time</td>
</tr>
<tr>
<td>-40</td>
<td>aget</td>
<td>Call not present</td>
</tr>
<tr>
<td>-41</td>
<td>aget</td>
<td>SU= name not found</td>
</tr>
<tr>
<td>-42</td>
<td>aget</td>
<td>More than 200 Areas and time-ordered search were requested</td>
</tr>
<tr>
<td>-50</td>
<td>adir</td>
<td>Can't resolve Area names</td>
</tr>
<tr>
<td>-51</td>
<td>adir</td>
<td>No Areas found fitting selection criteria</td>
</tr>
<tr>
<td>-52</td>
<td>adir</td>
<td>Can't search this many Area directory entries</td>
</tr>
<tr>
<td>-53</td>
<td>adir</td>
<td>Bad TIME clause</td>
</tr>
<tr>
<td>-80</td>
<td>atok</td>
<td>Area WQP protection violation</td>
</tr>
<tr>
<td>-81</td>
<td>atok</td>
<td>Area number out of valid range</td>
</tr>
<tr>
<td>-82</td>
<td>atok</td>
<td>Name of area object was not recognized</td>
</tr>
<tr>
<td>-95</td>
<td>writ</td>
<td>Write to server failed</td>
</tr>
<tr>
<td>-96</td>
<td>read</td>
<td>Communications with the server timed out</td>
</tr>
<tr>
<td>-97</td>
<td>read</td>
<td>Communications with the server are terminated</td>
</tr>
<tr>
<td>-98</td>
<td>cxout</td>
<td>cxcomm could not find program module - install err</td>
</tr>
<tr>
<td>-99</td>
<td>clserv</td>
<td>soc_init() failed, TCPIP not installed or not active</td>
</tr>
<tr>
<td>-100</td>
<td>mcserv</td>
<td>Cannot connect to foreign host</td>
</tr>
<tr>
<td>-101</td>
<td>mcserv</td>
<td>socket() call failed; this should not happen</td>
</tr>
<tr>
<td>-102</td>
<td>mcserv</td>
<td>Initial transmit to server failed; host or network died</td>
</tr>
<tr>
<td>-103</td>
<td>cxout</td>
<td>cxaaddr found a bad SERVER.RTE file</td>
</tr>
<tr>
<td>-104</td>
<td>mcserv</td>
<td>Bad command arguments to clserv</td>
</tr>
<tr>
<td>-105</td>
<td>mcserv</td>
<td>Cannot open null device (should not happen)</td>
</tr>
<tr>
<td>-106</td>
<td>mcserv</td>
<td>dup() returned an error (should not happen)</td>
</tr>
<tr>
<td>-107</td>
<td>mcserv</td>
<td>Server running as root: etc/inetd/conf in error</td>
</tr>
<tr>
<td>-108</td>
<td>mcserv</td>
<td>Cannot find password entry</td>
</tr>
<tr>
<td>-109</td>
<td>mcserv</td>
<td>malloc() failed (should not happen)</td>
</tr>
<tr>
<td>-110</td>
<td>mcserv</td>
<td>putenv() failed (should not happen)</td>
</tr>
<tr>
<td>-111</td>
<td>mcserv</td>
<td>Error in prefixing MCPATH</td>
</tr>
<tr>
<td>-112</td>
<td>mcserv</td>
<td>Error prefixing PATH</td>
</tr>
<tr>
<td>-113</td>
<td>mcserv</td>
<td>Error trying to chdir</td>
</tr>
<tr>
<td>-114</td>
<td>mcserv</td>
<td>Server requested was not found</td>
</tr>
<tr>
<td>-115</td>
<td>mcserv</td>
<td>Server requested cannot execute</td>
</tr>
<tr>
<td>-116</td>
<td>mcserv</td>
<td>Wrong protocol version found</td>
</tr>
</tbody>
</table>
Debugging

Since the servers are started indirectly through inetd and mcserv, debugging can be problematic. A call to mfsxtrce from within the server will write messages to the trce file if the trace flag is set. In Fortran, the trace flag is a single integer variable in COMMON/TRACE. In C, declare extern int trace_. In both cases, setting the variable to one turns on the trace; setting it to zero suppresses the messaging.

When developing server and client applications, invariably the wanted..did error message will be emitted by the client application. This means the server did not send as much data as the client expected. The usual cause of this is on the server side, since the server must calculate and notify the client how many bytes will be sent, and then send the data.

'Under the Hood'

The following describes what occurs under the hood on Unix workstations when a request is made to another Unix workstation. You don't have to understand this to be able to write ADDE applications or servers.

When a client requests a connection to a server, the request causes the creation of a pipe, a fork, and the exec of the ADDE communications module, mcserv. The client transmits over the pipe, and then receives on it. The first utterance on the pipe from the client is a 16-byte preamble, organized as four quantities of four bytes each. They are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>version number of the protocol</td>
<td>0x0001</td>
</tr>
<tr>
<td>IP address of the server machine</td>
<td></td>
</tr>
<tr>
<td>port number</td>
<td>500</td>
</tr>
<tr>
<td>service name; for example, aget</td>
<td>four ASCII characters</td>
</tr>
</tbody>
</table>

mcserv examines the server address. If it indicates the request will be handled locally, it execs a server, based on the service name. This server inherits the pipe, and does all further communication with the client.
If the IP address signifies a remote server, `mcserv` continues, and acts as a pipe extender, using TCP/IP to the remote system. It next reads the 160-byte request block, whose fields are as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>server IP address</td>
<td>4</td>
</tr>
<tr>
<td>server port</td>
<td>4</td>
</tr>
<tr>
<td>client IP address</td>
<td>4</td>
</tr>
<tr>
<td>user initials</td>
<td>4 ascii</td>
</tr>
<tr>
<td>project number</td>
<td>4</td>
</tr>
<tr>
<td>password</td>
<td>12 ascii (now ignored)</td>
</tr>
<tr>
<td>service name</td>
<td>4 ascii</td>
</tr>
<tr>
<td>input data length</td>
<td>4</td>
</tr>
<tr>
<td>request text</td>
<td>120 ascii</td>
</tr>
</tbody>
</table>

`mcserv` attempts to connect to the port and IP address of the server. If it fails, it reads the number of bytes equal to the input data length from the pipe to empty it, and sends a 92-byte trailer record back to the client indicating that the connect failed.

If `mcserv` succeeds in connecting to the port, it first sends a resynthesized 16-byte preamble and then the 160-byte request block to the server. `mcserv` then reads and sends the number of bytes stored in the input data length field. At this point, all the information has been sent to the server. `mcserv` continues as an intermediary between the client application and the server by copying the bytes sent by the server to the pipe being read by the application.

On the server machine, a `mcserv` is started by `inetc`. It goes through the same steps, except this time the service is found to be local. `mcserv` execs the server based on the service name, which reads the request and sends the response. When the server is finished sending its response, it sends the 92-byte trailer block, and exits.

The size of data sent to or from the server may be many megabytes. The design is explicitly stream oriented, so both the client and the server can be working simultaneously. The server locates the data and transmits it to the client via a pipe and/or TCP/IP. The client reads out of the pipe and operates on the data. Intermediate storage of the data is not needed on either end for the whole amount of data being sent. Since the pipe is a finite size, the server will wait to write if the pipe is full or the client will wait to read if the pipe is empty. If two minutes elapse with no activity on the pipe, the process stops. The process on the other end of the pipe also stops at this time.
Exercise

The image server consists of five main parts:

- main to all servers: subserv.c [E1-E102]
- directory server: mugadir.c [B1-B238]
- data server: mugaget.c [A1-A731]
- MUG-specific functions: mcmugutil.c [C1-C1515]
- generic functions: mcservutil.c

These modules have three include files: mug.h [D1-D210], servacct.h and servutil.h [F1-F136]. The only one unique to your server development is mug.h, which contains error messages/codes, default values and struct declarations.

The only piece from the above list that you will be concerned with is the fourth one: MUG-specific functions. This source file is based on seven basic functions that you must write whenever a new format of image data will be served:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IsMugFormat</td>
<td>validates the file format</td>
<td>[C412-C448]</td>
</tr>
<tr>
<td>ReadMugDir</td>
<td>reads a directory</td>
<td>[C449-C600]</td>
</tr>
<tr>
<td>MugNavImgToEarth</td>
<td>converts line/elem to lat/lon</td>
<td>[C601-C691]</td>
</tr>
<tr>
<td>MugNavEarthToImg</td>
<td>converts lat/lon to line/elem</td>
<td>[C692-C782]</td>
</tr>
<tr>
<td>ReadMugLine</td>
<td>reads a line of data</td>
<td>[C1047-C1209]</td>
</tr>
<tr>
<td>ReadMugCalcod</td>
<td>reads a calibration codicil</td>
<td>[C1210-C1246]</td>
</tr>
<tr>
<td>ReadMugNavCod</td>
<td>reads a navigation codicil</td>
<td>[C1247-C1369]</td>
</tr>
</tbody>
</table>

The other functions in the file are based on one or more of the above functions. They will also need to be written in future server development, but the existing ones in mcmugutil.c can be used as templates.

In the interest of time, you will only write the function to read a line of data (ReadMugLine). In addition, you will receive the basic skeleton including the interface and data calibration. Keep these three things in mind when completing the function:

- All lines are the same size.
- The function must handle skipped lines.
- Each file has three lines of header at the top.
Below is an example of the ReadMugLine interface.

```c
int ReadMugLine(char *name, READPARM *read, int band, short *buf, char *err)
/*
 * read a line of a MUG Training Course image
 * name - filename to read
 * read - READPARM struct containing read specs
 * band - band number of elements to read
 * buf - buffer containing image data
 * err - error string to return on failure
 */
```

The READPARM struct contains specifications that may be needed in the function. It is also a way to add parameters that may be needed in the future without changing the interface of the function. A sample READPARM struct is shown below. You will not use everything contained in the struct.

```c
typedef struct READPARM_
{
  char src_type[4]; /* source type (GVAR, MSAT, ...) */
  char des_unit[4]; /* destination units (RAW, BRIT, ...) */
  char src_unit[4]; /* source type (RAW, BRIT, ...) */
  int begele; /* beginning element */
  int beglin; /* beginning line */
  int bufsiz; /* size of buffer to read */
  int des_len; /* destination byte length of one pixel */
  int elem_res; /* resolution in element direction */
  int line_res; /* resolution in line direction */
  int maxele; /* last element in image */
  int maxlin; /* last line in image */
  int minele; /* first element in image */
  int minlin; /* first line in image */
  int numbnd; /* number of bands in image */
  int numele; /* number of elements to read */
  int numlin; /* number of lines to read */
  int src_len; /* source byte length of one pixel */
  int ul_elem; /* elem in upper left corner of image */
  int ul_line; /* line in upper left corner of image */
} READPARM;
```

After you've written ReadMugLine, you may compile it with the Makefile provided. Below are the options available for this Makefile:

**Makefile options**

<table>
<thead>
<tr>
<th>Makefile options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>make mugadir</td>
<td>make directory server</td>
</tr>
<tr>
<td>make mugaget</td>
<td>make data server</td>
</tr>
<tr>
<td>make</td>
<td>make both servers (default)</td>
</tr>
<tr>
<td>make clean</td>
<td>clean the environment of relevant .o files</td>
</tr>
</tbody>
</table>

The Makefile has an option to set a DEBUG flag, which you'll want to set for development. It causes a file named trace to be written in the first writable directory in your MCPATH. You may add lines to this trace file with a call to the function, m0sxtrce_(char *, FsLen *) in your source code.
To turn off the trace (debugging) from the servers, you only need to unset
the DEBUG flag in the Makefile, make clean and make. You do not need
to remove or comment any calls to m0sxtrc_e in your source code.

You may also notice that main to your servers (in this case mugadir.c and
mugaget.c) is always the code contained in subserv.c. The Makefile
automatically compiles subserv.c as the main to your server. You must be
careful to adhere to the Makefile compilation sequence in any future server
development.

See [C1047-C1209] for a possible solution to ReadMugLine.
Sample code

mugaget.c

A 1: #include <stdio.h>
A 2: #include <stdlib.h>
A 3: #include <strings.h>
A 4: #include "mug.h"
A 5: 
A 6: int
A 7: mugaget(servacct *control)
A 8: {
A 9: 
A 10: const char *dir; /* name of directory to search */
A 11: const char *dum; /* dummy for arg fetchers */
A 12: const char *img_place; /* image placement key (EC..EU) */
A 13: 
A 14: char dbg[MAX_ERR_LEN]; /* textual debug message */
A 15: char err[MAX_ERR_LEN]; /* err string from SelectImages */
A 16: 
A 17: double p_lat; /* center lat of returned image */
A 18: double p_lon; /* center lon of returned image */
A 19: 
A 20: float p_elem; /* center elem of returned image */
A 21: float p_line; /* center line of returned image */
A 22: 
A 23: int cards[MAX_NUM_CARDS*4]; /* mcidas comment cards */
A 24: 
A 25: int a_elem; /* requested area element */
A 26: int a_line; /* requested area line */
A 27: int aradir[IMG_DIR_LEN]; /* mcidas area directory */
A 28: int band; /* band number to read */
A 29: int base res; /* base resolution */
A 30: int begele; /* beg element of ret image */
A 31: int bday, eday; /* beginning/end day */
A 32: int bpos, epos; /* beginning/end position */
A 33: int bss, ess; /* beginning/end set */
A 34: int btm, etim; /* beginning/end time */
A 35: int bytes_per_line=0; /* number of bytes/image line */
A 36: int bytes_per_pix=0; /* number of bytes/pixel */
A 37: int bytes_to_send=0; /* number of data bytes to send */
A 38: int bytes_to_zero=0; /* number of bytes to zero */
A 39: int calcod[MAX_CAL_LEN]; /* mcidas calibration codicil */
A 40: int curline; /* loop variable for lines */
A 41: int data_bytes; /* number of bytes in read */
A 42: int elem_res; /* element resolution */
A 43: int elem_to_send; /* size of total bytes to send */
A 44: int four=4; /* loop variable */
A 45: int i; /* function return status */
A 46: int istat;
A 47: int len_calcod=0; /* length of mcidas cal codicil */
A 48: int len_navcod=0; /* length of mcidas nav codicil */
A 49: int line_res; /* line resolution */
A 50: int navcod[MAX_NAV_LEN]; /* mcidas navigation codicil */
A 51: int nelems; /* number of elements to send */
A 52: int nlines; /* number of lines to send */
A 53: int one=1; /* the number one */
A 54: int orig_nelems; /* unaltered # of lines to send */
A 55: int orig_nlines; /* unaltered # of lines to send */
A 56: int p_elem_off; /* center elem offset */
A 57: int p_line_off; /* center line offset */
A 58: int pt_elem; /* elem number to navigate */
A 59: int pt_line; /* line number to navigate */
A 60: int spac; /* byte size of one pixel */
A 61: int total_bytes=0; /* number of bytes to send */
A 62: int ul_elem; /* upper left elem of ret image */
A 63: int ul_line; /* upper left line of ret image */
A 64: int zeros_on_left; /* number of zero bytes on left */
A 65:
A 66: short *data=NULL; /* line of image data */
A 67:
A 68: CRITERIA *request=NULL;
A 69: FILELIST *satisfy=NULL;
A 70: READPARM *read=NULL;
A 71:
A 72: strcpy(dbg, "starting subcommand");
A 73: m30xstrc_(dbg, strlen(dbg));
A 74:
A 75: /*
A 76: * directory to be searched
A 77: */
A 78:
A 79: istat = Mcargquo(0, &dir);
A 80:
A 81: /*
A 82: * beginning position
A 83: */
A 84:
A 85: istat = Mcargint(0, " ", 2, 0, 999, -999, &bpos, &dum);
A 86:
A 87: /*
A 88: * set the ending position equal to the beginning
A 89: * position. the underlying function, SelectImages, allows
A 90: * a range to be specified, but the current ADDE protocol
A 91: * does not support this.
A 92: */
A 93:
A 94: epos = bpos;
A 95:
A 96: /*
A 97: * image placement key (EC, EU, ...)
A 98: */
A 99:
A 100: istat = Mcargstr(0, " ", 3, " ", &img_place);
A 101: /*
A 102: */
A 103: * base resolution
A 104: */
A 105:
A 106: istat = Mcargint(0, " ", 6, 1, 999, -999, &base_res, &dum);
A 107:
A 108: /*
A 109: * number of lines and elements to read
A 110: */
A 111:
A 112: istat = Mcargint(0, " ", 7, DEF_NUM_LINES, 999, -999, &nlines, &dum);
A 113: istat = Mcargint(0, " ", 8, DEF_NUM_ELEMS, 999, -999, &nelems, &dum);
A 114:
A 115: orig_nlines = nlines;
A 116: orig_nelems = nelems;
A 117:
A 118: /*
A 119: * band number to read
A 120: */
A 121:
A 122: istat = Mcargint(0, "BAND", 1, 1, 999, -999, &band, &dum);
A 123:
A 124: /*
A 125: * line and element resolution
A 126: */
A 127:
A 128: istat = Mcargint(0, "LMA", 1, base_res, 999, -999, &line_res, &dum);
A 129: istat = Mcargint(0, "EMA", 1, base_res, 999, -999, &elem_res, &dum);
A 130:
A 131: line_res = abs(line_res);
A 132: elem_res = abs(elem_res);
A 133:
A 134: /*
A 135: * size of one pixel (spac)
A 136: */
A 137:
A 138: istat = Mcargint(0, "SPA.C", 1, 0, -1, 0, &spac, &dum);
A 139:
A 140: /*
A 141: * this server will only serve one byte data
A 142: */
A 143:
A 144: if (spac > 1)
A 145: spac = 1;
A 146:
A 147: /*
A 148: * beginning and ending time
A 149: */
A 150:
A 151: istat = Mcargihr(0, "TIM.E", 1, -1, -1, 240000, &btim, &dum);
A 152: istat = Mcargihr(0, "TIM.E", 2, btim, -1, 240000, &etim, &dum);
A 153:
A 154: /*
A 155: * beginning and ending day
A 156: */
A 157:
A 158: istat = Mcargint(0, "DAY", 1, -1, 999, -999, &bday, &dum);
A 159: istat = Mcargint(0, "DAY", 2, bday, 999, -999, &eday, &dum);
A 160:
A 161: /*
A 162: * beginning and ending mcidas ss number
A 163: */
A 164:
A 165: istat = Mcargint(0, "SS", 1, -1, 999, -999, &bss, &dum);
A 166: istat = Mcargint(0, "SS", 2, bss, 999, -999, &ess, &dum);
A 167:
A 168: sprintf(dbg, "dir = [%s]", dir);
A 169: mstrcpy(dbg, strlev(dbg));
A 170:
A 171: sprintf(dbg, "bps = %d epo = %d", bpos, epos);
A 172: mstrcpy(dbg, strlev(dbg));
A 173:
A 174: sprintf(dbg, "btim = %d etim = %d", btim, etim);
A 175: mstrcpy(dbg, strlev(dbg));
A 176:
A 177: sprintf(dbg, "bday = %d eday = %d", bday, eday);
A 178: mstrcpy(dbg, strlev(dbg));
A 179:
A 180: sprintf(dbg, "bss = %d ess = %d", bss, ess);
A 181: mstrcpy(dbg, strlev(dbg));
A 182:
A 183: /*
A 184: * malloc space to fill a struct
A 185: */
A 186: * containing search criteria
A 187:
A 188: request = (CRITERIA *)malloc(sizeof(CRITERIA));
A 189:
A 190: if (request == NULL)
A 191: {
A 192: (void)strcpy(err, ERR_MSG_MALLOC);
A 193: return ERR_STAT_MALLOC;
A 194: }
A 195:
A 196: request->begday = bday;
A 197: request->senday = eday;
A 198: request->begtim = btim;
A 199: request->endtim = etim;
A 200: request->begss = bss;
A 201: request->ends = ess;
A 202:
/* malloc space to return a linked list of files meeting CRITERIA */
satisfy = (FILELIST *)malloc(sizeof(FILELIST));
if (satisfy == NULL) {
    (void)strcpy(err, ERR_MSG_MALLOC);
    return ERR_STAT_MALLOC;
}
/* malloc space for the read parameters */
read = (READPARM *)malloc(sizeof(READPARM));
if (read == NULL) {
    (void)strcpy(err, ERR_MSG_MALLOC);
    return ERR_STAT_MALLOC;
}
/* get a list of qualified files */

istat = SelectMugImages((char *)dir, bpos, epos, request, &satisfy, err);
if (istat == FAILURE) {
    (void)strcpy(control->errmsg, err);
    control->returncode = ERR_STAT_SELECT;
    return (control->returncode);
}
/* no images were found */

if (satisfy == NULL) {
    (void)strcpy(control->errmsg, ERR_MSG_NOIMG);
    control->returncode = ERR_STAT_NOIMG;
    return (control->returncode);
}
/* loop through the MUG files */

while (satisfy != NULL) {
    /* read the directories */
    sprintf(dbg, "reading file %s", satisfy->name);
    m0sxtrce_(dbg, strlen(dbg));
    istat = ReadMugDir(satisfy->name, aradir, err);

    if (istat == FAILURE) {
        sprintf(dbg, "failure reading file %s", satisfy->name);
        m0sxtrce_(dbg, strlen(dbg));
        printf(dbg, "%R: %s", err);
        m0sxtrce_(dbg, strlen(dbg));
    }
continue;

/* fill read specifications with
* the size of the read buffer,
* the min and max number of lines/elements,
* the size of a source pixel and
* the number of bands in the image
*/

read->bufsize = READ_BUFFER_SIZE;
read->elem_res = elem_res;
read->line_res = line_res;
read->minlin = 0;
read->minele = 0;
read->maxlin = aradir[8];
read->maxele = aradir[9];
read->src_len = sizeof(short);
read->numband = aradir[13];

/* readjust the number of lines
* and elements if needed. a number
* of 99999 from the client means that
* the entire image was requested
*/

if (nlines == 99999)
    { nlines = aradir[8];
      orig_nlines = nlines;
    }

if (nelems == 99999)
    { nelems = aradir[9];
      orig_nelems = nelems;
    }

/* calculate center line and element
* of requested image object
*/

/* pick up coordinate parameters */

switch (img_place[0])
    { case 'E': /* earth coordinates */
        { void)strcpy(dbg, "earth coord \n*");
          musxtrce_(dbg, strlen(dbg));
          istat = Mcargd1[0], " ", 4, (double)0,
              (double)999, (double)999, &p_lat, &dum);
          istat = Mcargd1[0], " ", 5, (double)0,
              (double)999, (double)999, &p_lon, &dum);
          /* convert lat/lon to line/element */
          if (istat == FAILURE)
A 347: } 
A 348: (void)strncpy(control->errormsg, err);
A 349: control->returncode = ERR_STAT_NONAV;
A 350: return (control->returncode);
A 351: 
A 352: break;
A 353: 
A 354: case 'A': /* file coordinates */
A 355: case 'I':
A 356: default:
A 357:     istat = Mca想去(0, "", 4, 0, 
A 358: 999, -999, &a_line, &dum);
A 359:     istat = Mca想去(0, "", 5, 0, 
A 360: 999, -999, &a_elem, &dum);
A 361:     p_line = (float)a_line;
A 362:     p_elem = (float)a_elem;
A 363:     break;
A 364: 
A 365: (void)sprintf(dbg, "place coord = %f %f \n", p_line, p_elem);
A 366: m0xstrce_(dbg, strlen(dbg));
A 367: 
A 368: /* calculate line/ele of upper left corner line/ele
A 369: */
A 370: switch (img_place[1])
A 371: {
A 372:     case 'D': /* lower right */
A 373:         p_line_off = orig_nlines;
A 374:         p_elem_off = orig_nelems;
A 375:         break;
A 376:     case 'C': /* centered */
A 377:         p_line_off = (orig_nlines / 2);
A 378:         p_elem_off = (orig_nelems / 2);
A 379:         break;
A 380:     default: /* upper left */
A 381:         p_line_off = 1/line_res;
A 382:         p_elem_off = 1/element_res;
A 383:         break;
A 384: 
A 385: ul_line = (int)p_line - (p_line_off * line_res) + 1;
A 386: ul_elem = (int)p_elem - (p_elem_off * element_res) + 1;
A 387:     (void)sprintf(dbg, "start coord = %d %d \n", ul_line, ul_elem);
A 388: m0xstrce_(dbg, strlen(dbg));
A 389: 
A 390: /*
A 391: * initialize read struct with
A 392: * starting line and element
A 393: */
A 394: read->ul_line = ul_line;
A 395: read->ul_elem = ul_elem;
A 396: 
A 397: /*
A 398: * check the bounds of the returned image
A 399: * to make sure it will contain data
A 400: * if not, return an error
A 401: */
A 402: istat = CheckImgBounds(read, orig_nlines, orig_nelems);
A 420:    if (istat == FAILURE)
A 421:    {
A 422:      (void)strncpy(control->errmsg, ERR_MSG_BLANK_IMG);
A 423:      control->returncode = ERR_STAT_BLANK_IMG;
A 424:      return (control->returncode);
A 425:    }
A 426:    
A 427:    /*
A 428:     * adjust output area directory
A 429:     */
A 430:    aradir[8] = nlines;
A 431:    aradir[9] = 4 * ((nelems + 3) / 4);
A 432:    nelems = aradir[9];
A 433:    if (spac == 0)
A 434:      spac = aradir[10];
A 435:    aradir[10] = spac;
A 438:    /*
A 439:     * fill read specifications with
A 440:     * the size of a destination pixel
A 441:     */
A 442:    read->des_len = aradir[10];
A 443:    /*
A 444:     * read mcidas navigation codicil
A 445:     */
A 446:    pt_line = 2 - ul_line;
A 447:    pt_elem = 2 - ul_elem;
A 448:    istat = ReadMugNavCod(satisfy->name, navcod,
A 449:      pt_line, pt_elem, &len_navcod);
A 450:    if (istat == FAILURE)
A 451:    {
A 452:      (void)strncpy(control->errmsg, ERR_MSG_NONAV);
A 453:      control->returncode = ERR_STAT_NONAV;
A 454:      return (control->returncode);
A 455:    }
A 456:    /*
A 457:     * read mcidas calibration codicil
A 458:     */
A 459:    istat = ReadMugCalCod(satisfy->name, calcod, &len_calcod);
A 460:    if (istat == FAILURE)
A 461:    {
A 462:      (void)strncpy(control->errmsg, ERR_MSG_NOCAL);
A 463:      control->returncode = ERR_STAT_NOCAL;
A 464:      return (control->returncode);
A 465:    }
A 466:    /*
A 467:     * compute total byte transfer to client
A 468:     */
A 470:    total_bytes = MAX_CARD_LEN *
aradir[63] +
aradir[33] +
aradir[8] *
bytes_per_line;

/* send total bytes */
control->reply_length += total_bytes;
swbyt4_(&total_bytes, &one);
sprintf(dbg, "sending %d bytes", total_bytes);
莫斯xtrce_(dbg, strlen(dbg));
mosxsend_(&four, &total_bytes);

/* send the directory */
aradir[0] = satisfy->pos;
(void)SendDir(aradir);

/* send the nav */
(void)SendCod(navcod, len_navcod);
(void)SendCod(calcod, len_calcod);

/* read and send the image one line at a time */
data_bytes = orig_nelems *
   read->arc_len *
   read->numband +
aradir[14];
data_bytes = read->bufsz;
data = (short *)malloc(data_bytes);
if (data == NULL)
   {
      (void)strcpy(control->errmsg, ERR_MSG_MALLOC);
      control->returncode = ERR_STAT_MALLOC;
      return (control->returncode);
   }

/* total elements needed */
curline = ul_line;
begele = ul_elem;
read->numlin = read->bufsz;
bytes_to_send = bytes_per_line;
bytes_per_pix = read->des_len * read->numband;
for (i=1; i<=orig_nlines; i++)
   {
      /* if the requested line is less than one
if (curline < 0 || curline > read->maxlin)
{
    bytes_to_zero = aradir[9] * read->des_len
    read->numBand;
    sprintf(dbg, "zero line %d", curline);
    m0sxtrec_ (dbg, strlen(dbg));
    (void) SendZeros (data, bytes_to_zero);
    /* increment the line pointer */
    curline += line_res;
    bytes_to_zero = 0;
    continue;
}

/* if the image requested does not contain data on the left side,
   send the appropriate number of zeros and adjust the starting
   element */
zeros_on_left = 0;

if (ul_elem < 0)
{
    bytes_to_zero = abs (ul_elem / elem_res) * read->des_len
    read->numBand;
    zeros_on_left = bytes_to_zero;
    sprintf(dbg, "%s left %d %d bytes", curline. bytes_to_zero);
    m0sxtrec_ (dbg, strlen(dbg));
    (void) SendZeros (data, bytes_to_zero);
    begele = 0;
    nelems = read->maxele;
    bytes_to_send = bytes_per_line - bytes_to_zero;
    bytes_to_zero = 0;
}

/* if the image requested does not contain data on the right side,
   adjust the number of elements.
   the appropriate number of zero filler
   will be send later (after the read) */

if (ul_elem + (aradir[9] * elem_res) > read->maxele)
{
    bytes_to_zero = (ul_elem / elem_res + aradir[9]) -
                    (read->maxele / elem_res) * read->des_len
                    ...
...
read->numband;
nelems = read->maxele - ulElem/elem_res;
if (ulElem < 0)
    nelems = read->maxele;

sprintf(dbg, "zero left %d bytes", zeros_on_left);
m0xstrcE_(dbg, strlen(dbg));
bytes_to_send = bytes_per_line - bytes_to_zero - zeros_on_left;

sprintf(dbg, "send %d bytes", bytes_to_send);
m0xstrcE_(dbg, strlen(dbg));

sprintf(dbg, "zero %d bytes", bytes_to_zero);
m0xstrcE_(dbg, strlen(dbg));
}
/*
 * fill read specifications with
 * beginning line/element and
 * the number of lines and elements
 * to read
 */
read->beglin = curline;
read->begele = begele;
read->numele = nelems;

/*
 * read a line of data
 */

istat = ReadMugLine(satisfy->name, read, band, data, err);
if (istat == FAILURE)
{
    (void)strcpy(control->errormsg, err);
    control->returncode = ERR_STAT_BADLINE;
    return (control->returncode);
}
/*
 * reduce the resolution
 */
elem_to_send = nelems / elem_res;

(void)m0resred((char *)data, &elem_res,
    &elem_to_send, &bytes_per_pix);

/*
 * send line of image data
 */

(void)SendLine(data, bytes_to_send);
/*
 * if we still have bytes to zero (set
 * because the image is padded on the right,
 * then we send those
 */
if (bytes_to_zero != 0)
    { (void)SendZeros(data, bytes_to_zero);
        bytes_to_zero = 0;
}
A 707:    /*
A 708:    * increment the line pointer
A 709:    */
A 710:    curline += line_res;
A 711: }
A 712:    */
A 713:    /* send the comment cards */
A 714:    if (aradir[63] != 0)
A 715:        (void)SendCards(aradir, cards);
A 716:    */
A 717:    */
A 718:    /* increment the pointer in the list of files */
A 719:    satisfy = satisfy->next;
A 720: }
A 721: }
A 722: return SUCCESS;
A 723: }
A 724: }
A 725: }
A 726: return SUCCESS;
A 727: }
A 728: }
A 729: return SUCCESS;
A 730: }
A 731: }
mugadir.c

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "mug.h"

int mugadir(servacct *control)
{
  const char *dir;    /* name of directory to search */
  const char *dum;    /* dummy for arg fetchers */
  char dbg[MAX_ERR_LEN]; /* textual debug message */
  char err[MAX_ERR_LEN]; /* err string from SelectImages*/
  int cards[MAX_NUM_CARDS*4]; /* mocidas comment cards */
  int aradir[IMG_DIR_LEN]; /* mocidas area directory */
  int bday, eday; /* beginning/ending day */
  int bpos, epos; /* beginning/ending position */
  int bss, ess; /* beginning/ending ss */
  int btim, etim; /* beginning/ending time */
  int dummy=9999; /* dummy pos number to send */
  int four=4; /* size of nbyte to send */
  int istat; /* function return status */
  int nbytes=0; /* number of bytes to send */
  int one=1; /* the number one (1) */

  CRITERIA *request=NULL;
  FILELIST *satisfy=NULL;

  strcpy(dbg, "starting mugadir");

  mosexrce_(dbg, strlen(dbg));

  istat = Mccargquo(0, &dir);

  /*
   * beginning and ending position
   */

  istat = Mccargint(0, " ", 2, 0, 999, -999, &bpos, &dum);
  istat = Mccargint(0, " ", 3, bpos, 999, -999, &epos, &dum);

  /*
   * if the bpos is negative, reorganize the positions
   * since the ADDE convention for time ordered requests
   * is to always have zero as the epos
   */

  if (bpos < 0)
  {
    epos = bpos;
    bpos = 0;
  }

  /* beginning and ending time
   */

  istat = Mccarih(0, "TIM.E", 1, -1, -1, 240000, &btim, &dum);
  istat = Mccarih(0, "TIM.E", 2, btim, -1, 240000, &etim, &dum);
b 67: /*
68: * beginning and ending day
69: */
70:
71: istat = Mcargint(0, "DAY", 1, -1, 999, -999, &bday, &dum);
72: istat = Mcargint(0, "DAY", 2, bday, 999, -999, &eday, &dum);
73:
74: /*
75: * beginning and ending mcidas ss number
76: */
77:
78: istat = Mcargint(0, "SS", 1, -1, 999, -999, &bss, &dum);
79: istat = Mcargint(0, "SS", 2, bss, 999, -999, &ess, &dum);
80:
81: sprintf(dbg, "dir = [%s]", dir);
82: mStrxtrce_(dbg, strlen(dbg));
83:
84: sprintf(dbg, "bpos = %d epos = %d", bpos, epos);
85: mStrxtrce_(dbg, strlen(dbg));
86:
87: sprintf(dbg, "btim = %d etim = %d", btim, etim);
88: mStrxtrce_(dbg, strlen(dbg));
89:
90: sprintf(dbg, "bday = %d eday = %d", bday, eday);
91: mStrxtrce_(dbg, strlen(dbg));
92:
93: sprintf(dbg, "bss = %d ess = %d", bss, ess);
94: mStrxtrce_(dbg, strlen(dbg));
95:
96: /*
97: * malloc space to fill a struct
98: * containing search criteria
99: */
100:
101: request = (CRITERIA *)malloc(sizeof(CRITERIA));
102:
103: if (request == NULL)
104: {
105:     (void)strcpy(err, ERR_MSG_MALLOC);
106:     return ERR_STAT_MALLOC;
107: }
108:
109: request->bogday = bday;
110: request->endday = eday;
111: request->begtim = btim;
112: request->endtim = etim;
113: request->begss = bss;
114: request->endss = ess;
115:
116: /*
117: * malloc space to return a linked
118: * list of files meeting CRITERIA
119: */
120:
121: satisfy = (FILELIST *)malloc(sizeof(FILELIST));
122:
123: if (satisfy == NULL)
124: {
125:     (void)strcpy(err, ERR_MSG_MALLOC);
126:     return ERR_STAT_MALLOC;
127: }
128:
129: istat = SelectMugImages((char *)dir, bpos, epos, request, &satisfy, err);
130:
131: if (istat == FAILURE)
132: {
133:     (void)strcpy(control->errmsg, err);
134:     control->returncode = ERR_STAT_SELECT;
135:     return (control->returncode);
136: }
137:
138: /*
if (satisfy == NULL)
{
    (void)strncpy(control->errmsg, ERR_MSG_NOIMG);
    control->returncode = ERR_STAT_NOIMG;
    return (control->returncode);
}

/*
 * loop through the MUG files
 */
while (satisfy != NULL)
{
    /* read the directories */
    sprintf(dbg, "reading file %s", satisfy->name);
    m0sxstrce_(dbg, strlen(dbg));
    istat = ReadMugDir(satisfy->name, aradir, err);
    if (istat != FAILURE)
    {
        /* build the comment cards */
        sprintf(dbg, "success reading file %s", satisfy->name);
        m0sxstrce_(dbg, strlen(dbg));
        istat = ReadMugCards(satisfy->name, aradir, cards, err);
        if (istat == FAILURE)
        {
            sprintf(dbg, "CARD ERR: %s", err);
            m0sxstrce_(dbg, strlen(dbg));
        }
        else
        {
            sprintf(dbg, "failure reading file %s", satisfy->name);
            m0sxstrce_(dbg, strlen(dbg));
            sprintf(dbg, "ERR: %s", err);
            m0sxstrce_(dbg, strlen(dbg));
            continue;
        }
    }
    /* transform to comm protocol */
    nbytes = (IMG_DIR_LEN * 4) + 4 + (aradir[63] * MAX_CARD_LEN);
    control->reply_length += nbytes;
    swbyt4_(&nbytes, &one);
    sprintf(dbg, "sending %d bytes", nbytes);
    m0sxstrce_(dbg, strlen(dbg));
    m0sxsend_(&four, &nbytes);
    /* send four bytes of dummy */
m0sxsend(_&four, &dummy);

aradir[0] = satisfy->pos;

/*
 * send the directory
 */

(void)SendDir(aradir);

/*
 * send the comment cards
 */

if (aradir[63] != 0)

(void)SendCards(aradir, cards);

/*
 * increment the pointer in the list of files
 */

satisfy = satisfy->next;

}

return SUCCESS;

}
mcmugutil.c

1: #include <dirent.h>
2: #include <stdio.h>
3: #include <stdlib.h>
4: #include <strings.h>
5: #include "mug.h"
6: 
7: char dbg[MAX_ERR_LEN]; /* debug message */
8: /
9: 
10: 
11: int
12: SelectMugImages(char *dir, int bpos, int epos,
13: CRITERIA *request, FILELIST **satisfy, char *err)
14: {
15: 
16: char fullname[MAX_NAME_LEN]; /* full specified name of a file */
17: char name[MAX_NAME_LEN]; /* name of a file */
18: 
19: int ALL; /* tag to select all images in dir */
20: int aradir[IMG_DIR_LEN]; /* mcidas area directory */
21: int curpos=0; /* initial position number */
22: int istat; /* function return status */
23: int one=1; /* the number one (1) */
24: int pos=0; /* stored position number */
25: int strmatch; /* string comparison variable */
26: int valid_type; /* flag for defining a valid format */
27: 
28: DIR *dirlst=NULL; /* directory listing of files to test */
29: struct dirent *dirfile; /* struct containing file from readdir */
30: 
31: FILELIST *head=NULL; /* top of list of files in directory */
32: FILELIST *cur=NULL; /* top of list of files in directory */
33: 
34: FILELIST *rawlist=NULL; /* top of list of files in directory */
35: 
36: FILELIST *test=NULL; /* top of list of files to test */
37: FILELIST *timelist=NULL; /* top of list of time ordered files */
38: 
39: /*
40: * the integer representation of 'ALL' means
41: * get all the images in dir
42: */
43: 
44: ALL = lit("ALL ");
45: swbyt4(&ALL, &one);
46: 
47: /*
48: * read contents of the directory
49: */
50: 
51: dirlst = opendir(dir);
52: 
53: if (dirlst == NULL)
54: {
55: (void)strcpy(err, ERR_MG_DIR);
56: return ERR_STAT_DIR;
57: }
58: 
59: while ((dirfile = readdir(dirlst)) != NULL)
60: {
61: /*
62: * ignore special cases of filenames "." and ","
63: */
64: strmatch = strcmp(dirfile->d_name, ".");
66:     if (strmatch == 0) continue;
67: 68:     strmatch = strcmp(dirfile->d_name, ".");
69:     if (strmatch == 0) continue;
70: 71:     /*
72:      * make a fully specified name
73:      */
74: 75:     (void)strcpy(fullname, dir);
76:     (void)strcat(fullname, "/");
77: 78:     (void)strcat(fullname, dirfile->d_name);
79:     (void)strcat(fullname, "/0");
80: 81:     /*
82:      * determine the validity of the format
83:      */
84: 85:     valid_type = IsMugFormat(fullname);
86: 87:     if (valid_type == FAILURE)
88:        continue;
89: 90:     /*
91:      * increment the position
92:      */
93: 94:     curpos++;
95: 96:     /*
97:      * push the file onto the stack
98:      */
99: 100:    istat = PushFileByName(fullname, curpos, &rawlist);
101: 102:    if (istat == FAILURE)
103:       {
104:       (void)strcpy(err, ERR_MSG_MALLOC);
105:       return ERR_STAT_MALLOC;
106:       }
107: 108:    ifdef STDOUT_DEBUG
109:       cur = rawlist;
110:       (void)printf("---------- \n");
111:       while (cur != NULL)
112:       {
113:       (void)printf("pos: %d file: %s \n", cur->pos, cur->name);
114:       cur = cur->next;
115:       }
116:    #endif
117: 118: }
119: 120: (void)closedir (dirlist);
121: 122: /*
123:  * renumber the positions in the list
124:  * to reflect the alphabetical sorting
125:  */
126: 127: curpos = 0;
128: 129: while (cur != NULL)
130: 131: {
132:   curpos++;
133: 134:   /*
135:    * test the file against the CRITERIA
136:    */
istat = TestMugImage(cur->name, request, aradir, err);

if (istat == FAILURE)
    {
        cur = cur->next;
        continue;
    }

/*
 * push this file onto the stack
 */

istat = PushFileByName(cur->name, curpos, &head);

if (istat == FAILURE)
    {
        (void)strcpy(err, ERR_MSG_MALLOC);
        return ERR_STAT_MALLOC;
    }

cur = cur->next;
}

#ifdef STDOUT_DEBUG
    cur = head;
    (void)printf("-----SORTED-----

    while (cur != NULL)
        {
            (void)printf("pos: \%d file: \%s
", cur->pos, cur->name);
            cur = cur->next;
        }

    (void)printf("-------END-------

#endif

free(cur);
free(rlist);

/*
 * begin pulling the need files off and putting them
 * in a new list that will be returned
 */

/*
 * ALL files requested
 */

if (bpos == ALL)
    {
        *satisfy = head;
    }

/*
 * absolute file positions requested
 */

else if (bpos > 0)
    {
        while (head != NULL)
            {
                /*
                 * get name and pos off of stack
                 */
                istat = PopFile(name, &pos, &head);

                /*
                 * only use files that fit the
                 * requested positions
                 */

                if (pos < bpos || pos > epos) continue;
            }
/* put a file on the list to be returned */

istat = PushFileByName(name, pos, &test);

if (istat == FAILURE)
{
    (void)strcpy(err, ERR_MSG_MALLOC);
    return ERR_STAT_MALLOC;
}

*satisfy = test;

/* time relative file positions requested */

else if (bpos <= 0)
{
    cur = head;
    while (cur != NULL)
    {
        /* push the file onto stack ordered by time */
        istat = PushFileByTime(cur->name, cur->pos, &test);
        cur = cur->next;
    }

curpos = 1;
while (test != NULL)
{
    /* decrement time dependent position */
    curpos--;

    /* get name and pos off of stack */
    istat = PopFile(name, &pos, &test);

    /* only use files that fit the requested positions */

    if (curpos > bpos || curpos < epos) continue;

    /* put a file on the list to be returned */

    istat = PushFileByTime(name, pos, &timelist);

    if (istat == FAILURE)
    {
        (void)strcpy(err, ERR_MSG_MALLOC);
        return ERR_STAT_MALLOC;
    }
}
*satisfy = timlist;
}
return SUCCESS;
}

int PushMugFileByTime(char *name, int pos, FILELIST **list)
{
    /*
    * add a name and pos to a linked list sorted
    * in a chronological sense on time with the
    * most recent image at the head of the list
    * name - filename to add to list
    * pos - position number of name
    * list - list to which to add item
    */
    *success = 1
    *failure = 0
    */

    char err[MAX_ERR_LEN]; /* error message */
    int aradir[IMG_DIR_LEN]; /* mcdas area directory */
    int istat; /* function return status */
    int time; /* image time */
    FILELIST *after;
    FILELIST *insert;
    FILELIST *new=NULL;
    /* malloc a new node */
    new = (FILELIST *)malloc(sizeof(FILELIST));
    if (new == NULL)
    {
        return FAILURE;
    }
    /* fill the new struct with values */
    (void)strcpy(new->name, name);
    new->pos = pos;
    /* compute an image time in seconds since 1/1/72 */
    istat = ReadMugDir(name, aradir, err);
    time = sksecs_(&aradir[3], &aradir[4]);
    new->time = time;
    /* this is the first element in the list */
    if ((*list) == NULL)
    {
        (*list) = new;
        return SUCCESS;
    }
new->next = NULL;
*list = new;
return SUCCESS;
}
*/

/* insert the new element at the head
* if the time since 1/1/72 is greater than the
* current time, meaning this image is newer,
* we insert here
*/

if ((*list)->time < time) {
    new->next = *list;
    *list = new;
    return SUCCESS;
}

/* insert the new element at
the appropriate place in the list
*/
insert = *list;
while (1)
    after = insert->next;
    /*
    * end of list?
    */
    if (after == NULL)
        break;
    /*
    * if the time since 1/1/72 is greater than the
    * current time, meaning this image is newer,
    * we insert here
    */
    if (after->time < time)
        break;
    insert = after;
}

int IsMugFormat(char *name)
/*
* determine if a file is the MUG training course format
*/
* name - filename to test
* success 1
* failure 0
*/

McIDAS Developer/Operator Training
October 23, 1995

Applications Development in the ADDE
2-55
c 425: {
C 427: int valid;    /* flag for a valid file format */
C 429: /*
C 430: * the format test is based only
C 431: * on the name of the file. this makes
C 432: * the name of the file extrememly important
C 433: */
C 434: valid = strcmp(name, BASE_FILENAME);
C 436: if (valid == 0)
C 438: {
C 439: return FAILURE;
C 440: }
C 441: else
C 442: {
C 443: return SUCCESS;
C 444: }
C 445: }
C 446: }
C 447: /
C 448: */
C 449: ***************************************************************************/
C 450: int
C 451: ReadMugDir(char *name, int *aradir, char *err)
C 452: /*
C 453: * read a MUG training course file and create a
C 454: * corresponding mcdas area directory
C 455: * NOTE: The indices for aradir in this function
C 456: * are zero-based.
C 457: * name - filename to read
C 458: * aradir - mcdas area directory for "name"
C 459: * success 1
C 460: * failure 0
C 461: */
C 462: 
C 463: {
C 464: FILE *fd;
C 466: char line[MAX_LINE_LEN]; /* max byte length of data line */
C 468: char argdum; /* dummy for arg fetchers */
C 469: int argh=0; /* handle for arg fetchers */
C 470: int calen=512; /* byte length of cal block */
C 471: int i; /* loop variable */
C 472: int imtg; /* nominal image time, hhmms */
C 473: int istat; /* function return status */
C 474: int julday; /* julian date of image, yyyddd */
C 475: int navlen=512; /* byte length of nav block */
C 476: int nelems; /* number of elems in image */
C 477: int nlines; /* number of lines in image */
C 478: int parsed_len; /* byte length of arg block */
C 479: 
C 480: /* initialize array
C 481: */
C 482: for(i=0; i<IMG_DIR_LEN; i++)
C 483: aradir[i] = 0;
C 484: */
C 496:  * fixed value mcwords
C 497:  */
C 498:  */
C 499:  aradir[0] = 0;  /* always 0 */
C 500:  aradir[1] = 4;  /* always 4 */
C 501:  aradir[2] = 0;  /* mcidas ss number */
C 502:  aradir[5] = 1;  /* upper left line */
C 503:  aradir[6] = 1;  /* upper left element */
C 504:  aradir[10] = 1;  /* bytes per pixel */
C 505:  aradir[11] = 1;  /* line resolution */
C 506:  aradir[12] = 1;  /* element resolution */
C 507:  aradir[13] = 1;  /* number of bands */
C 508:  aradir[18] = 1;  /* bandmap */
C 509:  */
C 510:  */
C 511:  * byte offsets
C 512:  */
C 513:  */
C 514:  aradir[34] = IMG_DIR_LEN * 4;  /* byte offset to nav */
C 515:  aradir[62] = aradir[34] + navlen;  /* byte offset to cal */
C 516:  aradir[33] = aradir[62] + callen;  /* byte offset to dat */
C 517:  */
C 518:  */
C 519:  * source and cal type
C 520:  */
C 521:  */
C 522:  aradir[51] = lit_("VISR");  /* source type */
C 523:  aradir[52] = lit_("BRIT");  /* cal type */
C 524:  */
C 525:  */
C 526:  * open the file and get a file descriptor
C 527:  */
C 528:  */
C 529:  fd = fopen(name, "r");
C 530:  */
C 531:  */
C 532:  * did we get a valid file descriptor?
C 533:  */
C 534:  if (fd == FAILURE)
C 535:  {  return FAILURE;
C 536:  }
C 537:  }  }  
C 538:  
C 539:  */
C 540:  */
C 541:  * get first line of file and make it
C 542:  * available to the arg fetching routines
C 543:  */
C 544:  */
C 545:  (void) fgets(line, sizeof(line), fd);
C 546:  */
C 547:  argh = Mcargparse(line, NULL, &sparsed_len);
C 548:  */
C 549:  */
C 550:  * number of lines (mcword 9)
C 551:  */
C 552:  istat = Mcargint(arth, "NR", 1, 0, 999, -999, &nlines, &argdum);
C 553:  */
C 554:  aradir[8] = nlines;
C 555:  */
C 556:  */
C 557:  * number of elements (mcword 10)
C 558:  */
C 559:  */
C 560:  */
C 561:  istat = Mcargint(arth, "NC", 1, 0, 999, -999, &elems, &argdum);
C 562:  */
C 563:  aradir[9] = elems;
C 564:  */
C 565:  */
C 566:  * get first line of file and make it
C 567:  * available to the arg fetching routines
(/)
570: (void) fgets(line, sizeof(line), fd);
571: /*
572:  istat = Mcarga_free( argh );
573:  argh = Mcarga_parse(line, NULL, &parsed_len);
574:  istat = Mcarga_int( argh, "VD.AY", 1, 0, 999, -999, &julday, &argdum );
575:  aradir[3] = julday;
576:  istat = Mcarga_int( argh, "VT.IME", 1, 0, 999, -999, &imgtime, &argdum );
578:  (void) close((int) fd);
579:  istat = Mcarga_free( argh );
580:  return SUCCESS;  
581:  return SUCCESS;  
582: }  
583:}

***********************************************************************

601: int
602: MugNavImgToEarth(  char *name, float line, float elem,
603:  float *lat, float *lon, char *err)
604: /*
605:  * convert MUG image line/element into lat/lon
606:  * name - filename to read
607:  * line - input image line
608:  * elem - input image element
609:  * lat - output latitude of line/element
610:  * lon - output longitude of line/element
611:  * err - error string returned
612:  * success 1
613:  * failure 0
614: */
615: {  
616:  static char  lastname[MAX_NAME_LEN];  /* filename last time in func.*/
617:  float  dum;  /* dummy variable */
618:  int  istat;
619:  int  len;
620:  int  llflag;
621:  int  mcias_nav codicil;
622:  int  one=1;
623:  int  two=2;
624:  if ( strcmp(lastname, name) != 0 )
625:  {  
626:     /*
627:      * get a mcias nav codicil
628:      */
629: }
istat = ReadMugNavCod(name, navcod, l, l, &len);
if (istat == FAILURE)
{
    (void)strcpy(err, ERR_MSG_NONAV);
    return FAILURE;
}

/* initialize mcidas nav transforms */

istat = nvpref(&one, navcod);
if (istat != 0)
{
    (void)strcpy(err, ERR_MSG_MCIDAS_NAV);
    return FAILURE;
}

llflag = lit("LL ");
istat = nvlini(&two, &llflag);
if (istat != 0)
{
    (void)strcpy(err, ERR_MSG_MCIDAS_NAV);
    return FAILURE;
}

(void)strcpy(lastname, name);

istat = nvliae(&line, &elem, &dum, lat, lon, &dum);
if (istat != 0)
{
    (void)strcpy(err, ERR_MSG_IMG_TO_EARTH);
    return FAILURE;
}

     /* convert longitude to the mcidas west positive convention */

#ifndef WEST_POSITIVE
    (*lon) = (*lon) * -1;
#endif

return SUCCESS;

int
MugNavEarthToImg(char *name, float lat, float lon,
float *line, float *elem, char *err)

/* convert MUG image lat/lon into line/elem */

/* name - filename to read */
/* lat - input latitude */
/* lon - input longitude */
/* line - output image line of lat/lon */
/* elem - output image element of lat/lon */
/* err - error string returned */
/* success 1 */
/* failure 0 */
710: c
711: { c
712: static char  lastname[MAX_NAME_LEN]; /* filename last time in func.*/
713: c
714: float  dum; /* dummy variable */
715: c
716: int  istat; /* function status */
717: c
718: int  len; /* byte len of navcod */
719: c
720: int  lflag; /* variable for nvlini */
721: c
722: int  navcod[MAX_NAV_LEN]; /* mcidas nav codicil */
723: c
724: int  one=1; /* the number one (1) */
725: c
726: int  two=2; /* the number two (2) */
727: c
728: if (strcmp(lastname, name) != 0) c
729: {
730: /*
731: * get a mcidas nav codicil
732: */
733: c
734: istat = ReadMugNavCod(name, navcod, 1, 1, &len);
735: c
736: if (istat == FAILURE) c
737: {
738: (void)strcpy(err, ERR_MSG_NONAV);
739: return FAILURE;
740: c
741: c
742: /*
743: * initialize mcidas nav transforms
744: */
745: c
746: istat = nvprep_(&one, navcod);
747: c
748: if (istat != 0) c
749: {
750: (void)strcpy(err, ERR_MSG_MCIDAS_NAV);
751: return FAILURE;
752: c
753: c
754: lflag = lit("LL ");
755: istat = nvlini_(&two, &lflag);
756: c
757: if (istat != 0) c
758: {
759: (void)strcpy(err, ERR_MSG_MCIDAS_NAV);
760: return FAILURE;
761: c
762: /*
763: * convert longitude to the mcidas west positive convention
764: */
765: c
766: ifndef WEST_POSITIVE
767: lon = lon ** -1;
768: endif
769: c
770: istat = nvleas_(&lat, &lon, &dum, line, elem, &dum);
771: c
772: if (istat != 0) c
773: {
774: (void)strcpy(err, ERR_MSG_EARTH_TO_IMG);
775: return FAILURE;
776: c
777: c
778: return SUCCESS;
779: c
780: c
781: c
782: Applications Development in the ADDE
783: McIDAS Developer/Operator Training
784: 2-60  October 23, 1995
C 782: /
C 783: *****************************************/
C 784: int
C 785: ReadMugRes(char *name, float line, float elem, float *resx, float *resy, char *err)
C 786: /
C 787: /*
C 788: * calculate MUG image resolution at center point
C 789: *
C 790: * name - filename to read
C 791: * line - input image line
C 792: * elem - input image element
C 793: * resx - output x-resolution at center of image (km)
C 794: * resy - output y-resolution at center of image (km)
C 795: * err - error string returned
C 796: *
C 797: * success 1
C 798: * failure 0
C 799: *
C 800: */
C 801: /
C 802: 
C 803: static char lastname[MAX_NAME_LEN]; /* filename last time in func. */
C 804: 
C 805: int istat; /* function status */
C 806: 
C 807: double azimuth; /* directional azimuth (unused) */
C 808: 
C 809: double range1; /* range from center down */
C 810: 
C 811: double range2; /* range from down 1 line to over 1 element */
C 812: 
C 813: float dum; /* dummy variable */
C 814: 
C 815: float elem_p1; /* element plus one */
C 816: 
C 817: float lat; /* lat to measure distance */
C 818: 
C 819: float lat2; /* lat to measure distance */
C 820: 
C 821: float line_p1; /* line plus one */
C 822: 
C 823: float lon; /* lon to measure distance */
C 824: 
C 825: float lon2; /* lon to measure distance */
C 826: 
C 827: int len; /* byte len of navcod */
C 828: 
C 829: int l1flag; /* variable for nvlini */
C 830: 
C 831: int navcod[MAX_NAV_LEN]; /* mcidas nav codicil */
C 832: 
C 833: int one=1; /* the number one (1) */
C 834: 
C 835: int two=2; /* the number two (2) */
C 836: 
C 837: 
C 838: 
C 839: 
C 840: 
C 841: 
C 842: 
C 843: 
C 844: 
C 845: 
C 846: 
C 847: 
C 848: 
C 849: 
C 850: McIDAS Developer/Operator Training
C 851: }
C 852: 
C 853: Applications Development in the ADDE
C 854: October 23, 1995
C 855: 2-61
llflag = lit("LL ");

istat = nvlini(&two, &llflag);
if (istat != 0)
{
    (void)strcpy(err, ERR_MSG_MCIDAS_NAV);
    return FAILURE;
}

(void)strcpy(lastname, name);

/* convert longitude to the mcidas west positive convention */

#ifndef WEST_POSITIVE
    lon = lon * -1;
#endif

istat = nvlsae(&line, &elem, &dum, &lat, &lon, &dum);

if (istat != 0)
{
    (void)strcpy(err, ERR_MSG_IMG_TO_EARTH);
    return FAILURE;
}

/* move down one line and find the lat/lon */

line_p1 = line + 1.0;

istat = nvlsae(&line_p1, &elem, &dum, &lat2, &lon2, &dum);

if (istat != 0)
{
    (void)strcpy(err, ERR_MSG_IMG_TO_EARTH);
    return FAILURE;
}

/* find range from center to down one line */

istat = lltora(&lat, &lon, &lat2, &lon2, &range1, &azimuth);

if (istat != 0)
{
    (void)strcpy(err, ERR_MSG_MCIDAS_NAV);
    return FAILURE;
}

/* move over one element and find the lat/lon */

lat = lat2;
lon = lon2;
elem_p1 = elem + 1.0;

istat = nvlsae(&line_p1, &elem_p1, &dum, &lat, &lon, &dum);

if (istat != 0)
{
    (void)strcpy(err, ERR_MSG_IMG_TO_EARTH);
    return FAILURE;
}

/*
* find range from center to down one line
* /
   istat = ltora_(&lat, &lon, &lat2, &lon2, &range2, &azimuth);
   if (istat != 0)
      (void)strcpy(err, ERR_MSG_MCIDAS_NAV);
         return FAILURE;
   /*
   * res is average of the 2 ranges (resx = resy)
   */
   resy = (float)(range1 + range2) / 2.0;
   resx = *resy;
   #ifdef DEBUG
   (void)fprintf(dbg, "res: %f", *resy);
   m0sxtrce_(dbg, strlen(dbg));
   #endif
   return SUCCESS;
   
   int TestMugImage(char *name, CRITERIA *request, int *aradir, char *err)
   {
   /*
   * test an image against the specification
   */
   * criteria detailed in struct request
   * NOTE: No tests will be performed for CRITERIA values of -1
   *
   * name - filename to add to list
   * request - CRITERIA to test name
   * aradir - mcidas area directory returned for name
   * err - error string from ReadImgDir
   */
   * success 1
   * failure 0
   */
   
   int imgday; /* day of image */
   int imgss; /* satellite id for image */
   int imgtim; /* time of image */
   int istat; /* function status */
   /*
   * get a mcidas area directory for this image
   */
   istat = ReadMugDir(name, aradir, err);
   if (istat != FAILURE)
      return FAILURE;
   /*
   * test beginning and ending time
   */
   if (request->begtim >= 0)
C 997: {
C 998:   imgtim = aradir[4];
C 999: }
C 1000: if (imgtim < request->begtim || imgtim > request->endtim)
C 1001:   {
C 1002:       (void)strcpy(err, "Image time does not meet search criteria \n");
C 1003:       return FAILURE;
C 1004:   }
C 1005: /*
C 1006: * test beginning and ending day
C 1007: */
C 1008: /*
C 1009: if (request->begday == 0)
C 1010:   {
C 1011:       imgday = aradir[3];
C 1012:       if (imgday < request->begday || imgday > request->endday)
C 1013:         {
C 1014:             (void)strcpy(err, "Image day does not meet search criteria \n");
C 1015:             return FAILURE;
C 1016:         }
C 1017:   }
C 1018: /*
C 1019: * test beginning and ending mcidas ss number
C 1020: */
C 1021: /*
C 1022: if (request->begss == 0)
C 1023:   {
C 1024:       imgss = aradir[2];
C 1025:       if (imgss < request->begss || imgss > request->endss)
C 1026:         {
C 1027:             (void)strcpy(err, "Image satellite id does not meet search criteria \n");
C 1028:             return FAILURE;
C 1029:         }
C 1030:   }
C 1031: }*/
C 1032: return SUCCESS;
C 1033: }
C 1034: }
C 1035: }
C 1036: }
C 1037: }
C 1038: }
C 1039: }
C 1040: }
C 1041: }*/
C 1042: int
C 1043: ReadMugLine(char *name, READPARM *read, int band, short *buf, char *err)
C 1044: {
C 1045: /*
C 1046: * read a line of a MUG Training Course image
C 1047: * read
C 1048: * name - filename to read
C 1049: * read - READPARM struct containing read specs
C 1050: * band - band number of elements to read
C 1051: * buf - buffer containing image data
C 1052: * err - error string to return
C 1053: */
C 1054: * success 1
C 1055: * failure 0
C 1056: */
C 1057: const char *ardum; /* dummy for arg fetchers */
C 1058: Applications Development in the ADDE
C 1059: McIDAS Developer/Operator Training
C 1060: 2-64
C 1061: October 23, 1995
C 1068: char line[MAX_LINE_LEN];      /* max byte length of data line */
C 1069:
C 1070: static char lastname[MAX_NAME_LEN];  /* filename last time in func. */
C 1071:
C 1072: static int lastline=0;          /* last line of file read */
C 1073:
C 1074: static FILE *fd;               /* file descriptor */
C 1075:
C 1076: int argc;                       /* handle for arg fetchers */
C 1077: int begele;                     /* first element to read */
C 1078: int beglin;                     /* first line to read */
C 1079: int bufsiz;                     /* max # of lines to buffer */
C 1080: int des_len;                    /* byte size of dest pixel */
C 1081: int i;                          /* loop variable */
C 1082: int index;                      /* index into data array */
C 1083: int istat;                      /* function return status */
C 1084: int maxele;                     /* max number of elem in image */
C 1085: int maxlin;                     /* max number of lines in image */
C 1086: int numband;                    /* number of bands in image */
C 1087: int numele;                     /* number of elements to read */
C 1088: int numlin;                     /* number of lines to read */
C 1089: int parsed_len;                 /* byte length of arg block */
C 1090: int src_len;                    /* byte size of source pixel */
C 1091:
C 1092:
C 1093: double *val;                    /* data values on a line */
C 1094:
C 1095: if (strcmp(lastname, name) != 0)
C 1096: {
C 1097:     /*
C 1098:     * get a file descriptor
C 1099:     */
C 1100:     fd = fopen(name, "r");
C 1101: }
C 1102: /*
C 1103:     * did we get a valid file descriptor?
C 1104:     */
C 1105:
C 1106:
C 1107:     if (fd == NULL)
C 1108:         {
C 1109:             return FAILURE;
C 1110:         }
C 1111:     /*
C 1112:         * malloc space for the intermediate read
C 1113:         */
C 1114:
C 1115:     val = (double *)malloc(read->numele * sizeof(double));
C 1116:
C 1117:     if (val == NULL)
C 1118:         {
C 1119:             (void)strcpy(err, ERR_MSG_MALLOC);
C 1120:             return ERR_STAT_MALLOC;
C 1121:         }
C 1122:     /*
C 1123:         * read the first 3 lines of header
C 1124:         */
C 1125:
C 1126:     (void)fgets(line, sizeof(line), fd);
C 1127:     (void)fgets(line, sizeof(line), fd);
C 1128:     (void)fgets(line, sizeof(line), fd);
C 1129:     (void)fgets(line, sizeof(line), fd);
C 1130:     (void)fgets(line, sizeof(line), fd);
C 1131:     (void)fgets(line, sizeof(line), fd);
C 1132:     (void)fgets(line, sizeof(line), fd);
C 1133:     (void)fgets(line, sizeof(line), fd);
C 1134:     (void)fgets(line, sizeof(line), fd);
C 1135:     (void)fgets(line, sizeof(line), fd);
C 1136:     (void)fgets(line, sizeof(line), fd);
C 1137:     (void)fgets(line, sizeof(line), fd);
C 1138:     (void)fgets(line, sizeof(line), fd);
C 1139:

McIDAS Developer/Operator Training
October 23, 1995

Applications Development in the ADDE 2-65
C 1140:  
C 1141:  begele = read->begele;  
C 1142:  beglin = read->beglin;  
C 1143:  bufsiz = read->bufsiz;  
C 1144:  des_len = read->des_len;  
C 1145:  maxele = read->maxele;  
C 1146:  maxlin = read->maxlin;  
C 1147:  numband = read->numband;  
C 1148:  numele = read->numele;  
C 1149:  numlin = read->numlin;  
C 1150:  src_len = read->src_len;  
C 1151:  */
C 1152:  /*
C 1153:  * read any unwanted lines. this will
C 1154:  * only happen when a blowdown is requested
C 1155:  */
C 1156:  */
C 1157:  for(i=1; i<beglin-lastline; i++)
C 1158:  {
C 1159:     (void)fgets(line, sizeof(line), fd);
C 1160:  }
C 1161:  }
C 1162:  lastline = beglin;
C 1163:  */
C 1164:  /*
C 1165:  * get next line of file and make it
C 1166:  * available to the arg fetching routines
C 1167:  */
C 1168:  (void)fgets(line, sizeof(line), fd);
C 1169:  }
C 1170:  }
C 1171:  argh = Mcargparse(line, NULL, &parsed_len);
C 1172: }
C 1173: #ifdef DEBUG
C 1174:     (void)sprintf(dbg, "begele=td numele=td, argh=td", begele, numele, argh);
C 1175:     m0sxtrce_(dbg, strlen(dbg));
C 1176: #endif
C 1177: }
C 1178: }
C 1179: index = 0;
C 1180: for (i=begele; i<=begele+numele-1; i++)
C 1181: {
C 1182:     istat = Mcsargdbl(argh, ",", i, (double)0,
C 1183:     (double)999, (double)-999, &val[index], &argdum);
C 1184:     index++;
C 1185: }
C 1186: /*
C 1187: */
C 1188: /* "calibrate" the line
C 1189: */
C 1190: }
C 1191: for (i=0; i<=index-1; i++)
C 1192: {
C 1193:     buf[i] = 255 - (val[i] - 75);  
C 1194: }
C 1195: #ifdef DEBUG
C 1196:     (void)printf(dbg, "buf[%d] = %d", i, buf[i]);
C 1197:     m0sxtrce_(dbg, strlen(dbg));
C 1198: #endif
C 1199: }
C 1200: }
C 1201: }
C 1202: }
C 1203: istat = Mcargfree(argh);
C 1204: }
C 1205: return SUCCESS;
C 1206: }
C 1207: }
C 1208: }
C 1209: */
C 1210: */
C 1211: int
C 1212: ReadMugCalCod(char *name, int *cod, int *len)
C 1213: /
C 1214: /*
C 1215: * read a mcidas calibration codicil
C 1216: *
C 1217: * name - filename of image
C 1218: * cod - output mcidas cal codicil
C 1219: * len - byte length of output mcidas cal codicil
C 1220: *
C 1221: * success 1
C 1222: * failure 0
C 1223: *
C 1224: */
C 1225: /
C 1226: {
C 1227: int rc;  /* return code */
C 1228: int i;
C 1229: int
C 1230: i;
C 1231: /
C 1232: /*
C 1233: * get cal codicil
C 1234: */
C 1235: /
C 1236: *len=MAX_CAL_LEN;
C 1237: for(i=0; i<*len; i++)
C 1238: cod[i] = 0;
C 1239: /
C 1240: rc = SUCCESS;
C 1241: /
C 1242: return (rc);
C 1243: /
C 1244: }
C 1245: /
C 1246: /
C 1247: */
C 1248: /
C 1249: ReadMugNavCod(char *name, int *cod, int pt_line, int pt_elem, int *len)
C 1250: /
C 1251: /*
C 1252: * read a mcidas navigation codicil
C 1253: *
C 1254: * name - filename of image
C 1255: * cod - output mcidas nav codicil
C 1256: * pt_line - line number of upper left latitude
C 1257: * pt_elem - elem number of upper left longitude
C 1258: * len - byte length of output mcidas nav codicil
C 1259: *
C 1260: * success 1
C 1261: * failure 0
C 1262: *
C 1263: */
C 1264: /
C 1265: {
C 1266: /
C 1267: const char *argdum;  /* dummy for arg fetchers */
C 1268: /
C 1269: char line[MAX_LINE_LEN]; /* max byte length of data line */
C 1270: /
C 1271: static FILE *fd;  /* file descriptor */
C 1272: /
C 1273: int argh=0;  /* handle for arg fetchers */
C 1274: int i;  /* loop variable */
C 1275: int istat;  /* function return status */
C 1276: int parsed_len;  /* byte length of arg block */
C 1277: int rc;  /* return code */
C 1278: /
C 1279: double latinc;  /* degree spacing of latitude */
C 1280: double loninc;  /* degree spacing of longitude */
C 1281: double uoffset;  /* upper left latitude */
C 1282: double  ullon;         /* upper left longitude         */
C 1283: C 1284: /*
C 1285: * open the file and get a file descriptor
C 1286: */
C 1287: C 1288: fd = fopen(name, "r");
C 1289: C 1290: /*
C 1291: * did we get a valid file descriptor?
C 1292: */
C 1293: C 1294: if (fd == FAILURE)
C 1295: {
C 1296:     return FAILURE;
C 1297: }
C 1298: C 1299: /*
C 1300: * get first line of file and make it
C 1301: * available to the arg fetching routines
C 1302: */
C 1303: C 1304: (void)fgets(line, sizeof(line), fd);
C 1305: C 1306: argh = Mcargparse(line, NULL, &parsed_len);
C 1307: C 1308: /*
C 1309: * number of lines (mcword 9)
C 1310: */
C 1311: C 1312: istat = Mcargdbl(argh, "LAT", 2, (double)0, (double)999, (double)-999,
C 1313:        &ullat, &argnum);
C 1314: C 1315: istat = Mcargdbl(argh, "LON", 2, (double)0, (double)999, (double)-999,
C 1316:        &ullon, &argnum);
C 1317: C 1318: istat = Mcargdbl(argh, "LAT.NC", 1, (double)0, (double)999, (double)-999,
C 1319:        &latinc, &argnum);
C 1320: C 1321: istat = Mcargdbl(argh, "LOI.NC", 1, (double)0, (double)999, (double)-999,
C 1322:        &loninc, &argnum);
C 1323: C 1324: /*
C 1325: * build a rectilinear nav codicil
C 1326: */
C 1327: C 1328: *len=MAX_NAV_LEN;
C 1329: C 1330: /*
C 1331: * initialize to zero
C 1332: */
C 1333: C 1334: for(i=0; i<=(*len)-1; i++)
C 1335:     cod[i] = 0;
C 1336: C 1337: cod[0] = lit_("RECT");
C 1338: cod[1] = pt_line;
C 1339: cod[2] = (int)(ullat * 10000.0);
C 1341: cod[4] = (int)(ullon * 10000.0);
C 1342: cod[5] = (int)(latinc * 10000.0);
C 1343: cod[6] = (int)(loninc * 10000.0);
C 1345: cod[8] = EARTH_ECCENTRICITY;
C 1346: cod[9] = 0;
C 1347: C 1348: #ifdef WEST_POSITIVE
C 1349: cod[10] = 1;
C 1350: #else
C 1351: cod[10] = -1;
C 1352: #endif
C 1353:
C 1354: #ifndef DEBUG
C 1355:  
C 1356:  
C 1357:  
C 1358:  
C 1359:  
C 1360:  
C 1361:  
C 1362:  
C 1363:  
C 1364:  
C 1365:  
C 1366:  
C 1367:  
C 1368:  
C 1369:  
C 1370:  
C 1371:  
C 1372:  
C 1373:  
C 1374:  
C 1375:  
C 1376:  
C 1377:  
C 1378:  
C 1379:  
C 1380:  
C 1381:  
C 1382:  
C 1383:  
C 1384:  
C 1385:  
C 1386:  
C 1387:  
C 1388:  
C 1389:  
C 1390:  
C 1391:  
C 1392:  
C 1393:  
C 1394:  
C 1395:  
C 1396:  
C 1397:  
C 1398:  
C 1399:  
C 1400:  
C 1401:  
C 1402:  
C 1403:  
C 1404:  
C 1405:  
C 1406:  
C 1407:  
C 1408:  
C 1409:  
C 1410:  
C 1411:  
C 1412:  
C 1413:  
C 1414:  
C 1415:  
C 1416:  
C 1417:  
C 1418:  
C 1419:  
C 1420:  
C 1421:  
C 1422:  
C 1423:  
C 1424:  
C 1425:
add = AddCard(one_card, MAX_CARD_LEN, cards, num_cards);
if (add == FAILURE)
    { (void)strcpy(err, ERR_MSG_MALLOC);
      return FAILURE;
    }
    (void)sprintf(one_card, CEN_LON_CARD, (lon);
    num_cards++; 
    add = AddCard(one_card, MAX_CARD_LEN, cards, num_cards);
if (add == FAILURE)
    { (void)strcpy(err, ERR_MSG_MALLOC);
      return FAILURE;
    }
#endif

/* center point failure */
    return FAILURE;

/* get the resolution at the center point of the image */
    istat = ReadMugRes(name, line, elem, &resx, &resy, err);
    if (istat != FAILURE)
        {
        /* build cards for the resolution */
        (void)sprintf(one_card, RES_X_CARD, (int)(resx+0.5));
        num_cards++; 
        add = AddCard(one_card, MAX_CARD_LEN, cards, num_cards);
if (add == FAILURE)
    { (void)strcpy(err, ERR_MSG_MALLOC);
      return FAILURE;
    }
    (void)sprintf(one_card, RES_Y_CARD, (int)(resy+0.5));
    num_cards++; 
    add = AddCard(one_card, MAX_CARD_LEN, cards, num_cards);
if (add == FAILURE)
    { (void)strcpy(err, ERR_MSG_MALLOC);
      return FAILURE;
    }
        }
    else
    { /* resolution failure */
      return FAILURE;
    }

C 1498: /*
C 1499: * get the valid unit type for this image
C 1500: */
C 1501: istat = ReadUnits(aradir, cards, &num_cards, err);
C 1504: /*
C 1506: * change mcword 64 in the area
C 1507: * directory to reflect the number
C 1508: * of comment cards
C 1509: */
C 1510: aradir[63] = aradir[63] + num_cards;
C 1512: C 1513: return SUCCESS;
C 1514: C 1515: }
mug.h

/*
This is the main include file for the 1995 MUG Training Course ADDE server.
It contains data structures that are used by the image servers.
*/

#include <unistd.h>
#include "mcdas.h" /* McIDAS include file */
#include "servacct.h" /* ADDE include file */

/*
return status
*/
#define FAILURE 0
#define SUCCESS 1

/*
error handling
*/

#define ERR_STAT_DIR -2
#define ERR_MSG_DIR "Unable to read contents of directory \n"
#define ERR_STAT_MALLOC -1
#define ERR_MSG_MALLOC "Unable to allocate memory \n"
#define ERR_STAT_BADLINE -41
#define ERR_MSG_BADLINE "Unable to read image line \n"
#define ERR_STAT_BLANK_IMG -47
#define ERR_MSG_BLANK_IMG "There's nothing to see here. \n"
#define ERR_STAT_NOIMG -51
#define ERR_MSG_NOIMG "There are no images out here. \n"
#define ERR_STATselectAll -50
#define ERR_MSG_SELECT "Unable to initialize navigation \n"
#define ERR_MSG_EARTH_TO_IMG "Unable to transform lat/lon to line/element \n"
#define ERR_MSG_FORMAT "Unable to read this data format \n"
#define ERR_MSG_IMG_TO_EARTH "Unable to transform line/element to lat/lon \n"
#define ERR_MSG_MCIDAS_NAV "Error in McIDAS navigation \n"
#define ERR_MSG_UNKNOWN_SAT "Calibration units unknown for this satellite \n"

/*
max lengths
*/

#define IMG_DIR_LEN 64 /* len of image directory */
#define MAX_CAL_LEN 512 /* byte len of calibration codicil */
#define MAX_CARD_LEN 80 /* len of a single comment card */
#define MAX_DIR_LEN 1024 /* len of directory where data lives */
#define MAX_ERRMSG_LEN 72 /* byte len of error msg back to client */
#define MAX_ERR_LEN 256 /* len of error string */
#define MAX_LINE_LEN 10000 /* max. len of a line in the image file */
#define MAX_NAME_LEN 1024 /* max. len of a image path and name */
#define MAX_NAV_LEN 512 /* byte len of calibration codicil */
#define MAX_NUM_BAND 32 /* max. number of bands in an image */
#define MAX_NUM_CARDS 250 /* max. number of mciadas comment cards*/
/* default values */
#define DEF_NUM_ELEMS 640 /* default num of elements to send */
#define DEF_NUM_LINES 480 /* default num of lines to send */
#define READ_BUFFER_SIZE 1 /* # of image lines to buffer on read */
#define RES_AT_CENTER 1 /* resolution in km at center of image */
#define BASE_FILENAME "MRF" /* MUG format files name convention */
/* nav parameters */
#define EARTH_RAD_METERS 6378388 /* earth radius in meters */
#define EARTH_ECCENTRICITY 81992.0 /* earth eccentricity * 1e6 */
/* longitude convention */
/* comment this line out if a west negative convention is desired */
#define WEST_POSITIVE 1
/* comment card strings */
#define CEN_LAT_CARD "Center latitude = %f"
#define CEN_LON_CARD "Center longitude = %f"
#define RES_X_CARD "Longitude resolution (km) = %d"
#define RES_Y_CARD "Latitude resolution (km) = %d"
#define VALID_UNIT_CARD "Valid calibration unit for band %d = %s \"%s\"
/* CRITERIA holds the search parameters for */
/* image selection */
typedef struct CRITERIA_
{
    int begday;
    int enday;
    int begtim;
    int endtim;
    int begss;
    int endss;
} CRITERIA;
/* FILELIST holds the a list of files */
/* to be compared to the parameters in CRITERIA */
typedef struct FILELIST_
{
    char name[MAX_NAME_LEN];
    int pos;
    int time;
} FILELIST;
typedef struct READPARM {
    char src_type[4];
    char des_unit[4];
    char src_unit[4];
    int begele;
    int begin;
    int bufsiz;
    int des_len;
    int elem_res;
    int line_res;
    int maxele;
    int maxlin;
    int minele;
    int minlin;
    int numband;
    int numele;
    int numlin;
    int src_len;
    int ul_elem;
    int ul_line;
} READPARM;

/*
mcidas prototypes
*/

char *clit_(int *);

int lit_(char *);

int lltora_(float *, float *, float *, float *, double *, double *);

int m0resred_(char *, int *, int *, int *);

int m0sxxsend_(int *, int *);

int m0sxtrzc_(char *, int *);

int nvlneas_(float *, float *, float *, float *, float *, float *);

int nvlini_(int *, int *);

int nvprep_(int *, int *);

int nvlisae_(float *, float *, float *, float *, float *, float *);

int skssecs_(int *, int *);

MUG interface prototypes
*/

AddCard(char *, int, int *, int);

CheckImgBounds (READPARM *, int, int);

IsMugFormat(char *);

MugNavEarthToImg(char *, float, float, float, float, float, char *);

MugNavImgToEarth(char *, float, float, float, float, char *);

PopFile(char *, int *, FILELIST **);

PushFile(char *, int, FILELIST **);

PushFileByName(char *, int, FILELIST **);

PushMugFileByTime(char *, int, FILELIST **);

ReadMugCards(char *, int *, int *, char *);

ReadMugCalcCod(char *, int *, int *, char *);

ReadMugRes(char *, float, float, float, float, char *);

ReadMugDir(char *, int, char *);

ReadMugLine(char *, char *, float, short, char *);

ReadMugNavCod(char *, int *, short, int, int *);

ReadUnits(int *, int *, int *, char *);

SelectMugImages(char *, int, int, CRITERIA *, FILELIST *, char *);

SendCards(int *, int *);

SendCod(char *, int *);

SendDir(int *);

SendLine(short *, int);

SendZeros(short *, float, int);

TestMugImage(char *, CRITERIA *, int, char *);
This is called by exec from subserv.c
It reconstructs the server environment, and continues the service

Note: the variable Sname is not defined in this file
It should be defined in the -DSname=value field on the
compile. value will be interpreted as the name of the
application function that will perform the work. Eg aget_

#include <memory.h>
#include <string.h>
#include <stdlib.h>
#include "mciadas.h"
#include "servacot.h"
/* turn off trace */
extern int trace;

#define __EMX__ /* f2c compatibility for os/2 */
int xargc;
char **xargv;
#undef /*__EMX__ */

void main( int argc, char *argv[])
{
    servacot request_block; /* control block for transaction */
    char  dbg[180];
    int  istat;
    int  parsed_len;

    #ifdef __EMX__ /* f2c compatibility for os/2 */
    xargc=xargc;
    xargv=xargv;
    #endif /*__EMX__ */

    /*
      We do not validate the transaction user and project.
      That should have been done by the main server previously */
    trace_ = 0;

    #ifdef DEBUG
    trace_ = 1;
    #endif
    (void)strcpy(dbg, "starting subserv");
    m0sxtrce_(dbg, strlen(dbg));

    /* initialize McIDAS environment */
{ 
    int i;
    initblok_(&i);
    if (i != 0) return;
}

/* construct request block */
memset(&request_block, 0, sizeof(request_block));

* (int *) &request_block.server_address = atoi(argv[1]);
request_block.server_port = atoi(argv[2]);
* (int *) request_block.user = atoi(argv[3]);
request_block.project = atoi(argv[4]);
* (int *) request_block.transaction = atoi(argv[5]);
request_block.input_length = atoi(argv[6]);

/* put at least part of the request into the rb */
strncpy(request_block.text, argv[7],
        strlen(argv[7]) < sizeof(request_block.text) ?
        strlen(argv[7]) : sizeof(request_block.text));

/* initialize log fields */
m0xlogi_(&request_block);

/* parse the command */
istat = Mcfgfree(0);
istat = M0cmdput(M0cmdparse(argv[7], &parsed_len));

/* do the guts */
Sname(&request_block);

/* termination */
m0xdone_(&request_block);
servutil.h

/****
**** servutil.h
**** This is the main include file for the McIDAS
**** ADDE server utilities.
****
F 12: #include <stdio.h>
F 13: #include <stdlib.h>
F 14: #include <strings.h>
F 15: #include "mcidas.h"

F 16: */
F 17: */
F 18: return status
F 19: */
F 20: */
F 21: #define FAILURE 0
F 22: #define SUCCESS 1
F 23: */
F 24: */
F 25: error handling
F 26: */
F 27: */
F 28: #define ERR_STAT_DIR "Unable to read contents of directory \n"
F 29: #define ERR_MSG_DIR -2
F 30: */
F 31: #define ERR_STAT_MALLOC -1
F 32: #define ERR_MSG_MALLOC "Unable to allocate memory \n"
F 33: */
F 34: #define ERR_MSG_UNKNOWN_SAT "Calibration units unknown for this satellite \n"
F 35: */
F 36: */
F 37: max lengths
F 38: */
F 39: */
F 40: #define IMG_DIR_LEN 64 /* len of image directory */
F 41: #define MAX_CARD_LEN 80 /* len of a single comment card */
F 42: #define MAX_DIR_LEN 1024 /* len of directory where data lives */
F 43: #define MAX_ERRMSG_LEN 72 /* byte len of err msg back to client */
F 44: #define MAX_ERR_LEN 256 /* len of error string */
F 45: #define MAX_NAME_LEN 1024 /* max. len of a image path and name */
F 46: #define MAX_NUM_BAND 32 /* max. number of bands in an image */
F 47: #define MAX_NUM_CARDS 250 /* max. number of mcidas comment cards*/
F 48: */
F 49: */
F 50: comment card strings
F 51: */
F 52: */
F 53: #define VALID_UNIT_CARD "Valid calibration unit for band %d = %s \"%s\"
F 54: */
F 55: */
F 56: CRITERIA holds the search parameters for
F 57: image selection
F 58: */
F 59: typedef struct CRITERIA_
F 60: {
F 61: int begband;
F 62: int endband;
F 63: int begday;
F 64: int endday;

typedef struct FILELIST_
    char name[MAX_NAME_LEN];
    int pos;
    int time;
    struct FILELIST_ *next;
} FILELIST;

typedef struct READPARM_
    char src_type[4];
    char des_unit[4];
    char src_unit[4];
    int begele;
    int beginin;
    int bufsiz;
    int des_len;
    int elem_res;
    int line_res;
    int maxele;
    int maxlin;
    int minele;
    int minlin;
    int numbnd;
    int numele;
    int numlin;
    int src_len;
    int ul_elem;
    int ul_line;
} READPARM;

/* mcidas routines */

int m0xsxsend(int *, int *);
int m0xextrce(char *, int);

/* server utility interface prototypes */

int AddCard(char *, int, int *, int, int);
int CheckImgBounds(READPARM *, int, int);
int PopFile(char *, int *, FILELIST **);
int PushFile(char *, int, FILELIST **);
int PushFileByName(char *, int, FILELIST **);
int ReadCodLengths(char *, int *, int *, char *);
int ReadUnits(int *, int *, int *, int *, char *);
int SendCards(int *, int *);
int SendCod(int *, int);
int SendDir(int *);
int SendLine(int *, int *);
int SendZeros(int *, int *);
mugarea.pgm

G 1: C ? MUGAREA -- Manipulate area data received by server for MUG demo
G 2: C ?
G 3: C ? Parameters:
G 4: C ?
G 5: C ? source: source dataset name, contains areas
G 6: C ? pos1: position of 1st area to use (default, most recent)
G 7: C ? oper: mathematical operation (ADD, SUB, AVG) are valid entries
G 8: C ? pos2: position of 2nd area to use (default, next most recent)
G 9: C ? destination.pos | destination dataset and position
G 10: C ? Keywords:
G 11: C ? SIZE = nlines nleles : Size of area to get from server
G 12: C ?
G 13: SUBROUTINE MAIN0
G 14: IMPLICIT NONE
G 15: C
G 16: C
G 17: C ------- external functions
G 18: character*12 cfr  ! left-justified integer->string
G 19: integer iftok  ! make a character string an integer
G 20: integer lsdgch ! is the character string digits characters?
G 21: integer mcacal ! get the calibration
G 22: integer mcacou ! write the comment cards
G 23: integer mcanav ! get the navigation
G 24: integer mcanav ! get the navigation
G 25: integer mcacrd ! read the comment cards
G 26: integer mcsaln ! get a line of data from pipe
G 27: integer mcasrt ! get sort conditions to pass to server
G 28: integer mcmdint ! get integer from command line
G 29: integer mcmdkey ! validate command line entries
G 30: integer mcmdnum ! number of entries with keyword
G 31: integer mcmdstr ! get string from command line
G 32: integer mcmdstr ! get string from command line
G 33: integer mcasrt ! get sort conditions to pass to server
G 34: integer mcmdnum ! get number of positions in dataset
G 35: C ------- parameter
G 36: integer MAXCARD
G 37: parameter (MAXCARD = 500) ! max number of comment cards
G 38: integer NLINMAX
G 39: parameter (NLINMAX=1000) ! max number of lines
G 40: integer NELEM0
G 41: parameter (NELEM0=1000) ! max number of elements
G 42: C ------- local variables
G 43: character*12 cbday ! begin day
G 44: character*12 ceday ! end day
G 45: character*12 cpwrd ! position of destination dataset
G 46: character*12 oper_str ! operation to perform on area1&2
G 47: character*40 sort1(20) ! sort strings to pass to server, 1st position
G 48: character*40 sort2(20) ! sort strings to pass to server, 2nd position
G 49: character*40 psorts(20) ! sort string for putting
G 50: character*40 dname ! destination dataset name
G 51: character*40 sname ! store source name in here
G 52: character*40 name ! source dataset name, destination storage
G 53: integer buffer1(nelem0) ! buffer for ADDE read/write
G 54: integer buffer2(nelem0) ! buffer for ADDE read/write
G 55: integer buffer3(nelem0) ! buffer for ADDE read/write
G 56: integer cards1(MAXCARD) ! area1 comment cards
G 57: integer cards2(MAXCARD) ! area2 comment cards
G 58: integer cards3(MAXCARD) ! area3 comment cards
G 59: integer dirl (64) ! area directory for 1st area

McIDAS Developer/Operator Training
October 23, 1995

Applications Development in the ADDE 2-79
integer dir2(64) ! area directory for 2nd area
integer dir3(64) ! area directory for 3rd area
integer dpos ! destination dataset position
integer h1 ! handle for mcaget
integer h2 ! handle for mcaget
integer h3 ! handle for mcaget!
integer i ! loop index
integer ica(10000) ! call block, area 1
integer ica1(10000) ! call block, area 2
integer ica2(10000) ! call block, area 3
integer iele ! number of elements to fetch
integer ilin ! number of lines to fetch
integer inav1(1024) ! nav block, area 1
integer inav2(1024) ! nav block, area 2
integer inav3(1024) ! nav block, area 3
integer iread ! number of directories read
integer iret ! return code
integer iret1 ! return code
integer iret2 ! return code
integer ix ! loop bound
integer j ! loop bound
integer jy ! loop bound
integer npos ! number of positions in dataset
integer pos1 ! 1st dataset position
integer pos2 ! 2nd dataset position
integer prefix1(250) ! prefix for area 1
integer prefix2(250) ! prefix for area 2
integer prefix3(250) ! prefix for area 3
integer neles ! number of elements
integer nlines ! number of lines
integer nsor1 ! number of sort conditions, 1st posn
integer nsor2 ! number of sort conditions, 2nd posn
integer nsorp ! number of put sort strings
integer xdir1(65) ! expanded area directory for 1st area
integer xdir2(65) ! expanded area directory for 2nd area
integer xdir3(65) ! expanded area directory for 3rd area
integer numkeys ! number of valid keyword entries
parameter(NUMKEYS=10)
character*12 key_words(NUMKEYS) ! valid keywords
data key_words/"AUX", "BAN.D", "CAL", "DAY", "LOC.ATE", "MAG.NIFY", "SIZ.E", "SUE", "TIM.E", "DPO.SITION"/

nsor1 = 0
nsor2 = 0
nsorp = 0

C--------check to see if keywords on command line are valid

if (mcmdkey(numkeys,key_words) .lt. 0) then
  call edest('Ambiguous, illegal, or invalid keywords',0)
goto 2000
endif
C--------read in the source dataset name
if (mcmdstr( ' ',1,'0',sname) .lt. 0) goto 2000
if (sname .eq. '0') then
call edest('Source dataset must be entered',0)
goto 2000
endif
if (isdgh(sname) .eq. 1) then
call edest('Invalid source dataset name: //sname,0)
goto 2000
endif
C--------read in the destination dataset name
if (mcmdstr( ' ',5,'0',name) .lt. 0) goto 2000
if (name .eq. '0') then
  call edest('Destination dataset must be entered',0)
goto 2000
endif
if (isdgch(name) .eq. 1) then
  call edest('Invalid destination dataset name: //name,0')
goto 2000
endif
C-----Separate the dataset name from the position
C
C
C
dname = name(1:index(name,('.'))-1)
cposd = name(index(name,'.')+1):
C-----cposd must be positive integer, do not allow anything else..
C
C
if (cposd(1:1) .lt. '1' .or. cposd(1:1) .gt. '9') then
  call edest('Invalid position specified in destination //
  & 'datast parameter.',0)
  call edest('Must be a positive integer',0)
goto 2000
endif
C-----make sure that cposd is a number
C
if (isdgch(cposd) .ne. 1) then
  call edest('Invalid position specified in destination //
  & 'datast parameter.',0)
  call edest('Must be a positive integer',0)
goto 2000
endif
C-----read in the bounds of the destination dataset name
C
C
npos = mdsnum(dname,'AREA')
if (npos .lt. 0) then
  if (npos .eq. -1) then
    call edest('Unable to resolve dataset dname= //dname,0)
goto 2000
  else if (npos .eq. -2) then
    call edest('Unable to resolve dataset dname= //dname,0)
goto 2000
  endif
endif
C
C
C
if (dpos .gt. npos) then
  call edest('Destination dataset position ('//cposd//') '
  & 'exceeds dataset limits ('//cfr(npos)//'),0)
goto 2000
endif
C-----read in the bounds for the source dataset sname
C
C
npos = 8
if (npos .lt. 0) then
  if (npos .eq. -1) then
    call edest('Unable to resolve dataset sname= //sname,0)
G 211:      goto 2000
G 212:   elseif (npos .eq. -2) then
G 213:     call edest('Unable to resolve dataset sname= '/sname,0)
G 214:     goto 2000
G 215:   endif
G 216:   call edest('Bad mcdsnum return code: '/cfns(npos),0)
G 217:   goto 2000
G 218:   endif
G 219: C
G 220: C------Read in the first position
G 221: C
G 222:   if (mccmdint(' ','2,'First Dataset Position',0,
G 223:     &    -npos,npos,pos1) .lt. 0) goto 2000
G 224: C
G 225: C------Read in the second position
G 226: C
G 227:   if (mccmdint(' ','4,'Second dataset position',-1,
G 228:     &    -npos,npos,pos2) .lt. 0) goto 2000
G 229: C
G 230: C------read in the operation to perform
G 231: C
G 232:   if (mccmdstr(' ','3','ADD',oper_str) .lt. 0) then
G 233:     call edest('Invalid operator read in from command line:'
G 234:     &    '/oper_str,0)
G 235:     call edest('Use ADD, SUB, or AVG',0)
G 236:     goto 2000
G 237:   endif
G 238: C
G 239: C------make sure operation entered is valid
G 240: C
G 241:   if (oper_str(1:3) .ne. 'ADD' .and. oper_str(1:3) .ne. 'SUB'
G 242:     &    .and. oper_str(1:3) .ne. 'AVG') then
G 243:     call edest('Invalid operator: '/oper_str,0)
G 244:     call edest('Use ADD, SUB, or AVG',0)
G 245:     goto 2000
G 246:   endif
G 247: C
G 248: C------read in the DAY requested
G 249: C
G 250:   if (mccmdnum('DAY') .gt. 0) then
G 251:   endif
G 252: G 253:   if (mccmdstr('DAY',1,'X',cbday) .lt. 0) goto 2000
G 254:   if (mccmdstr('DAY',2,cbday,ceday) .lt. 0) goto 2000
G 255: C
G 256:   nsorts1 = nsorts1 + 1
G 257:   sort1(nsorts1) = 'DAY '/cbday//ceday
G 258:   nsorts2 = nsorts2 + 1
G 259:   sort2(nsorts2) = 'DAY '/cbday//ceday
G 260:   endif
G 261: C
G 262:   nsorts1 = 0
G 263:   nsorts2 = 0
G 264: C
G 265:   if (mcasort(nsorts1,sorts1,1) .lt. 0) then
G 266:     call edest('Failed to return standard sort parms',0)
G 267:     call edest('for first position',0)
G 268:     goto 2000
G 269:   endif
G 270: C
G 271:   if (mcasort(nsorts2,sorts2,1) .lt. 0) then
G 272:     call edest('Failed to return standard sort parms',0)
G 273:     call edest('for second position',0)
G 274:     goto 2000
G 275: C
G 276: C
G 277: C------add position to the sort strings
G 278: C
G 279:   nsorts1 = nsorts1 + 1
G 280:   sorts1(nsorts1) = 'POS '/cfns(pos1)
G 281:   nsorts2 = nsorts2 + 1
G 282:   sorts2(nsorts2) = 'POS '/cfns(pos2)
G 283: C------get the Size
G 284: C
G 285: C
G 286: if (mccmdint('SIZ.E',1,'Number of lines',145,1,1000,lin)
G 287: & .lt. 0) goto 2000
G 288: if (mccmdint('SIZ.E',2,'Number of eles',288,1,1000,iele)
G 289: & .lt. 0) goto 2000
G 290: nsorstl = nsorstl + 1
G 291: sortsl(nsortsl) = 'SIZE '//cfr(lin)//cfr(iele)
G 292: nsorsts2 = nsorsts2 + 1
G 293: sortst(nsortst) = 'SIZE '//cfr(lin)//cfr(iele)
G 294: C
G 295: C
G 296: C----make the call to mcaget to start the serving of position 1
G 297: C
G 298: name = sname
G 299: iret = mcaget(name,nsorsts1,sorts1,'TEMP','I4',
G 300: & NELEMAX*4,1,dirl1,h1)
G 301: if (iret .lt. 0) then
G 302: call edest('Fail in mcaget for position 1',0)
G 303: goto 2000
G 304: endif
G 305: C
G 306: C
G 307: C----make the call to mcaget to start the serving of position 2
G 308: C
G 309: name = sname
G 310: iret = mcaget(name,nsorsts2,sorts2,'TEMP','I4',
G 311: & NELEMAX*4,1,dirl2,h2)
G 312: if (iret .lt. 0) then
G 313: call edest('Fail in mcaget',0)
G 314: goto 2000
G 315: endif
G 316: C
G 317: C
G 318: C------Make sure the two areas are of identical size and shape
G 319: C
G 320: do 600 ix = 7,15
G 321: if (dirl1(ix) .ne. dir2(ix)) then
G 322: call edest('Area mismatch at position '//cfr(ix),0)
G 323: call edest('Area 1 value: '//cfr(dirl1(ix)),0)
G 324: call edest('Area 2 value: '//cfr(dir2(ix)),0)
G 325: goto 2000
G 326: endif
G 327: continue
G 328: ix = 600
G 329: C
G 330: C
G 331: C
G 332: C------get the nav and call blocks
G 333: C
G 334: if (mcanav(h1,inav1) .ne. 0) goto 2000
G 335: if (mcacal(h1,ical1) .ne. 0) goto 2000
G 336: if (mcanav(h2,inav2) .ne. 0) goto 2000
G 337: if (mcacal(h2,ical2) .ne. 0) goto 2000
G 338: iread = 1
G 339: C
G 340: C
G 341: C
G 342: C
G 343: C
G 344: C------start the server transaction to put
G 345: C
G 346: C
G 347: if (psorts(l) = 'POS '//cposd
G 348: iret = mcput(dname,1,psorts,dirl1,inav1,ical1)
G 349: if (iret .ne. 0) then
G 350: call edest('Error in setting up put',0)
G 351: goto 2000
G 352: endif
G 353: C
G 354: C
C-------read in the lines of imagery from the two areas
C
1000 continue
iret1 = mcain(h1,buffer1)

if (iret1 .lt. 0) then
   call edest('Fail in mcain for area1 at line '//' & cfr(irread),0)
go to 2000
endif

if (mcafx(h1, prefix1) .lt. 0) then
call edest('Prefix Read in area 1 has failed',0)
go to 2000
endif

iret2 = mcain(h2,buffer2)

if (iret2 .lt. 0) then
   call edest('Fail in mcain for area2 at line '//' & cfr(irread),0)
go to 2000
endif

if (mcafx(h2, prefix2) .lt. 0) then
call edest('Prefix Read in area 2 has failed',0)
go to 2000
endif

if (iret1 .eq. 0 .and. iret2 .eq. 0) then
C------got something from pipe that is area data, perform operation
C
do 1200 ix = 1, dir1(10)
if (oper_str(1:3) .eq. 'ADD') then
   buffer3(ix) = min0(buffer1(ix) + buffer2(ix),255)
elseif (oper_str(1:3) .eq. 'SUB') then
   buffer3(ix) = max0(buffer1(ix) - buffer2(ix),0)
elseif (oper_str(1:3) .eq. 'AVG') then
   buffer3(ix) = (buffer1(ix) + buffer2(ix))/2.
else
   call edest('Bad operator specified',0)
go to 2000
endif

do 1200 continue
C------put the buffer
C
C
call pack(dir1(10),buffer3,buffer3)

if (mcaout(buffer3) .lt. 0) then
call edest('Write failed ',0)
go to 2000
endif

iread = iread + 1

go to 1000

elseif ( (iret1 .eq. 0 .and. iret2 .eq. 1) .or.
   (iret1 .eq. 1 .and. iret2 .eq. 0) ) then
   call edest('Pipes 1 and 2 did not empty at the same time',0)
else
   if (mcacdr(h1,cards1) .ne. 0) then
call edest('Read of comment cards from area 1 failed',0)
go to 2000
endif

if (mcacdr(h2,cards2) .ne. 0) then
call edest('Read of comment cards from area 2 failed',0)
G 427:       goto 2000
G 428:       endif
G 429:       C
G 430:       C------now write the comment cards
G 431:       C
G 432:       if (dirl(64) .gt. 0) then
G 433:       endif
G 434:       if (mcacou (cards1) .ne. 0) then
G 435:       call edest('Failed to write comment cards',0)
G 436:       goto 2000
G 437:       endif
G 438:       endif
G 439:       endif
G 440:       endif
G 441:       call sdect('MUGAREA: Done',0)
G 442:       format(1x,i3,1x,2(i12,1x))
G 443:       900
G 444:       2000 return
G 445:       end
Development Environment for McIDAS-X and -OS2

Presented by
Tom Whittaker
McIDAS Development Team Manager

Session 3
McIDAS Developer/Operator Training
October 23-25, 1995
# Table of Contents

Setting up the environment for McIDAS-X ........................ 3-1
   Setting up your user environment .......................... 3-2
   The McIDAS-X library ....................................... 3-3
   Testing your code ........................................... 3-6
   Making HELPs ................................................ 3-7

Setting up the environment for McIDAS-OS2 ...................... 3-8
   Managing the software for locally developed code ......... 3-8
   Required software for local development .................. 3-9
   Considerations for functions and subroutines ............. 3-9
   Compiling and linking ....................................... 3-10
   Building HELP files .......................................... 3-11
   Dynamic Link Libraries (DLLs) .............................. 3-12
Setting up the environment for McIDAS-X

To create a development environment for McIDAS-X, you must first set up a proper user environment. Verify that you or your system administrator has properly installed the McIDAS software in the user account named mcidas. The instructions are in Chapter 1 of the McIDAS-X Users Guide. Once McIDAS is installed, the mcidas account will have two sets of directories:

- package directories
- installation directories

Package directories

Each version of McIDAS-X and other McIDAS packages, such as McIDAS-XCD, builds its own set of directories. The names of the directories depend on the package name and version number. For example, the McIDAS-X 2.1 package directories and their contents are listed below.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>~mcidas/mcidas2.1/src</td>
<td>McIDAS-X 2.1 source files, help files</td>
</tr>
<tr>
<td></td>
<td>and binaries</td>
</tr>
<tr>
<td>~mcidas/mcidas2.1/data</td>
<td>McIDAS-X 2.1 data files</td>
</tr>
</tbody>
</table>

Installation directories

The installation directories and their contents are listed below.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>~mcidas/admin</td>
<td>administrative files</td>
</tr>
<tr>
<td>~mcidas/bin</td>
<td>program executables</td>
</tr>
<tr>
<td>~mcidas/data</td>
<td>data files</td>
</tr>
<tr>
<td>~mcidas/help</td>
<td>help files</td>
</tr>
<tr>
<td>~mcidas/inc</td>
<td>include files</td>
</tr>
<tr>
<td>~mcidas/lib</td>
<td>libraries</td>
</tr>
</tbody>
</table>
Setting up your user environment

Your user account

Verify that your account is set up as a normal McIDAS user, following the steps below. See Chapter 2 of the McIDAS-X Users Guide for additional information.

1. Login to your account.

2. Modify the environment variable PATH in your .profile (ksh) or .cshrc (csh) files. Insert $HOME/mcidas/bin, the directory with your site’s locally developed code (for example, ~mclocal/mcidas/bin), and ~mcidas/bin in your PATH, in that order.

3. You may also need to modify your PATH if it does not contain all of the required directories or have them in the correct order, as listed below.

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX 3.2.5 and 3.2.5p</td>
<td>none</td>
</tr>
<tr>
<td>HP-UX 9.0.3 and 9.0.5</td>
<td>add /usr/bin/X11</td>
</tr>
<tr>
<td>IRIX 5.2 and 5.3</td>
<td>none</td>
</tr>
<tr>
<td>SunOS 4.1.3</td>
<td>add /usr/local/bin, /usr/openwin/bin, /opt/SUNWspiro/bin, /usr/ccs/bin, and /usr/openwin/bin; if your PATH contains /usr/ucb, it must follow /opt/SUNWspiro/bin, /usr/ccs/bin, and /usr/bin</td>
</tr>
<tr>
<td>Solaris 2.3 and 2.4</td>
<td>add /opt/SUNWspiro/bin, /usr/ccs/bin, and /usr/bin</td>
</tr>
</tbody>
</table>

4. Logout and login again for the changes to take effect.

Your development environment

Before doing local development, you must create the following directories.

- $HOME/mcidas
- $HOME/mcidas/bin
- $HOME/mcidas/data
- $HOME/mcidas/src
- $HOME/mcidas/help
- $HOME/mcidas/lib
Every McIDAS-X upgrade provides a new library of functions, which is placed in the file ~/.mcidas/lib/libmcidas.a. As a developer, you can link to this library if you are only developing commands. If you are developing functions and subroutines, you can either copy this library to your $HOME/mcidas/lib directory, or create and use your own development (usually temporary) library for your function and subroutine object code.

There are two ways to compile your code: using the fx script, which invokes the mccomp and mcbar scripts to do the compiles and linking, or using the latter routines directly. mccomp and mcbar are provided to keep system-dependent compiler and library options transparent to the developer.

If you develop subroutines or functions, as opposed to just commands, it is better to use mccomp and mcbar directly because you can put your object code into a separate library and link from there. The fx script, on the other hand, puts all object code into the McIDAS library, which forces you to copy libmcidas.a into your $HOME/mcidas/lib each time you start a different project.

If you use the fx script for compiling, follow these steps:

1. Make a fresh link to the main.o file in the mcidas account.

   cd $HOME/mcidas/src
   rm main.o
   ln -s -mcidas/lib/main.o

2. Make a fresh link to the libmcidas.a file in the mcidas account. If you have library routines to put in libmcidas.a, use cp instead of ln -s below.

   cd $HOME/mcidas/lib
   rm libmcidas.a
   ln -s -mcidas/lib/libmcidas.a

3. Make fresh links from $HOME/mcidas/src to the include files in ~/.mcidas/inc.

4. The fx script uses the scripts mccomp and mcbar to do the compiles and library updates. They are located in ~/.mcidas/bin. If you need to modify the compiler options, make a private version of mccomp in your $HOME/mcidas/bin directory. Before modifying the archive options, make a private version of mcbar in your $HOME/mcidas/bin directory.
5. On IRIX, there is a system-supplied command called fx in /usr/bin. To use the McIDAS-X fx command, you must have the mcidas bin directories (such as ~/mcidas/bin and $HOME/mcidas/bin) before /bin and /usr/bin in your PATH.

6. Recompile your locally developed software using the new fx script and mcidas library. You may encounter problems when compiling. For example, using Fortran direct I/O with RECL= is not supported since it is not portable. On some platforms, RECL=7 means the record length is 7 bytes; on other platforms, it means 7 words. Use the McIDAS LW I/O or LB I/O instead.

We recommend using a makefile for compiling. For example:

```
# Sample makefile for local McIDAS development...September, 1995
#
# First section are things you'll set up for every project...
#
# The list of .pgm's to compile; comment out if not used.
MCFPSRC = run.pgm \ 
vranot.pgm

# The list of .for's to compile; comment out if not used.
MCFPSRC = runaix.for

# The list of .c's to compile; comment out if not used.
MCCFSRC =

# The list of .dlm's for compile; comment out if not used.
# MCDLMSRC =

# Derive the object file names; comment out any not used...
MCFPIM = $(MCFPSRC:.pgm=.mx)
MCFPOBJ = $(MCFPSRC:.for=.o)
# MCCFPOBJ = $(MCCFSRC:.c=.o)
# MCDLMOBJ = $(MCDLMSRC:.dlm=.o)

# The name of the development library; if not using development
# library (that is, there is no MCFPSRC or MCCFSRC), be sure
# to set MYMCLLIB =
MYMCLIB = dev
MYMCLIBA = lib$(MYMCLIB).a
MYMCLIBL = -1$(MYMCLIB)

# MYMCLLIB =
#
# Now define the McIDAS environment and the developer's home
# directory.
#
# McIDAS root directory and associated subdirectories
MROOTDIR = /home/mcidas
MCINC = $(MROOTDIR)/inc
MCLIB = $(MROOTDIR)/lib
MCBINDIR = $(MROOTDIR)/bin

# the suffix of the name of the McIDAS library
```
MCLIB = mcidas
# developer's root directory and associated subdirectories
MYROOTDIR = $(HOME)
MYMCLIBDIR = $(MYROOTDIR)/mcidas/bin
# the location of the core version of main.o
MAIN_o = $(MCLIBDIR)/main.o

###
# Set the DEBUG flags for the compiler.
#-------------------
DEBUG =

#-------------------
# L_OBJ can force certain object files to be used during the
# link instead of getting it from a library; This macro can
# also be modified on the command line of the make call
#-------------------
L_OBJ =

#-------------------
# Now define the location of includes, libraries, and compile
# scripts.
#-------------------

# the list of include file arguments
INCARGS = -I. -I$(MCINCDIR)
# the list of library file arguments
LIBARGS = -L. -L$(MCLIBDIR) $(MYMCLIBL) -L$(MCLIB) -lx11
# the compile, link and library archive commands for McIDAS
COMPCMD = $(MCLIBDIR)/mccomp
LINKCMD = $(MCLIBDIR)/mcrap
LIBARC = $(MCLIBDIR)/mcbar
CONVDLM = $(MCLIBDIR)/convdlm

#-------------------
# Create the suffix rules for compiles, etc.
# ( the "$*" refers to the root name of the prerequisite file )
#-------------------

.SUFFIXES: .o .mx .for .c .pgm
# compile the .c's
.c.o:
$COMPCMD $(INCARGS) $(DEBUG) -c $*.c
$LIBARC $(MYMCLIBA) $*.o
# compile the .for's
.for.o:
$COMPCMD $(INCARGS) $(DEBUG) -c $*.for
$LIBARC $(MYMCLIBA) $*.o
# compile the .dlm's
.dlm.o:
$CONVDLM $*.dlm
$COMPCMD $(INCARGS) $(DEBUG) -c $*.f
$COMPCMD $(INCARGS) $(DEBUG) -c $*.f
Testing your code

Keep your testing environments separate from all user environments. The primary points to keep in mind are the following:

- get a good data sample; for example, the dataset provided with the *McIDAS-X Learning Guide*

- verify that the account is set up like a user account

- set your MCPATH environment variable appropriately

- use different names for the source and executables if you change core McIDAS code

- be aware that we will never issue a core McIDAS command beginning with the letter y

If your changes will become part of your routine operations, then you should be prepared to recompile, link, and retest with each upgrade that you install. Using makefiles will help this effort.
Making HELPs

Create McIDAS-style HELPs (lines prefixed by `C ?` in the source code) for all locally developed commands using the template below. Note that macro commands, which have the `.mac` extension, use the double-quote ("”) instead of `C` as the comment character.

```
C ? NAME -- Describe the purpose of this command
C ? NAME FUNCT1 parm1 parm2 <keywords> "quote
C ? Parameters:
C ? FUNCT1 describe the purpose of this function switch/option
C ? FUNCT2 describe the purpose of this function switch/option
C ? parm1 describe this parameter (def=default value)
C ? parm2 describe this parameter (def=default value)
C ? "quote describe the contents of the quote string
C ? Keywords:
C ? KEYNAME= describe values (def=default)
C ? KEY2=YES describe effect (def=default)
C ? Remarks:
C ? Add remarks, from most to least important. Use complete
C ? sentences. If there are multiple remarks, separate them
C ? with a single blank line, as below.
C ? Always end the help section with a line of 10 dashes, as below.
```

To produce a McIDAS help file from your source code, do the following:

1. Change to the help directory.
   
   Type: `cd $HOME/mcidas/help`

2. Make help files from your code in your `$HOME/mcidas/src` directory by entering the two command lines below.
   
   Type: `mcmkhelp $HOME/mcidas/src/*.pgm`
   `mcmkhelp $HOME/mcidas/src/*.mac`

3. Have all users who run locally developed commands add the appropriate directories to the MCPATH setting. For example, users who run commands in the `mclocal` account must add `~mclocal/mcidas/help` to their MCPATH. Users who run commands in their own account, must add `$HOME/mcidas/help` to their MCPATH.
Setting up the environment for McIDAS-OS2

The McIDAS-OS2 Users Guide discusses the user environment, while the old McIDAS Applications Programming Manual provides details of the developer environment. In both cases, the directory tree and all developer settings are established during the installation, except as noted below.

The directory tree established after installing both the McIDAS-OS2 and the Development software is shown below.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>\mcidas\tools</td>
<td>libraries, scripts, editor</td>
</tr>
<tr>
<td>\mcidas\source</td>
<td>core source code: do not change</td>
</tr>
<tr>
<td>\mcidas\working</td>
<td>local source code</td>
</tr>
<tr>
<td>\mcidas\code</td>
<td>core executable code: do not change</td>
</tr>
<tr>
<td>\mcidas\user\code</td>
<td>local executable code</td>
</tr>
<tr>
<td>\mcidas\data</td>
<td>data of all kinds</td>
</tr>
<tr>
<td>\mcidas\help</td>
<td>.HLP files</td>
</tr>
</tbody>
</table>

Managing the software for locally developed code

Keep the source code for all locally developed code in the \mcidas\working subdirectory. The McIDAS upgrade procedures do not affect the contents of this directory. Also, begin the names of your .PGM files with the letter y.

Do not change the LIBMC.LIB as delivered with each upgrade; by default, the fcmd uses MCUSER.LIB for local subroutines and functions. Delete this library between projects.

Finally, use make or create compile scripts to deal with projects having several things to compile and/or link. You will generally need to recompile and link your local code after installing each upgrade. For example:

```
REM Rebuild the "YPROG" command
SETLOCAL
CD \MCIDAS\TOOLS
DEL MCUSER.LIB
CALL F MYSUB1 CLI
CALL F MYSUB2 LI
CALL F YPROG L
```
Required software for local development

McIDAS-OS2 is built using the emx/gcc compilers and the f2c Fortran translator. You may obtain a copy of these from SSEC either on diskette or via anonymous ftp, following the instructions in the upgrade information. Once you obtain the .ZIP file and unpack its contents, a directory tree under \EMX\ is created. You should read the information in \EMX\DOC\ and \EMX\BOOK\, Use the view command in OS/2 to read the .INF files.

The \EMX directory contains a script called setemx.cmd. If you run this script, the environment is set up such that you can compile your code within that session only. If you do a large amount of local development, take the set commands from this script and, after substituting the correct drive value, put them in the \CONFIG.SYS file; Do NOT put any other statements from the setemx.cmd file into \CONFIG.SYS.

Considerations for functions and subroutines

The IBM linker, LINK386, restricts the searching of libraries to a prescribed order; once a library is completely searched, the next one in sequence is searched. In the f.cmd compile script, MCUSER.LIB is always searched before LIBMC.LIB.

If you create a subprogram that is called only by a routine in the LIBMC.LIB (this would only happen if you intended to replace something in LIBMC.LIB), your routine will not be linked because the object code is put into the MCUSER.LIB by f.cmd. In this case, copy the LIBMC.LIB into MCUSER.LIB before compiling. In the example recompile script above, change the line:

    DEL MCUSER.LIB

to:

    COPY LIBMC.LIB MCUSER.LIB
Compiling and linking

The `fcmd` compile script is provided with McIDAS-OS2 in the `\MCIDAS\TOOLS` directory when you install the Development software. This script, written in REXX, provides all the commands you will need to compile and link your code. The general form of the command is:

\[
\text{F <name> <option>}
\]

where: `<name>` is the name of the source file without an extension

`<option>` describes the type of compile to do and the type of source code

Entering F without `<name>` `<option>` produces this help information:

```
-------- McIDAS compile script --------
F <name> <function: L,L1,NM,MAC,NO,C,CLI>
F <name> DL <defname> <dllname>
```

Specify `<name>` with NO extensions!

Extensions supplied are:

- `L` --> `.PGM` (Fortran McIDAS commands)
- `LI` --> `.FOR` (Compile and LIB Fortran subprograms)
- `DL` --> `.DLM` (Dynamic load modules)
- `NM` --> `.FP` (Non-McIDAS programs; put .EXE into `\MCIDAS\TOOLS`)
- `NMC` --> `.FP` (Non-McIDAS programs; put .EXE into `\MCIDAS\CODE`)
- `MAC` --> `.MAC` (McIDAS macros)
- `NO` --> `.FOR` (Compile Fortran code only - no LIB step)
- `C` --> `.C` (Compile C code only - no LIB step)
- `CLI` --> `.C` (Compile C code and insert into library)
- `CP` --> `.C` (C language non-McIDAS commands)
- `CL` --> `.C` (C language McIDAS commands)

This will search for `<name>` first in the current directory, then in `\mcidas\working`, and finally in `\mcidas\source`.

The environment variables are shown below:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>useworking=NO</td>
<td>suppresses the normal search of <code>\mcidas\working</code></td>
</tr>
<tr>
<td>mcout=c:\directory</td>
<td>writes .exe files into <code>\directory</code></td>
</tr>
<tr>
<td>mapout=c:\directory</td>
<td>writes MAP files into <code>\directory</code></td>
</tr>
</tbody>
</table>
Building HELP files

The McIDAS-OS2 HELP command uses the files placed in the \mcidas\help\ directory. The files are the names of McIDAS-OS2 commands followed by a .HLP extension. For example, IMGDISP.HLP is the HELP information for the IMGDISP command. If you follow the standards previously described for McIDAS-X for formatting the help section of your locally developed commands, you can use the mkhelp program in \mcidas\tools\ to extract this information and place it in the proper directory.

The command syntax for mkhelp is shown below. You must run it from the OS/2 command prompt, not within McIDAS-OS2.

```
mkhelp <filename> <options> <output_directory>
```

where <filename> is the name of the source code file for the McIDAS-OS2 command you want to extract the HELP information from. You may also specify a wildcard with an extension (eg., *.PGM) to do all the files with this extension.

<options> may be LIST if you just want the extracted HELP information listed to the screen the name of a 'template' file containing lines of text that are to be included in the HELP (we recommend that you include a message like "This code was developed at XYZ") X if you want neither of the above options <output_directory> to specify the directory where the .HLP file(s) will be written. The default is \mcidas\help\.

If you run this command with no arguments, the following one-line help message is displayed:

```
MKHELP <filename> <templatefile | LIST | X> <output directory>
```
Dynamic Link Libraries (DLLs)

Navigation and calibration routines are handled in McIDAS-OS2 as Dynamic Link Library modules (DLLs). They are not actually linked into your program when it's compiled, but rather remain in a special file that can be loaded into memory and then linked into your program after it begins to run. OS/2 makes extensive use of this feature; McIDAS-OS2 uses it not only for navigation and calibration, but also for all graphics displays and part of the MDX command group.

You must make three DLLs for each navigation or calibration module and one for each graphics driver (platform). You must explicitly perform each of these, as the fcmd does not. For example, to compile a navigation module for type ABCD, you would run these commands:

```
F NVXABCD DL NAVLIB NV1ABCD
F NVXABCD DL NAVLIB NV2ABCD
F NVXABCD DL NAVLIB NV3ABCD
```

The NAVLIB parameter defines the definitions file to use for creating navigation library DLL routines. The final parameter is the name of the created .DLL file. You must replace the X in the root name (for example, nvXgvar) with the digits 1-3 (for example, nv1gvar, nv2gvar, nv3gvar) as illustrated.

A similar process is used for calibration:

```
F KBXABCD DL CALLIB KB1ABCD
F KBXABCD DL CALLIB KB2ABCD
F KBXABCD DL CALLIB KB3ABCD
```

CALLIB designates the definitions file for calibration.

Graphics are compiled using just one F.CMD per display type:

```
F GRADVCn DL GRALIB GRADVCn
```

where n is the number assigned to the particular display type.
Writing GUIs for McIDAS using Tcl/Tk

Presented by

Susan Gorski - McIDAS Applications Programmer
David Glowacki - McIDAS Systems Programmer

Session 4
McIDAS Developer/Operator Training
October 23-25, 1995
Table of Contents

Overview .................................................. 4-1

Terminology ............................................. 4-2

Tcl syntax and structure ............................... 4-3

Variables ............................................... 4-5

Tk syntax ................................................ 4-7

Terms .................................................... 4-7

Examples ............................................... 4-8

Error reporting ........................................ 4-8

Packing in Tk ........................................... 4-9

Using external procs ................................... 4-10

Widgets .................................................. 4-11

Useful procs ............................................. 4-16

Find a User Common value ........................... 4-16

Get today’s date ...................................... 4-16

Get the current time .................................. 4-17

List the U.S. states ................................... 4-18

List all available areas. ............................... 4-19

Set standard options .................................. 4-21

Run a McIDAS command; send output to listbox .... 4-21

Run a McIDAS command; send output to text window 4-23

Considerations for building a GUI .................. 4-26

Sample GUI code ....................................... 4-27

Solution set ............................................ 4-38

Resources .............................................. 4-44
Overview

McIDAS user interfaces in the past have included: a command window, a soft tablet, a hard tablet and the F Key menu system. When we began developing a Graphical User Interface for McIDAS, we assessed different underlying packages and chose Tcl/Tk.

Tcl (Tool command language) and the Tk (Tool kit) are two software packages developed by John Ousterhout at the UC-Berkeley. Together they make up a system for developing Graphical User Interface (GUI) applications. Tcl is an interpretive scripting language with variables and control structures; it contains the control portion of a GUI. Tk is a tool kit of graphical widgets that can be accessed from a Tcl script. Tk contains the visual portion of a GUI.

The ease of programming in Tcl/Tk allows for rapid McIDAS GUI development. With very little effort, a GUI can be prototyped for a look-and-feel evaluation before any functionality is added. Because Tcl/Tk source code is freely available, you can obtain the development package at no cost, making it more accessible. Tcl/Tk code makes it easy for you to create your own GUI, and alter the GUIs we create.

The McIDAS GUI is a series of intuitive menus and command GUIs for running McIDAS commands. It displays defaults, offers easy selection of options and limits commands to decrease complexity. One such GUI is Merlin, a freely distributed package that uses McIDAS and Tcl/Tk as its base. Because Merlin has no command line, all requests for data display are through the GUI. To use MERLIN as a base for other projects with more specific applications, we also developed sidecars.

This training session provides the information you need to write your own GUI using Tcl/Tk.
Terminology

The terms below are used throughout this section.

**GUI**
- Graphical User Interface

**pack**
- Tk command for arranging slave widgets inside a master widget

**proc**
- submodule of Tcl code that allows the code to be reused in different scripts

**progressive disclosure**
- giving users only as many choices as they need to make a simple command, but allowing expanded functionality as needed

**sidecar**
- a set of GUI applications developed for a project that can be run alongside MERLIN

**Tcl**
- Tool command language; the scripting language that contains the control portion of a GUI developed in Tcl/Tk

**tclsh**
- Tcl shell application

**Tk**
- Tool kit; a package of graphical widgets that contain the visual portion of a Tcl/Tk GUI

**unpack**
- the `pack forget` command in Tk; when a widget is unpacked, it is not displayed in the master widget

**widget**
- basic building block for a GUI in Tk; a window with a particular appearance and behavior

**wish**
- a windowing shell application that includes all of tclsh and the commands defined by Tk
Tcl syntax and structure

Tcl is the control portion of the Tcl/Tk package. Below is a brief description of its syntax and structure.

**Variable - can hold numeric, string or list value**

```
set a 1
set a "string"
set a "a [2 3]"
set b $a
```

**Array - variable with a string index**

```
set a(0) "spoon"
set a(fork) "knife"
set b(1) $a(fork)
```

**Control structures**

```
while <boolean> <body>
if <boolean> [then] <body1> [ else <body2> ]
foreach <variable> <list> <body>
for <loop_init> <boolean> <loop_incr> <body>
switch <flags> <value> <pattern1> <body1> [ <pattern2> <body2> ...]
break
continue
return <string>
```

**Variable creation/deletion**

```
set <name> <value>
unset <name> [ <name> ... ]
```

**Variable manipulation**

```
append <name> <value> [ <value> ... ]
incr <name> [ <increment> ]
```

**Numeric expression-related commands**

```
expr <expression>
```
String-related commands
format, regexp, regsub, scan, string, subst

Array-related commands
array

List-related commands
concat, join, lappend, lindex, linsert, list,
llength, lrange, lreplace, lsearch, lsort, split

File-related commands
cd, close, eof, file, flush, gets, glob, open,
puts, pwd, read, seek, tell

Tcl function-related commands
catch, error, global, proc, rename, return, uplevel, upvar

Process-related commands
eval, exec, exit, pid, source

Miscellaneous commands
auto_nkindex, history, info, time, trace, unknown
Variables

In Tcl/Tk, words are grouped together using double quotes or curly brackets, depending on when the variables inside the groups are substituted. Variables inside double quotes are substituted; variables inside curly brackets are not substituted.

For example:

```
set a "test"
puts "this is a $a"
```

will print out *this is a test*.

However,

```
set a "test"
puts {this is a $a}
```

will print out *this is a $a*.

You can use this to your advantage when running Tcl code triggered by an event. Because Tcl/Tk code is interpretive, the script is run first, all variables are substituted (except those in brackets), and all widgets are created. When the GUI is displayed, all code at the lowest level is already interpreted, and wish now sits and waits for a user event. When a user performs an action in a window, an event is generated. Some events are captured and cause some branch of Tcl code to be interpreted at that time. For example, when a user presses the left mouse button over a button widget, an event causes the widget’s `-command` option to run.

The example below sets the color variable to red. When the button `.setit` is pressed, the variable is set to blue. The button `.colorbutton` prints the value of the color variable. Since the value of `$color` is substituted when this script is run (because the command is in quotes, not brackets), the output is always red, even if the `.setit` button is pressed before the `.colorbutton` button.

```
set color red
button .setit -command "set color blue"
button .colorbutton -command "puts $color"
```
The example below outputs the current value of the color variable. Putting
the event code in brackets reflects the current state of the variables at the
time the button is pressed, instead of the values at startup. So another
widget can change the value of a variable, and that value is always current.
Pressing the .colorbutton outputs red. Pressing the .setit button, followed
by the .colorbutton outputs blue.

```
set color red
button .setit -command "set color blue"
button .colorbutton -command {set swatch $color}
```

All widgets have a defined set of actions that trigger events. Additionally,
you can use the bind command to add trigger events to a widget definition,
allowing different actions to cause events.
Tk syntax

To create a Tk widget, use the command below:

```
widget_type widget_name -option1 value1 -option2 value2 \n  -option3 value3
```

Terms

Each term in the command is defined below.

- `widget_type` one of the valid Tk widget types, for example: label, button, frame
- `widget_name` name of the widget; all references to this widget use this name. The hierarchy of the widget is contained in its name. For example, .frame.button is a widget named button inside the frame .frame. Precede all widget names with a period. Also use the period to separate fields of widgets in a hierarchy.
- `-option#` option name; options can be general to all widgets, for example: -background -foreground -relief. All widget types have options specific to that widget; you can find these on the man pages. Most options have valid synonyms to shorten the option name; for example, -background shortens to -bg.
- `value#` value assigned to the option; relates to the class of widget. A color option like -foreground has valid values of #XgXbX or a color name.

\ncontinuation character indicating the command is continued on the next line. The command is more readable if the next line is indented from the first.
Examples

The command below makes a frame named `.showme`, which appears raised and has a red background color.

```
frame .showme -relief raised -bg red
```

The command below makes a button named `.showme.now`, implying that `now` is a child of the `.showme` widget. The button's background is green, the foreground is yellow, the words *Press Me* appear on the button, and, when pressed, the button prints *Now what?* via the Tcl `puts` command.

```
button .showme.now -command {puts "Now what?"} -bg green -fg yellow \\
-text "Press Me"
```

Error reporting

Syntax errors are reported as they are encountered. Because Tcl/Tk is interpretive, this may occur immediately after the program is run, or as part of a context that will be interpreted when an event is invoked. If the errors occur when the command starts, you will probably see the errors reported in the originating window. If an error occurs when running the GUI, you will get a pop-up window reporting the error, and giving the option to quit or view the stack.
Packing in Tk

Use the `pack` command to manage the widget layout. Widgets will not appear on the screen until they are managed by the geometry manager via a `pack`. You can arrange the widgets in a frame vertically or horizontally. They can appear in any order, fill the frame, or have space padding them within the frame. If you don't specify the widget's size, it can expand to accommodate the attributes.

Use the steps below to create a frame and then pack it.

1. Make a frame called `a`.
   ```
   frame .a -relief raised -borderwidth 4
   ```

2. Make two buttons called `a.b` and `a.c`, which are children of `a`.
   ```
   button .a.b -text "button1" -command "puts $junk"
   button .a.c -text "button2" -command "puts $test"
   ```

3. Pack the two buttons horizontally.
   ```
   pack .a.b .a.c -side left
   ```

4. Pack the buttons vertically.
   ```
   pack .a.b .a.c -side top
   ```

5. Pack the buttons with a 2 millimeter pad on all sides.
   ```
   pack .a.b .a.c -side left -padx 2m -pady 2m
   ```

6. Fill the available space in the frame.
   ```
   pack .a.b .a.c -side top -fill x -fill y
   ```
Using external procs

If you create a proc that will be used in more than one GUI, add that proc to your proc library by running a script containing the following lines:

```bash
#!/usr/local/bin/wish -f
auto_mkindex . myproc.tcl
exit
```

Or, run `wish` and, at the prompt, enter these two lines:

```bash
auto_mkindex . myproc.tcl
exit
```

This adds a reference to your proc to the file TcIIndex in your current directory. When a GUI is run, it will look in that file for the location of the external proc.
Widgets

The widgets used in the McIDAS GUI are described below.

Frame

The frame widget is the GUI's basic building block. Use it to group other widgets and to build nested layouts. In a frame, you can arrange widgets vertically or horizontally.

Possible use: arrange a label widget to the left of an entry widget.

```
frame .a -width 15m -height 10m -relief raised -borderwidth 1.4m
frame .b -width 15c -height 10c -relief sunken -borderwidth 1.4c
frame .c -width 0.59i -height 0.39i -relief flat -borderwidth 0.04i
frame .d -width 42p -height 29p -relief groove -borderwidth 3p
frame .e -width 0.59i -height 10m -relief ridge -borderwidth 4
pack .a .b .c .d .e -side left -padx 2m -pady 2m
```

Button

The button widget, also called a command button, is a rectangular box to press to perform an action. The label on the button can be text or a bitmap.

Possible use: bring up a window to query the user when the button is pressed.

```
frame .a -borderwidth 4
pack .a -side bottom

button .a.b1 -text QUIT -command {destroy .}
button .a.b2 -text PRINT
   -command {puts "Pushed Print button"}
button .a.b3 -text NOTHING
pack .a.b1 .a.b2 .a.b3 -side right
```
Message

The message widget displays multi-line text strings. Text appears exactly as typed, including spaces and carriage returns. Enter text as one long line so it displays appropriately if the widget is resized.

Possible use: describe the GUI’s purpose in a line at the top of the GUI.

```tcl
set text1 "Some left-justified text in the first messagebox"
set text2 "This text will be centered in the second messagebox"
message .a -width 5c -justify left -text $text1
message .b -aspect 50 -justify center -relief groove -text $text2
pack .a .b -side top
```

Label

The label widget displays a label, which can be text or a bitmap. No action is performed.

Possible use: display a label opposite an entry widget to describe what the user can type into it.

```tcl
label .a -bitmap questhead
label .b -text "I don’t understand"
pack .b .a
```

Checkbutton

The checkbutton widget is a toggle switch for a variable. The variable is set when the button is pressed. Checkbuttons act independently of each other.

Possible use: describe the attributes of a display where the attributes can be off or on.

```tcl
frame .a -borderwidth 4
pack .a -side bottom

checkbutton .a.cl -text CHECK1 -variable chk1 \  
-command {puts "$chk1=$chk1, chk2=$chk2"}
checkbutton .a.c2 -text CHECK2 -variable chk2 \  
-command {puts "$chk1=$chk1, chk2=$chk2"}
pack .a.cl .a.c2 -side left

puts "chk1=$chk1, chk2=$chk2"
```
**Listbox**

The listbox widget gives the user a list of options. You can configure the widget so users choose one or a range of options, or double-click on a value. Each option in the list is separated by commas and appears on a separate line. To view long lists, tie the list to a vertical scrollbar.

Possible use: give user a list of states to choose from.

```tcl
listbox .a
for {set i 0} {$i < 10} {incr i} {
    .a insert end $i
}
pack .a
bind .a <Double-Button-2> {
    puts [selection get]
}
```

**Radiobutton**

The radiobutton toggle, when toggled on, toggles another button off. Radiobuttons are grouped and interconnected. Although they are attached to the same variable, each has its own value. When a button is pressed, the variable is assigned the value attached to the button.

Possible use: query the user on a color attribute, where only one color can be chosen.

```tcl
frame .a -borderwidth 4
pack .a -side bottom

radiobutton .a.r1 -text R1 -variable shared -value r1 \
    -command {puts "shared=$shared"}
radiobutton .a.r2 -text R2 -variable shared -value r2 \
    -command {puts "shared=$shared"}
radiobutton .a.r3 -text R3 -variable shared -value r3 \
    -command {puts "shared=$shared"}
radiobutton .a.r4 -text R4 -variable shared -value r4 \
    -command {puts "shared=$shared"}
pack .a.r1 .a.r2 .a.r3 .a.r4 -side left
puts "shared=$shared"
```
The menubutton widget, when pressed, presents the user with a menu of buttons. This widget supports: radiobuttons, checkbuttons, command buttons, and cascade buttons.

Radiobuttons and checkbuttons are defined in this section. Command buttons run a block of code. When pressed, cascade buttons will cascade down another set of buttons. You can place separators between sets of buttons to logically differentiate subsets. Use a menubutton to implement progressive disclosure.

Possible use: let a user choose map boundaries from a predefined set of boundaries or a list of states.

```tcl
menubutton .a -text Menu -menu .a.b
pack .a

menu .a.b
.a.b add checkbutton -label "Option 1" -variable option1
.a.b add checkbutton -label "Option 2" -variable option2
.a.b add separator
.a.b add radiobutton -label "Exclusive 1" -variable exclusive -value 1
.a.b add radiobutton -label "Exclusive 2" -variable exclusive -value 2
.a.b add separator
.a.b add command -label "Select" -command {puts "Selected"}
.a.b add separator
.a.b add cascade -label "Print..." -menu .a.b.c

menu .a.b.c
.a.b.c add command -label "Print options" \ -command {puts "$option1 $option2"}
.a.b.c add command -label "Print exclusive" \ -command {puts "$exclusive"}
```

**Scale**

The scale widget, also called a slider, is a button that can be moved from end to end to choose a value from a spectrum of values. This widget works well to select from a small, evenly spaced set of numerical values.

Possible use: set the color level on a scale of 0-255.

```tcl
scale .a -label Decimal -orient horizontal -from 0 -to 9 -command dumpScale
pack .a
proc dumpScale value {puts "Value is $value"}
```
Scrollbar

The scrollbar widget is a slider that you can attach to another widget. When the slider on the scrollbar is moved, the contents of the other widget are also scrolled.

Possible use: attach to a small text window, so the user can scroll with the scrollbar to see more information.

```tcl
listbox .a -yscrollcommand ".b set"
for {set i 0} {$i < 10} {incr i} {
    .a insert end $i
    pack .a -side left
}
bind .a <Double-Button-2> {
    puts [selection get]
}
scrollbar .b -command ".a yview"
pack .b -side right -fill y
```

Entry

The entry widget lets the user type and edit a one-line text string. Use the -textvariable option to display a variable in the entry widget. The variable is updated whenever the user edits the entry widget.

Possible use: display a field containing the name of a file to save to disk. The user can edit the file name before running an operation on the file.

```tcl
label .a -text "Type something"
entry .b -textvariable text -relief sunken
bind .b <Return> {
    puts "$text"
    .b delete 0 end
}
pack .a .b -side left
```
Useful procs

The procs below are ones that we found useful when creating GUIs. The numbers on the left are for reference only and are not part of the code.

Find a User Common value

This proc will find a specific value in User Common.

1. #
2. # UCval index
3. # get value from User Common at index
4. # Arguments:
5. #    index - index of value to obtain
6. #    return - value at index in UC
7. #
8. proc UCval {index} {
9.    global uc
10.   open pipe and run UCU PEEK command
11.   set ucupeek [open "|echo UCU PEEK $index | ucu.mx $uc" r]
12.   read output from th command
13.   gets $ucupeek line
14.   close the pipe
15.   catch {close $ucupeek}
16.   scan output line for value (get 4th value on line
17.   set value [lindex $line 4]
18.   return the value.
19.   return $value
20. }

Get today's date

This proc will give you today's Julian date.

1. #
2. # Today
3. # get todays Julian date from McIDAS
4. # Arguments:
5. #    return - todays Julian Date
6. #
7. proc Today {} {
8. }

Writing GUIs for McIDAS using Tcl/Tk

McIDAS Developer/Operator Training
October 24, 1995
12. global uc
13. 14. # start up a McIDAS command and set the output pipe to "date"
15. set date [open "|echo TL Y | tl.mx $uc" r]
16. # get the first line of output from $date and store in "line"
17. gets $date line
18. # close the pipe
19. catch {close $date}
20. # pick the second value off the line and set that to the return value
21. set value [lindex $line 2]
22. # return $value as the return value
23. return $value
24. 25. 26. }

Get the current time

To designate the current time, which is the default for the TIME parameter, use the Now proc.

1. #
2. # Now
3. # Get the current time
4. # Arguments:
5. #
6. #
7. #
8. # return- Current time
9. #
10. proc Now {} {
11.  
12.    global uc
13.  
14. # Start up McIDAS command and pipe data to "date"
15. set date [open "|echo TL H | tl.mx $uc" r]
16. # first line of output goes to line
17. gets $date line
18. # close the pipe
19. catch {close $date}
20. # value gets set to second value in output string
21. set value [lindex $line 2]
22. # get the first two digits out of the string (pull the HH out of HHMMSS)
23. set val [string range $value 0 1]
24. # return the HH value
25. return $val
26. 27. 28. }
To present the user with a list of U.S. states, tie the command portion of a button to the getstate proc. Getstate sets the variable stateid to the state the user selects.

```tcl
proc getstate {} {
    # If a toplevel window known as states already exists, kill it.
    # This would happen if the user pushed the button twice, we only
    # want one widget.
    catch {destroy .states}
    # the toplevel widget is called .states
toplevel .states
    # define the title and icon title
    wm title .states "Image Listing"
    wm iconname .states "Images"
    # define the minimum size allowed.
    wm minsize .states 11
    # define User Common pointer as global from calling routine - this allows
    # us to access McIDAS commands from outside of the command window
    global uc
    # make a message widget describing this widget.
    message .states.msg -font -Adobe-times-medium-r-normal--*-180*
    -aspect 300 -relief raised
    -text "Select Image by double clicking left mouse button.
    Press Dismiss to exit."
    # Make a frame which will hold the listbox information.
    frame .states.frame -borderwidth 10
    # Make a scrollbar so that we can scroll down in the states.
    scrollbar .states.frame.scroll -relief sunken
    -command ".states.frame.list yview"
    # Make a listbox which will attach the scrollbar to this listbox.
    # The listbox will be 25 characters wide and will display 10 states
    # at a time.
    listbox .states.frame.list -yscroll ".states.frame.scroll set"
    -scroll 1
    -geometry 25x10 -setgrid 1
    # pack the scrollbar and the list into the frame
    pack append .states.frame .states.frame.scroll {right fill} 
    .states.frame.list {left expand fill}
    # insert all of the states into the listbox. Each quoted field will be
    # on a separate line.
    .states.frame.list insert 0 "Alabama (AL)" "Alaska (AK)" "Arizona (AZ)"
    "Arkansas (AR)" "California (CA)" "Colorado (CO)" "Connecticut (CT)"
    "Delaware (DE)" "Florida (FL)" "Georgia (GA)" "Hawaii (HI)" "Idaho (ID)"
    "Illinois (IL)" "Indiana (IN)" "Iowa (IA)" "Kansas (KS)" "Kentucky (KY)"
    "Louisiana (LA)" "Maine (ME)" "Maryland (MD)" "Massachusetts (MA)"
    "Michigan (MI)" "Minnesota (MN)" "Mississippi (MS)" "Missouri (MO)"
    "Montana (MT)" "Nebraska (NE)" "Nevada (NV)" "New Hampshire (NH)"
    "New Jersey (NJ)" "New Mexico (NM)" "New York (NY)" "North Carolina (NC)"
    "North Dakota (ND)" "Ohio (OH)" "Oklahoma (OK)" "Oregon (OR)"
    "Pennsylvania (PA)" "Rhode Island (RI)" "South Carolina (SC)"
```
"South Dakota (SD)" "Tennessee (TN)" "Texas (TX)" "Utah (UT)" \
"Vermont (VT)" "Virginia (VA)" "Washington (WA)" "West Virginia (WV)" \
"Wisconsin (WI)" "Wyoming (WY)" 
# bind the double key press to an action
bind .states.frame.list <Double-1> \ 
{set line [selection get] 
 set ind [string first '{' $line] 
 set state [string range $line [expr $ind +1] [expr $ind +2]] 
 set stateid [set std $state; liststa] 
 destroy .states 
}

# Make a button for the users to exit without making a selection
button .states.ok -text Dismiss -command "destroy .states"

# pack up the message, frame and button
pack .states.msg -side top -fill x
pack .states.frame -side top
pack .states.ok -side bottom

List all available areas

To prompt the user with a list of available areas, call listarea from a
custom in a button. This proc lists the areas and sets the variable area to
the area the user selects.

proc ListArea {area} {
  # destroys any widgets called .listarea that are displayed - in case
  # the user hit the button twice.
  catch {destroy .listarea}

  # we are going to change the bitmap to the hourglass because this can
  # take a while, and then the user knows that we are working on it.
  # get the label value, set it to the hourglass, and force to the screen
  global blabel
  $blabel configure -bitmap @-mcidas/gui0.1/hourglass.xbm
  update

  # define the toplevel widget as .listarea
toplevel .listarea

  # define the window title
  wm title .listarea "Image Listing"

  # define the icon title
  wm iconname .listarea "Images"

  # do not allow resizing past a set size
  wm minsize .listarea 11

  # allow access to user common
  global uc

  # set the message to some instructions for the user
  message .listarea.msg -font -Adobe-times-medium-r-normal---*-180* -aspect 300 -relief raised
  -text "Select Image by double clicking left mouse button.

  Press Dismiss to exit. "
}
# define a frame to hold the output
frame .listarea.frame -borderwidth 10

# make a scrollbar which will attach to the listbox
scrollbar .listarea.frame.scroll -relief sunken
   -command ".listarea.frame.list yview"

# make a listbox to hold the output. The listbox will be 80 characters
# wide and 10 lines long.
listbox .listarea.frame.list -yscroll ".
   .listarea.frame.scroll set"
   -relief sunken
   -geometry 80x10 -setgrid 1

# pack the scrollbar and listbox into the frame
pack append .listarea.frame .listarea.frame.scroll {right fill}
   .listarea.frame.list {left expand fill}

# initialize localarea
set localarea ""

# run the McIDAS command routing output to lalist pipe
set lalist [open "|echo LISARA 1 9999 | lisara.mx $uc" r]

# define a label for above the listbox as a title bar
label .listarea.labell -width 79

# read the first line of output (always a title line),
# and set this to label
gets $alab1 lab1
   .listarea.labell configure -text $lab1

# get the second line which is all underline characters, and toss it
gets $alab2 lab2

# get all the lines and display them in the listbox
while {[gets $alist line] > -1} {
   .listarea.frame.list insert end $line
}

# set the label back to the SSEC logo and flush to the screen
$blab1 configure -bitmap @-mcidas/gui0.1/sseclogo.xbm
update

# close the pipe
close $alalist

# add the action that an item is selected when they double click on
# the line, and parse the line for the area number
bind .listarea.frame.list <Double-1> \ 
   [set areal [selection get]
    set localarea [string range $areal 2 5]
    set area $localarea
    destroy .listarea
   ]

button .listarea.ok -text Dismiss -command "destroy .listarea"
pack .listarea.msg -side top -fill x
pack .listarea.labell -side top -anchor w
pack .listarea.frame -side top
pack .listarea.ok -side bottom
Set standard options

The setoptions proc sets the default colors for the mandatory entry widgets, the font used in the message section, and any other standard options to the values specified in the gui.options file.

```tcl
proc setoptions { } {
  # name two special SSEC options
  #    mand is the color level of mandatory entry widgets
  # messfont is the font of the message widget
  global mand
  global messfont
  # All of the other options from the gui.options file are
  # set in this loop, these will be the defaults for the entire
  # gui, unless a value is set to override them.
  set options [open "gui.options" r]
  while {[gets $options optline] >1} {
    set name [lindex $optline 0]
    set value [lindex $optline 1]
    if {($name == "*mandatoryfield")} {
      set mand $value
    } else {if {($name == "*messageFont")} {
      set messfont $value
    } else {
      option add $name $value
    }}
  }
}
```

Run a McIDAS command; send the output to a listbox

```tcl
proc RouteComm {command headsize} {
  # RouteComm command num_lines_header
  # Run a McIDAS command, sending output to a listbox
  # Arguments:
  # command - command string to enter
  #
  # If there is a .command widget, destroy it.
  catch {destroy .command}
  # access the SSEC Logo and change it to an hourglass, forcing
  display change to the screen with and *update*.
  global blabel
  $blabel configure -bitmap @-mcidas/gui0.1/hourglass.xbm
  update
  # Define the toplevel widget as .command.
  toplevel .command
  # Find the first blank in the string
  set blank [string first " "$command]
```
Parse out the command name
set name [string range $command 0 [expr $blank - 1 ]]

we need a pointer to User common to run McIDAS command
global uc

we need the terminal number in order to write to the text window
global term

Make the window title the same as the McIDAS command
this isn’t great because you get windows titled ‘SL’ which
isn’t very informative, but we are keeping the proc general.
wm title .command [format "%s Output" $name]
wm minsize .command 1 1

make a frame for the listbox and scrollbar
name .command.frame

make a scrollbar for scrolling through the output
scrollbar .command.frame.scroll -relief sunken 
-command "...command.frame.listbox yview"

make a listbox to receive output
listbox .command.frame.listbox 
-yscroll "...command.frame.scroll set" 
-relief sunken -geometry 80x15

pack the scroll and the list into the frame
pack append .command.frame 
...command.frame.scroll {right filly} 
...command.frame.listbox {left expand fill}

convert the command name to lowercase
set lowname [string tolower $name]

add a ".mx" to make the executable name
set executable [format "%s.mx" $lowname]

start up the McIDAS command, routing output to the executable pipe
set execout [open "|echo $command | $executable $uc" r]

run through the header lines, and put them into a label
for {set i 0} { $i <$headsize} {incr i} {
gets $execout lab($i)
set length [string length $lab($i)]
label .command.label$i -text $lab($i) -width $length
pack .command.label$i -side top -anchor w
}

read through the rest of the pipe and write each line into
the listbox
while {[gets $execout line] > -1} {
...command.frame.listbox insert end $line
}

return the label bitmap to SSEC logo
CustomLabel configure -bitmap @-mcidas/gu10.1/sseclogo.xbm

force the screen to display the change
update

close the pipe
catch {close $execout}

button .command.dismiss -text Dismiss -command "destroy .command"
pack .command.frame -side top -fill x
pack .command.dismiss -side bottom
Run a McIDAS command; send the output to the text window

1. # RunCommand command
2. # Run a McIDAS command, sending output to the text window.
3. 
4. # Arguments:
5. # command - command string to enter
6. 
7. proc RunCommand {command} {
8. 
9.   # We need uc to fire off the McIDAS command which needs to point
10.  # to user common since we aren't running from a command line.
11.  
12.   global uc
13. 
14.   # We need the terminal number so that we can route output to the text
15.   # window
16.   
17.   global term
18. 
19.   # RCMP is the SSEC logo at the bottom of the GUI. We need it
20.   # so that we can change it to an hourglass when we run a command.
21.   
22.   global RCMP
23. 
24.   # Set up the pipe to the McIDAS text window
25.   set MCIPIPE [format "McText\%s\%0" $term]
26.   set pipeid [open $MCIPIPE w]
27. 
28.   # change the bitmap to an hourglass.
29.   $ RCMP configure -bitmap @-mcidas/gui0.1/hourglass.xbm
30. 
31.   # force the changes to the screen.
32. 
33.   update
34.   
35.   # echo the command to the text window
36.   puts $pipeid $command flush $pipeid
37. 
38.   # Find the first blank in the command line.
39.   set blank [string first " " $command]
40. 
41.   # set name to the first character up to the first blank
42.   set name [string range $command 0 [expr $blank - 1]]
43. 
44.   # Change the name to lowercase
45.   set lowname [string tolower $name]
46. 
47.   # make the executable name (with a .mx on the command)
48.   set executable [format "%s.mx" $lowname]
49. 
50.   # fork off an exec of the command
51.   catch {exec $executable $uc << $command > $MCIPIPE} 
52. 
53.   # close the pipe.
54.   catch {close $pipeid}
55. 
56.   # Return the label to the SSEC logo
57.   $ RCMP configure -bitmap @-mcidas/gui0.1/sseclogo.xbm
58. 
59.   # flush display changes to the screen.
60.   update
61. }
Skeleton of a McIDAS GUI

To customize, add widgets inside of .frame and alter the bold text to reflect your application.

62. #!/usr/local/bin/wish -f
63. # THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED.
64. # set path to access library procs
65. set auto_path "-mcidas/gui0.2 $auto_path"
66. # set default font and colors from gui.options file
67. setoptions
68. # set up to access McIDAS session
69. set term [lindex Sargv 0]
70. set file [format "UCTERM.0$s" $term]
71. if {$term > 9} { set file [format "UCTERM.0$s" $term] }
72. set odline [open "$od -d $file +6" r]
73. gets Odline line
74. set uc [lindex $line 1]
75. close $odline
76. #
77. # Skeleton for GUI for McIDAS commands
78. #
79. #
80. #
81. # set the window title
82. wm title "NAME OF APPLICATION HERE"
83. # Make a frame to contain the help button in the upper right corner
84. frame .top
85. button .top.help -text "Help" -command mkhelp
86. pack .top.help -side right
87. # set the message area at the top of the gui. Give a brief explanation
88. # of the command. The font will be the default font from gui.options. The
89. # width of the ZZ
90. message .msg -width 500 -relief raised
91. -font $messfont -borderwidth 1 -text " *** Brief explanation of the
92. command***. To execute press OK button. To exit without performing an
93. action, press Dismiss Button."
94. # Make a frame which will contain the body of the GUI, place all of
95. # the widgets here.
96. # frame .frame -borderwidth 10
97. # make a frame which will hold the OK and cancel buttons and the SSEC logo
98. frame .bottom -relief raised
99. # a button to execute the command. All variables will be evaluated as the
100. # value they are when the button is pressed.
101. button .bottom.ok -text OK -width 20 \
102. -command {
103. # set command to the uppercase McIDAS command
104. # which you will be executing
105. set command " ****COMMANDDDD"
106. # runcommand executes the command and routes
107. # the output to the text window
108. RunCommand $command
109. }
# a button to cancel the command without executing anything
button .bottom.cancel -text Dismiss -command {destroy .} -width 20

# a ssec logo between the OK and the cancel buttons
label .bottom.label -bitmap @-mcidas/gui0.2/sseclogo.xbm

# set a global variable 'blabel' to the label widget, so that we can
# change the bitmap of the label to an hourglass when the command is
# running.
set blabel .bottom.label

# Pack the OK, & cancel buttons and label into the bottom frame, padding
# 10 pixels around the button (x and y direction), order will be in order
pack .bottom.ok -side left -padx 10 -pady 10
pack .bottom.label -side left -padx 10 -pady 10
pack .bottom.cancel -side left -padx 10 -pady 10

# Pack the message, the command and the OK buttons into the base widget

# Proc for help.
proc mkHelp {} {

# name the toplevel widget
toplevel .help

# set the title (change 'command name' to your gui name)
wm title .help "Help - command name"

# set the minimum size for the help window
wm minsize .help 10 10

# make a text widget, to hold the help text, note height should be changed
# to be the number of lines of help text.
text .help.frame -relief raised -height 10

# set the text widget to contain the actual text
.help.frame insert 0.0 {

Put all of the help you want to document between the curly brackets!

}

# make a button to remove the window when pressed
button .help.ok -text "OK"
-relief raised -borderwidth 2 -command {destroy .help}

# pack the text widget and the OK button into the window
.pack .help.frame -side top
.pack .help.ok -side bottom
Considerations for building a GUI

It is a good idea whenever possible to make the procs that you build reusable. Just like code in other languages, modularization is a good rule.

Build GUIs with the user in mind. Set reasonable defaults so the user will only have to type or choose minimum information. Don’t include rarely used options or keywords from McIDAS commands in your GUI. You may want to make one command into several GUIs if the functionality of the command is complex. Conversely, you may want to combine the functionality of several commands into one GUI. For example, you can include an option to display a map over an image immediately after it is displayed.
Sample GUI code

1. #!/usr/local/bin/wish -f
2. # The above line needs to be the first line in the script to specify which
3. # scripting language we will use, and where it's located.
4. #
5. #
6.  gu gui
7. #
8. #
9. # THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED.
10. #
11. #
12. # Set path to point to the -mcidas/gui0.2 directory as well
13. # as the Tcl/Tk options set on installation of Tcl/Tk.
14. #
15. set auto_path "-mcidas/gui0.2 $auto_path"
16. #
17. # call the proc to set the default options for this widget
18. #
19. setoptions
20. #
21. # Set up access to User Common. This is needed to call McIDAS
22. # commands outside of the command line. The variable $term is
23. # picked up as the first argument of the call to the GUI.
24. #
25. set term [lindex $argv 0]
26. #
27. #
28. # Based on the term number, we choose the file to read.
29. #
30. set file [format "UCTERM.00%" $term]  
31. if {$term > 9} {set file [format "UCTERM.0%" $term]}
32. #
33. # Read the file, which will contain the UC block attached to this
34. # terminal number, this is set to $uc variable.
35. set odline [open "$odline -chmod $term +6" r]
36. set uc [lindex $line 1]
37. close $odline
38. #
39. #
40. #
41. #
42. #
43. #
44. #
45. #
46. #
47. #
48. #
49. #
50. #
51. #
52. #
53. #
54. #
55. #
56. #
57. #
58. #
59. #
60. #
61. #
62. #
63. #

GUI for the Graphics utility application

Set the title which appears in the window manager
frame surrounding the GUI.
wm title "Edit Graphics Colors"

Set up a button which allows the user to call up the Help window.
The button will be made in the frame ".top"
button .top.help -text "Help" -command "Help"

Pack the help button on the right side of the .top frame.
pack .top.help -side right

define a message area, with a width of 500 pixels. The border
of the message area will be raised with a width of 1. The font
was set in the setoptions proc (external).
The -text field specifies the text to appear inside of the widget.
.message .msg -width 500 -relief raised
.font $messfont -borderwidth 1 -text "Utility for altering the graphics color table."
64. The graphics color table can be modified, saved and restored.
65. 
66. To run press OK button. To exit without performing an action, press Dismiss button.
67. 
68. # Create a frame which will hold all the command options.
69. frame .frame -borderwidth 10
70. 
71. # Create a frame which will hold the options which the user can select
72. # to be prompted with the widgets specific to that option.
73. frame .frame.option
74. 
75. # Make a button, which will prompt the user with a series of
76. # radio buttons when pressed.
77. menubutton .frame.option.menu -text "Option" -relief raised
78. \menu .frame.option.menu.choices
79. 
80. # Define the menu which appears when the menubutton is pressed.
81. menu .frame.option.menu.choices
82. 
83. # "add" all of the radio buttons to the menu. Define the label which
84. # will appear with the button, the variable which will be set when
85. # the button is pressed, the value the variable will be set to, and
86. # the command which will be performed when the button is pressed.
87. # Note, in this menu, when an option is changed, the old option
88. # will be removed from the screen with the "pack forget", and the
89. # new options widgets will be presented with a "pack".
90. 
91. # Radio button for "MAKE" option
92. .frame.option.menu.choices add radio -variable option \ 
93. -label "Edit Color" -value MAKE -command "
94. pack forget .frame.restore
95. pack forget .frame.save
96. pack forget .frame.make
97. 
98. # Radio button for "RESTORE" option
99. .frame.option.menu.choices add radio -variable option \ 
100. -label "Restore Table" -value REST -command "
101. pack forget .frame.restore
102. pack forget .frame.save
103. pack forget .frame.make
104. pack .frame.restore -side top -fill x
105. 
106. # Radio button for "SAVE" option
107. .frame.option.menu.choices add radio -variable option \ 
108. -label "Save Table" -value SAVE -command "
109. pack forget .frame.restore
110. pack forget .frame.make
111. pack .frame.save -side top -fill x
112. 
113. # Radio button for "RESET" option
114. .frame.option.menu.choices add radio -variable option \ 
115. -label "Reset" -value RESET -command "
116. pack forget .frame.restore
117. pack forget .frame.make
118. pack forget .frame.save"
119. 
120. # define an entry widget which will hold the value which was
121. # set by the radiobutton chosen above. Note the "textvariable"
122. # is the same as the "variable" in the radiobuttons above.
123. # The current value of the variable will be displayed in the
124. # entry widget, even as the variable changes values by the menu
125. # above. The width will be sunken and will be 5 characters wide.
126. #
127. entry .frame.option.entry -relief sunken -width 5 -textvariable option
128. 
129. # Pack the menubutton and entry button left to right
130. pack .frame.option.menu -side left
131. pack .frame.option.entry -side right
132. 
133. # Pack the option frame into ".frame".
134. pack .frame.option -side top -fill x
135. # Restore option - consists of a button, and an entry widget.
136. # Note that this frame is not packed at this time. It will not
137. # be packed until the RESTORE option is selected on the radio
138. # buttons above.
139.
140. frame .frame.restore
141.
142. # Button which when pressed runs the proc "Getgra". The text
143. # is "Graphics file name".
144. button .frame.restore.button -text "Graphics File Name" -command Getgra
145.
146. # Entry widget to show the value selected by the user in getgra
147. # proc. The variable associated with this entry is gratab.
148. entry .frame.restore.entry -textvariable gratab -relief sunken -width 12
149.
150. # Pack the button on the left and the entry on the right.
151. pack .frame.restore.button -side left
152. pack .frame.restore.entry -side right
153.
154. # save option
155.
156. frame .frame.save
157. label .frame.save.label -text "Graphics File Name"
158. entry .frame.save.entry -textvariable gratab -relief sunken -width 12
159. pack .frame.save.label -side left
160. pack .frame.save.entry -side right
161.
162. # make option
163.
164. # set the variables red, green, blue and clev to zero.
165. set red 0
166. set green 0
167. set blue 0
168. set clev 0
169.
170. # Create a frame for the MAKE option.
171. frame .frame.make
172.
173. # Create a color frame within the make option.
174. frame .frame.make.color
175.
176. # Create a button which when pressed will run the getcolor proc.
177. button .frame.make.color.button -text "Color Level" -command "getcolor"
178.
179. # Create an entry widget which will display the results of the
180. # getcolor proc.
181. entry .frame.make.color.entry -textvariable clev -relief sunken -width 3
182.
183. # red, green and blue (which are dereferenced with a $), were
184. # returned from the getcolor proc. The values are put into
185. # a format which can be used to specify a color for the
186. # swatch below.
187. set color [format "%%02x%%02x%%02x" $red $green $blue]
188.
189. # Make a frame which is nothing more than a color swatch - a
190. # rectangle of specified color.
191. frame .frame.make.color.swatch -width 200 -height 20 -bg $color
192.
193. # Pack the button on the left, the swatch to the right of it, and
194. # the entry widget on the right.
195. pack .frame.make.color.button -side left
196. pack .frame.make.color.swatch -side left -padx 20
197. pack .frame.make.color.entry -side right
198.
199. # Pack the color frame into the make frame.
200. pack .frame.make.color -side top -fill x
201
202. # Now we make a frame which will contain radio buttons allowing
203. # the user to specify how they will select the new color (by
204. # name or by sliders)
frame .frame.make.how
  # Make a radiobutton which allows the user to select a named color.
  # variable is "how", value will be set to "bymame" if chosen, the
  # label associated with the button is "Select Named Color", and
  # when radiobutton is selected the "name" frame will be displayed.
  # Since the users are specifying a named color, the red, green,
  # and blue sliders will not be needed, and thus will be "unpacked".
  radiobutton .frame.make.how.byname -variable how -value byname \ 
    -text "Select Named Color" -command {
      pack .frame.make.name -side top -fill x
      pack forget .frame.make.red
      pack forget .frame.make.green
      pack forget .frame.make.blue
    }
  # Now, make one to select the color by intensity levels.
  # Here the the variable will be set to "byval", and
  # the "name" frame will be "unpacked", as the red, blue
  # and green frames are brought into view. We also set the
  # values of the variable associated with the sliders
  # to the values of the chosen color level to start.
  radiobutton .frame.make.how.byval -variable how -value byval \ 
    -text "Select Color Intensities" -command {
      pack forget .frame.make.name
      pack .frame.make.red -side top -fill x
      pack .frame.make.green -side top -fill x
      pack .frame.make.blue -side top -fill x
      .frame.make.red set $red
      .frame.make.green set $green
      .frame.make.blue set $blue
    }
  # Pack byname and byval side by side.
  pack .frame.make.how.byname -side left
  pack .frame.make.how.byval -side left
  # Pack the how frame into the MAKE frame.
  pack .frame.make.how -side top -fill x -pady 10
  # Frame which will contain the color name to change the level to.
  frame .frame.make.name
  # Button which runs GetNamedColor proc
  button .frame.make.name.button -text "Color Name" -command GetNamedColor
  # Entry widget which contains the value set in the GetNamedColor
  # proc to the colorname variable.
  entry .frame.make.name.entry -textvariable colorname -width 12 -relief sunken
  # Pack the button and entry widgets side by side.
  pack .frame.make.name.button -side left
  pack .frame.make.name.entry -side right
  # Make a scale to set the red pigment of the color.
  # The scale will be horizontally oriented, will contain values
  # from 0 to 255, and will be 256 pixels long. The value will be
  # displayed as it changes, and the color of the slider is red.
  # As the slider is moved, the SetColor proc will be called to
  # update the color of the swatch with the changing values.
  # When a command is called from a scale widget, the current value
  # of the widget is appended to the proc call, so the proc actually
  # has 2 arguments, although only one is specified in the -command
  # option. That value is not actually used in the proc, but
  # because it will appear in the calling sequence, it must be
  # enumerated in the argument list. This is a subtle trick of
  # Tk to be aware of!
scale .frame.make.red -from 0 -to 255 -label "Red Intensity" -length 256
    -command {SetColor "num"}

# Green scale, which is largely the same as the red scale.
scale .frame.make.green -from 0 -to 255 -label "Green Intensity" -length 256
    -command {SetColor "num"}

# Blue scale, just like the red and green one - only it's blue!
scale .frame.make.blue -from 0 -to 255 -label "Blue Intensity" -length 256
    -command {SetColor "num"}

# Set the values of the sliders to the red, green and blue values
# which are associated with the color level, this will cause the
# the slider to move to this value.
.frame.make.red set $red
.frame.make.green set $green
.frame.make.blue set $blue

# This frame will be the bottom portion of the gui. It will hold
# the OK, and cancel buttons, and a bitmap of the SSEC logo.
.frame .bottom -relief raised

# OK button executes commands based on option.
# because the command is in {}s it will be evaluated when the
# button is pressed instead of when the GUI is started.
# The command is built based on the option which was chosen
# by the user.

button .bottom.ok -text OK -width 20
    -command {
        if {$option == "$REST"} {set command "GU REST $gratab"}
        if {$option == "$SAVE"} {set command "GU SAVE $gratab"}
        if {$option == "$RESET"} {set command "GU REST"}
        if {$option == "$MAKE"} {
            if {$show == "$byname"} {
                set col $colname
                if {$show == "$byval"} {
                    set col [format "$% % %$ $blue $green $red"]
                set command "GU MAKE $clev $col"
            }

# This is a proc we created to execute a McIDAS command.
    RunCommand $command
    }

# cancel button in case user decides to bail out, it will destroy
# the window.
.button.cancel -text Dismiss -command {destroy .} -width 20

# label containing the SSEC logo. It will change to an hourglass
# bitmap when RunCommand is executed. RunCommand uses the
# variable blabel to change the value of the bitmap.

label .bottom.label -bitmap @-mcidas/gui0.2/sseclogo.xbm
set blabel .bottom.label

# Pack the ok button, bitmap label, and cancel buttons side by side,
# with a 10 pixel padding on all sides.
.pack .bottom.ok -side left -padx 10 -pady 10
.pack .bottom.label -side left -padx 10 -pady 10
.pack .bottom.cancel -side right -padx 10 -pady 10

# Pack the top of the frame into the top of the main window.
# The frame will be expanded to fit into the window.
pack .top -side top -fill x
# Next, pack the message, expanding to fit
pack .msg -side top -fill x
# Now pack the frame which contains the command widgets.
pack .frame -side top -fill x
# OK, now we can pack the bottom frame of the widget
pack .bottom -side bottom -fill x
# Set the minimum dimensions allowed for the underlying window
# during interactive resizing.
wm minsize 10 10
# Help proc - display the help associated with this GUI
proc Help () {
  # define the toplevel window
toplevel .help
  # Set the title bar
  wm title .help "Help - Edit Graphics Colors"
  # set the minimum size
  wm minsize .help 10 10
  # create a text widget. The border is raised, it is 16 lines long,
  # the body of the widget is inserted in another step.
  # text .help.text -relief raised -height 16
  # insert the lines of text
  .help.text insert 0.0 {
    Option
    Select the option to perform on the graphics color table
    Edit Color Select the color to be edited, and the color name,
    or color intensities to set it to.
    Save Table Save the current frames graphics color table to a
    named file.
    Restore Table Restore a previously stored graphics color table
    to the current frame.
    Reset Reset the color table to the standard color table.
  }
  # make a button which will dismiss the window
  button .help.ok -text "OK"
  -relief raised -borderwidth 2 -command {destroy .help}

  # Pack the text and OK frame into the toplevel frame.
pack .help.text -side top
  pack .help.ok -side bottom
}

# Getgra lists graphics files which can be selected for restoring
# the graphics table
proc Getgra {} {
  # gratab is a global variable, it is also the variable attacted
  # to an entry widget above. It must be a global variable to be
  # updated in the entry widget of another frame when it changes
  # within a proc.
global gratab

# If there is already a window called .listgra - destroy it.
catch [destroy .listgra]

define the toplevel widget
toplevel .listgra

title, icon name and minimum size of the window created inside
this proc
wm title .listgra "Graphics Files Listing"
winfo .listgra "Graphics"
winfo .listgra "11"

# uc needs to be global, so we can access it here, we need it
# because we'll be running a McIDAS command.
global uc
message .listgra.msg -font -g 12 -adobe-times-medium-r-normal--*-180* -aspect 300 200 -relief raised -text "Select Graphics File by double clicking left mouse button."

Press Dismiss to exit.

# Make a frame which will contain the listbox and scrollbar
frame .listgra.frame -borderwidth 10

# Make a scrollbar. The command is to change the view of
# the listbox
scrollbar .listgra.frame.scroll 
-relief sunken -command "$listgra.frame.list yview"

# Make a listbox. Attach the scrollbar to it. The listbox will
# appear sunken, and will be 80x10 characters.
listbox .listgra.frame.list -yscroll "$listgra.frame.scroll set" 
-relief sunken -geometry 80x10 -setgrid 1

pack .listgra.frame.scroll -side right -fill y
pack .listgra.frame.list -side left -expand yes -fill y

# set a temporary variable to " ", so Tcl knows it's a character value
set temp " 

# Fire off a "GU LIST" command to McIDAS.
set gralist [open "|echo GU LIST | gu.mx $uc" r]

# grab the first line off the output to make a title bar
global labl
label .listgra.labell -width 79
gets $gralist labl

# Grab the second line off the output and toss it.
listgra.labell configure -text "$labl"
gets $gralist lab2

# Get each line, of output, and insert it into the listbox.
while {([gets $gralist line] > -1)} {
    .listgra.frame.list insert end "$line"
}

# Close the pipe for the McIDAS - failure to do so will leave
# defunct processes around.
catch [close $gralist]

# Set the action within {} to what occurs when a double click of
# the button takes place. Within {} is set "thisval" to the
# line which was selected, we grab characters 4-15 of that line
# we set the global variable gratab to this value, and it will
# appear in the entry widget. Then we destroy the widget.
bind .listgra.frame.list <Double-1> 
  {set thisval [selection get] 
   set temp [string range $thisval 4 15] 
   set gratab $temp 
   destroy .listgra 
  }

# We make a button for the user to exit without making a selection.
# button .listgra.ok -text Dismiss -command "destroy .listgra"

# Pack the message, title bar, widget frame, and OK button into
# the toplevel window top to bottom.
pack .listgra.msg -side top -fill x
pack .listgra.label1 -side top -anchor w
pack .listgra.frame -side top
pack .listgra.ok -side bottom

# Works exactly as does the getgra proc above, but doesn't execute
# a McIDAS command. Instead it reads in McIDAS MCRGB.TXT file.

proc GetNamedColor {} {
catch {destroy .colors}
toplevel .colors
wm title .colors "Image Listing"
winfo iconname .colors "Images"
winfo minsize .colors 11

message .colors.msg -font -Adobe-times-medium-r-normal---*180* 
  -aspect 300 -relief raised
  -text "Select Image by double clicking left mouse button."

frame .colors.frame -borderwidth 10
scrollbar .colors.frame.scroll -relief sunken 
  -command ".colors.frame.list yview"
listbox .colors.frame.list -yscroll ".colors.frame.scroll set" 
  -relief sunken -geometry 20x10 -setgrid 1
pack .colors.frame.scroll -side right -fill y
pack .colors.frame.list -side left -expand yes -fill y
set collist [open "MCRGB.TXT" r]

# These variables need to be global because we access this info
# in the main GUI.
global i
global reds
global greens
global names
global blues

set i 0

# Read in the contents of MCRGB.TXT (collist), and store the
# values in the names, reds, greens, and blues array
while {[gets $collist line] > -1} {
  .colors.frame.list insert end [lindex $line 3]
  set names($i) [lindex $line 3]
  set reds($i) [lindex $line 0]
  set greens($i) [lindex $line 1]
  set blues($i) [lindex $line 2]
  incr i
}

# close the pipe
catch {close $collist}
global colorno

global lowno

# set a double click event to do the following between {}.
# Events within {} include getting, the line, getting the
555. # colormame from the line, making it lowercase, and calling
556. # SetColor proc to change the swatch to the selected color.
557. bind .colors.frame.list <Double-1> \
558. [set color [selection get] \
559. set cName [string trim $cname] \
560. set colormame [string tolower $colormame] \
561. SetColor "name" 0 \
562. destroy .colors \
563. }
564. 565. # Pack all of the widgets into the toplevel window.
566. button .colors.ok -text Dismiss -command "destroy .colors" \
567. pack .colors.msg -side top -fill x \
568. pack .colors.frame -side top \
569. pack .colors.ok -side bottom \
570. 571. }
572. 573. 574. # The SetColor proc sets the color of the swatch in the MAKE 
575. # proc SetColor {how junk} { 
576. proc SetColor {how junk} { 
577. 578. # We need access to all of these variables set in other procs. 
579. global lowname 
580. global colormame 
581. global red 
582. global green 
583. global blue 
584. global i 
585. global reds 
586. global greens 
587. global blues 
588. set j 0 
589. 590. # If they chose a named color, figure out which color they picked, 
591. # set red, green, and blue to that value. 
592. if {$show == "name"} { 
593. while { $j < $i } { 
594. global names 
595. if {$colormame == $names($j)}{ 
596. set red $reds($j) 
597. set green $greens($j) 
598. set blue $blues($j) 
599. incr j 
600. }
601. 602. 603. # Set color variable to format, #RRGGBB where RR GG BB 
604. # hex representation of the Red Green and Blue color levels. 
605. set color [format "%02x%02x%02x" $red $green $blue] 
606. 607. # Set the swatch to chosen color 
608. .frame.make.color.swatch config -bg $color 
609. 610. # If they chose a color by color levels, get those values off of 
611. # the scale widgets 
612. } else { 
613. set red [.frame.make.red get] 
614. set blue [.frame.make.blue get] 
615. set green [.frame.make.green get] 
616. 617. # Set color variable to format, #RRGGBB where RR GG BB 
618. # hex representation of the Red Green and Blue color levels. 
619. set color [format "%02x%02x%02x" $red $green $blue] 
620. 621. 622. # Set the swatch to chosen color 
623. .frame.make.color.swatch config -bg $color 
624. }
proc getcolor {} {
    # getcol - display McIDAS graphics colors, allowing the
    # the user to choose one for their application.
    # These are the global variables we set or use inside of this proc.
    # global uc
    # global lev
    # global clev
    # global red
    # global green
    # global blue
    # If there is already a colors widget out there - destroy it.
    catch [destroy .colors]
    toplevel .colors
    wm title .colors "Color"
    wm minsize .colors 1 1
    message .colors.msg -font -Adobe-times-medium-r-normal--*-180* -relief sunken -text "Select Color Level by clicking on the color"
    Make a frame which will hold the buttons.
    frame .colors.frame
    # Start up a McIDAS command to list out the current RGB levels of the graphics color levels.
    set guout [open |echo GU TAB | gu.mx $uc| x]
    # Retrieve the number of Graphics levels of this session from user common.
    set maxlev [UCVal 500]
    # Go through all of the output from the GU output.
    # while {[gets $guout guine] > -1} {
    # Determine which color level this is.
    # if {$level <= 9} { set level [string range $guine 3 4] }
    # grab the RGB values for this level
    # set ired [string range $guine 35 37]
    # set igreen [string range $guine 30 32]
    # set iblue [string range $guine 24 26]
    # make sure its a good level
    # if {$level >= 0 && $level <= $maxlev} {
    # Set rgb variable to #RRGGBB format
    # set rgb [format "#%02x%02x%02x" $ired $igreen $iblue]
    # Make a frame for each level - name it lev1, lev2, lev3, etc.
    frame .colors.frame.lev#$level
    # Make a button which has a bitmap label which is a box, the color
    # of the RGB levels we retrieved for this level. The $level, ired,
    # igreen, and iblue will be dereferenced now. The swatch will
    # be set to this RGB level when this button is pressed.
    button .colors.frame.lev#$level.button -bitmap @-mcidas/gui0.2/box.xbm -foreground $rgb -command "set lev $level ;set clev $level; set red $ired set green $igreen set blue $iblue .frame.make.color.swatch config -bg $rgb destroy .colors"

    # Writing GUIs for McIDAS using Tcl/Tk
    # McIDAS Developer/Operator Training
    # October 24, 1995
}
-activebackground gray
-activeforeground $rgb

# Make a label which has the level number as the text
label .colors.frame.lev$level.label -text $level

# Pack the button, and label into each levels frame
pack append .colors.frame.lev$level\n    .colors.frame.lev$level.label {left}\n    .colors.frame.lev$level.button {right}

# Pack each levels frame into the large frame
pack append .colors.frame \n    .colors.frame.lev$level{top fill}

} \n
# close the pipe
catch {close $gout}

# button to dismiss the GUI without taking action.
button .colors.dismiss -command "destroy .colors" -text "Dismiss"
pack append .colors \n    .colors.msg {top fill}\n    .colors.frame {top fill}\n    .colors.dismiss {top fill}

}
Below is a solution set for the in-class assignment.

```tcl
#!/usr/local/bin/wish -f
# THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED.
# set path to access library procs
set auto_path "-mcidas/data -mcidas/gui0.2 $auto_path"
# set default font and colors from gui.options file
setoptions

# set up to access McIDAS session
set term [lindex $argv 0]
set file [format "UCTERM.00$s" $term]
if {$term > 9} {set file [format "UCTERM.0$s" $term]}
set odline [open $odline -dev $file -mode 6]
gets $odline line
set uc [lindex $line 1]
close $odline

# GUI for MUG demo to run MUGAREA command
wm title "MUG Demo Command"

# Make a frame to contain the help button in the upper right corner
frame .top
button .top.help -text "Help" -command mkHelp
pack .top.help -side right

# set the message area at the top of the gui. Give a brief
# explanation of the command. The font will be the default font
# from gui.options. The width of the ZZ
message .msg -width 500 -relief raised
messagefont -borderwidth 1 -text "Retrieve 2 McIDAS GRIDS in an
AREA format via ADDS, and combine them with arithmetic function,
sending result to a McIDAS AREA formatted file. To execute press OK
button. To exit without performing an action, press Dismiss Button."

# Make a frame which will contain the body of the GUI, place all of
# the widgets here.
frame .frame -borderwidth 10

# set the default source dataset name
set source_ds "MUG/MRFT"

# frame to hold the source dataset name widgets
frame .frame.source

# label describing the entry widget value to the right
label .frame.source.label -text "Source Dataset Name"

# entry widget holding the source dataset name
entry .frame.source.entry -textvariable source_ds -width 20 -relief sunken
```
# pack widgets left and right
.pack .frame.source .side -left
.pack .frame.source .side -right

# pack the source dataset frame into main frame
.pack .frame.source .side top -fill x

# frame to hold the position widgets
.frame .frame.pos1

# button which will run the list_adde_image proc
.button .frame.pos1 .button -text "Source Position 1"
.command [set which first; list_adde_image]

# entry which will hold the results of the position selection
.entry .frame.pos1 .entry -textvariable pos1 -width 5 -relief sunken

# pack the widgets left and right
.pack .frame.pos1 .button .side -left
.pack .frame.pos1 .entry .side -right

# install the position 1 frame in the GUI
.pack .frame.pos1 .side top -fill x

.frame .frame.pos2

# frame to hold the position widgets

# button which will run the list_adde_image proc
.button .frame.pos2 .button -text "Source Position 2"
.command [set which second; list_adde_image]

# entry which will hold the results of the position selection
.entry .frame.pos2 .entry -textvariable pos2 -width 5 -relief sunken

# pack the widgets left and right
.pack .frame.pos2 .button .side -left
.pack .frame.pos2 .entry .side -right

# install the position 2 frame in the GUI
.pack .frame.pos2 .side top -fill x

# frame to hold the options
.frame .frame.option

# radiobutton with text of ADD which will set variable option to ADD
.radiobutton .frame.option.add .text "Add"
-variable option .value ADD .width 15

# radiobutton with text of Subtract which will set variable option to SUBTRACT
.radiobutton .frame.option.subtract .text "Subtract"
-variable option .value SUBTRACT .width 15

# radiobutton with text of Average which will set variable option to AVERAGE
.radiobutton .frame.option.average .text "Average"
-variable option .value AVERAGE .width 15

# pack all of the radiobuttons onto a line
.pack .frame.option.add .frame.option.subtract .frame.option.average .side -left

# install the options on the GUI
.pack .frame.option .side top -fill x

# frame for destination dataset name
.frame .frame.dest

# label describing entry field on the right
.label .frame.dest .label -text "Destination Dataset Name"
# entry field holding the value of the destination dataset name
entry .frame.dest.entry -textvariable dest -width 20 -relief sunken

# pack the label and entry, left to right
pack .frame.dest.label -side left
pack .frame.dest.entry -side right

# install the destination name widgets
pack .frame.dest -side top -fill x

# frame for destination position number
frame .frame.dpos

# label describing entry
label .frame.dpos.label -text "Destination Position"

# entry holding value of the destination position
entry .frame.dpos.entry -textvariable dpos -width 5 -relief sunken

# pack the label & entry left and right
pack .frame.dpos.label -side left
pack .frame.dpos.entry -side right

# install the destination position number widgets
pack .frame.dpos -side top -fill x

# make a frame which will hold the OK and cancel buttons and the SSEC logo
frame .bottom -relief raised

# a button to execute the command. All variables will be evaluated as the value they are when the button is pressed.
button .bottom.ok -text OK -width 20 \
-command {

    # set command to the uppercase McIDAS command
    set command to the uppercase McIDAS command
    which you will be executing
    set dsname [format "%s.%s" $dest $dpos]
    set command "MCDIAS $source ds $pos $option $pos2 $dsname"
    runcommand executes the command and routes the output to the text window
    RunCommand $command

    } -width 20

# a button to cancel the command without executing anything
button .bottom.cancel -text Dismiss -command {destroy .} -width 20

# a ssec logo between the OK and the cancel buttons
label .bottom.label -bitmap @-mcdas/gui2/sseclogo.xbm

# set a global variable ‘blabel’ to the label widget, so that we can change the bitmap of the label to an hourglass when the command is running.
set blabel .bottom.label

# Pack the OK, & cancel buttons and label into the bottom frame,
# padding 10 pixels around the button (x and y direction), order
# will be in order specified OK, label, cancel (left to right)
pack .bottom.ok -side left -padx 10 -pady 10
pack .bottom.label -side left -padx 10 -pady 10
pack .bottom.cancel -side left -padx 10 -pady 10

# Pack the message, command and OK buttons into the base widget
pack .top -side top -fill x
205: pack .msg -side top -fill x
206: pack .frame -side top -fill x
207: pack .bottom -side bottom -fill x
208:
209: # set the minimum size for the gui
210: wm minsize . 10 10
211:
212:
213: # Proc for help.
214: proc mkHelp {} {
215:
216: # name the toplevel widget
217: toplevel .help
218:
219: # set the title (change 'command name' to your gui name)
220: wm title .help "Help - command name"
221:
222: # set the minimum size for the help window
223: wm minsize .help 10 10
224:
225: # make a text widget, to hold the help text, note height should
226: # be changed to be the number of lines of help text.
227: text .help.frame -relief raised -height 15
228:
229: # set the text widget to contain the actual text
230: .help.frame insert 0.0 {
231:
232: Source Dataset Name Name of source dataset which contains grids
233: served as ADDE images
234:
235: Source Position 1 position of one of the grids to perform option on
236:
237: Source Position 2 position of other grids to perform option on
238:
239: Option Add, Subtract or Average option
240:
241: Destination Dataset name Name of dataset to write resultant image to
242:
243: Destination Position Position of destination image in dataset
244: }
245:
246: # make a button to remove the window when pressed
247: button .help.ok -text " OK " -relief raised \n248: -borderwidth 2 -command {destroy .help}
249:
250: # pack the text widget and the OK button into the window
251: pack .help.frame -side top
252: pack .help.ok -side bottom
253:
254: }
255:
256:
257:
258:
259: # mkListArea w
260:
261: # Create a top-level window containing a listbox showing a bunch of
262: # McIDAS Images
263: # Arguments:
264: # w -Name to use for new top-level window.
265:
266:
267:
268: # THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED.
269:
270: proc list_adde_image {} {
271:
272: # destroy a .listarea widget if it exists
273: catch {destroy .listarea}
274:
275: # label of the main window, set it to the hourglass while we get
276: # listing
global blabel

$blabel configure -bitmap @-mcidas/gui0.1/hourglass.xbm

update

toplevel .listarea

# set the characteristics of the window
wm title .listarea "Image Listing"
wm iconname .listarea "Images"
wminsize .listarea 1 1

echo $rc

# bring in uc and which as global variables - we'll use them later
foreach {global uc global which} {eval \$\_

# make a message widget to hold the instructions
message .listarea.msg -font -Adobe-times-medium-r-normal--*-180* -aspect 300 -relief raised
-text "Select Image by double clicking left mouse button.
Press Dismiss to exit."

# make a frame to hold the workings of the widget
frame .listarea.frame -borderwidth 10

# make a scrollbar to allow scrolling of the listing
scrollbar .listarea.frame.scroll -relief sunken
-command "$listarea.frame.list yview"

# make a listbox to hold the image listings
listbox .listarea.frame.list -yscroll "$listarea.frame.scroll set"
-relief sunken
-geometry 80x10 -setgrid 1

# pack the scrollbar on the right of the listbox
pack append .listarea.frame .listarea.frame.scroll {right fill}
.pack .listarea.frame.list {left expand fill}

set localarea ""

# set lalist to be the pipe from the IMGLIST command
set lalist [open "$mcidas/gui0.1/imglst.0.mux" r]

# set the label to the first line of output
global lab1
label .listarea.label1 -width 79
gets $lalist lab1
.listarea.label1 configure -text "$lab1"

# get the next 3 lines of output and discard them
gets $lalist lab2
gets $lalist lab2
gets $lalist lab2

# retrieve all of the rest of the lines and display them in the listbox
while {($gets $lalist line) > -1} {
.listarea.frame.list insert end $line
}

# set the label to the SSEC logo again
$blabel configure -bitmap @-mcidas/gui0.1/sseclogo.xbm
update

# close the pipe
close $lalist

# bind the double left button click
bind .listarea.frame.list <Double-1> \n
# get the selected line
{set areal [selection get]}

Writing GUIs for McIDAS using Tcl/Tk

McIDAS Developer/Operator Training

October 24, 1995
#set a variable to position 2 - 5 of the line
set localarea [string range $areal 2 5]

# set pos 1 if they are to set the first position
if { $which == "first" } {
    set pos1 $localarea
} else {
    set pos2 $localarea
}
destroy .listarea

# make a button to get rid of the widget
button .listarea.ok -text Dismiss -command "destroy .listarea"

# pack the widgets into the window
pack .listarea.msg -side top -fill x
pack .listarea.label2 -side top -anchor w
pack .listarea.frame -side top
pack .listarea.ok -side bottom

Resources

Tcl/Tk books
Tcl and the Tk Toolkit
John K. Ousterhout
Addison Wesley
ISBN 0-201-63337-X

Practical Programming in Tcl and Tk
Brent B. Welch
Prentice Hall

Internet resources
World Wide Web site
http://www.sunlabs.com/research/tcl

Tcl/Tk distribution sites
ftp://ftp.cs.berkeley.edu/ucb/tcl/

Tcl/Tk extensions, programs, utilities

List of Frequently Asked Questions
ftp://rtfm.mit.edu/pub/usenet-by-group/comp.lang.tcl/

Usenet newsgroup
comp.lang.tcl

McIDAS GUI ftp site
ftp.ssec.wisc.edu - email gui@ssec.wisc.edu for ftp instructions
and password.

Alternatives to Tcl
Tk for perl
Tkinter for Python
McIDAS Navigation and Calibration Subsystems

Presented by
Dave Santek
McIDAS Applications Project Leader

Session 5
McIDAS Developer/Operator Training
October 23-25, 1995
Table of Contents

Overview .................................................. 5-1
  Terminology ........................................... 5-1
  Source file naming conventions .................. 5-2
  Applications interfaces .............................. 5-2
  Components ........................................... 5-3
  API ...................................................... 5-4
  Internal function names ................................ 5-4
Under the hood on OS/2 ................................. 5-5
Under the hood on Unix ................................. 5-6
Guidelines for writing a module ...................... 5-8
References ............................................... 5-10
Overview

A redesign of the McIDAS Area file format took place in the mid-1980s to accommodate the ever increasing types of remotely sensed data. A generalized method for storing, navigating, and calibrating these data was developed.

The design allows the addition of new data sets from a variety of platforms (satellite, aircraft, radar, etc.) with no changes to existing software or file format. This is done by defining a file format (Areas) that can accommodate multibanded, multibyte data along with a variety of ancillary data. In addition, it is recognized that two basic functions for working with remotely sensed data needs to be handled in software: navigation and calibration. This is done by defining an API (subroutine names, calling sequence, and functionality) that all navigation and calibration modules will adhere to, and a mechanism to access the appropriate module at application run time.

Terminology

The terms defined below are used in this section.

- **ancillary data**: additional information needed to identify, quantitate and manipulate image data, including the directory, navigation and calibration blocks
- **calibration**: the process of converting data values received from the satellite to a useful, physical quantity such as temperature, radiance, or albedo
- **DLL**: Dynamic Link Library; the library of routines used in dynamic linking
- **dynamic linking**: subprograms loaded at run time
- **navigation**: the process of transforming row/column in the image file to lat/lon and vice versa
- **slot number**: 1, 2 or 3 to allow for simultaneous use of up to three navigation and calibration modules
- **static linking**: subprograms included at compile/link time
Source file naming conventions

Use the naming conventions below.

<table>
<thead>
<tr>
<th>Calibration</th>
<th>Navigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVS: MUKBXnam</td>
<td>MUNVXnam</td>
</tr>
<tr>
<td>OS/2 and Unix: KBXname.DLM</td>
<td>NVXname.DLM</td>
</tr>
</tbody>
</table>

where name represents the name of the data format, such as GVAR, TIRO, AAA, etc. In MVS, the naming convention only allows three characters; so in all but one case, the four character names are truncated to three. The one exception is the GVAR navigation, which is in MUNVXGVR.

Applications interfaces

The McIDAS applications interface to the navigation and calibration modules is totally independent of the modules themselves. This division allows new modules to be written without affecting the applications. A few of these interfaces are described below.

For navigation, a call to nvset will load and initialize the appropriate navigation module. The API functions (discussed in the API section) can then be called to perform the navigation transformations. Other applications will make use of higher level functions (itvll and illtv), hiding some of the details.

For calibration, a call to araopt (set Area options) will set up a calibration module only if the physical quantity, which is the UNIT option in araopt, is different than that stored in the image file, which is word 53 of the Area directory. Calls to redara (read a line of data) will make use of a calibration module only if needed. Most applications do not call the API functions directly; two notable exceptions are IMGPROBE and D.
Components

Below `nvset` and `araopt` are the subroutines that provide the interface to the dynamic navigation and calibration modules; they are `nvprep` and `kbprep`, respectively. `nvprep` and `kbprep` build the name of the module to load, based on the slot number and type of data.

The slot number (1, 2 or 3) allows for the simultaneous use of up to three different navigation and calibration modules. The maximum of three was an arbitrary decision. Up to this point, no applications have made use of more than two navigation modules. Being able to remap a satellite image into a different projection illustrates this need. For example, to remap a GOES satellite image into a Mercator projection, the application would load the GOES navigation into slot 1 and the Mercator projection into slot 2.

There have been problems, though, with a limit of three calibration modules on MVS, where up to six Area files can be read at one time. In the ADDE (Abstract Data Distribution Environment) this is not an issue, since every file read is provided by a different server.

The data type (GVAR, TIRO, AAA, etc.) is also needed to construct the DLL to load. For calibration, the data type is stored in word 52 of the Area directory; for navigation, it is stored in the first word of the navigation block. For GOES-8, the type is GVAR for both navigation and calibration. Thus, the name constructed, using slot number 1, for navigation is NV1GVAR. It will load the module `nv1gvar.dll`. 
The navigation and calibration modules are not accessed through their internal function names (as seen in the next section), but rather through the names below, allowing more than one module to be used simultaneously.

The numeric is the *slot number*. To further describe the remap scenario presented in components, the application would be able to do navigation transformations in the Mercator projection by calling NV2EAS and NV2SAE, and in the GOES projection by calling NV1EAS and NV1SAE.

Navigation:
- nv1ini, nv1eas, nv1sae, nv1opt
- nv2ini, nv2eas, nv2sae, nv2opt
- nv3ini, nv3eas, nv3sae, nv3opt

Calibration:
- kb1ini, kb1cal, kb1opt
- kb2ini, kb2cal, kb2opt
- kb3ini, kb3cal, kb3opt

*Internal function names*

Each navigation or calibration module has the following internal function names. The calling sequence and function of these routines are identical across all data types. This is the key to adding new data types or navigation algorithms transparent to the applications.

<table>
<thead>
<tr>
<th>Entry points</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigation:</td>
<td></td>
</tr>
<tr>
<td>nvxini</td>
<td>initialization</td>
</tr>
<tr>
<td>nvxeas</td>
<td>earth-to-satellite transformation</td>
</tr>
<tr>
<td>nvsae</td>
<td>satellite-to-earth transformation</td>
</tr>
<tr>
<td>nvsopt</td>
<td>additional operations</td>
</tr>
<tr>
<td></td>
<td>(satellite sub-point, for example)</td>
</tr>
<tr>
<td>Calibration:</td>
<td></td>
</tr>
<tr>
<td>kbxini</td>
<td>initialization</td>
</tr>
<tr>
<td>kbxcal</td>
<td>calibration for an array of data values</td>
</tr>
<tr>
<td>kbxopt</td>
<td>additional operations (what type of calibration is possible, for example)</td>
</tr>
</tbody>
</table>

Now let’s go ‘Under the Hood’ to look at how the routines in the DLLs are actually accessed for OS/2 and Unix.
Under the hood on OS/2

This section details the processes well below the API that affect the
dynamic linking for the navigation and calibration modules. This
information is not needed to write a navigation or calibration module.

A call to getdhll, in os2glue.c, loads the DLL (nv1gvar.dll, for example)
through the system call DosLoadModule, and returns a file handle. To
access the individual routines in the DLL, the addresses are obtained by
calling getadr (in os2glue.c), with the file handle, which calls
DosQueryProcAddr. In addition to the API routines listed above, the
routine dllsub (in dllsub.for) is included in the DLLs to set up the McIDAS
environment (access to User Common and the LW file subsystem). The list
of entry points in the DLLs is contained in navlib.def and callib.def.
dllsub is called from nvpref and kbprep.

COMMON/NVXCOM/ stores the addresses of the API routines. An
additional layer was imposed to map the API (nv1ini, etc.) calls to the
appropriate entry point (nvxini, etc.) in the DLL. The API routines are
stored in .for files of the same name (nv1ini.for, for example). When an
API routine is called, it in turn calls one of the icalln routines (in
os2glue.c), passing the address of the entry point in the DLL along with all
the parameters. The icalln is a mechanism to call a subroutine by address,
rather than by name. The code for nv1eas and icall6 is listed below.

Extracted from nv1eas.for:

    INTEGER FUNCTION NV1EAS(X1,X2,X3,X4,X5,X6)
    COMMON/NVXCOM/THANDL (3), NVADDR (3,4)
    NV1EAS=ICALL6(NVADDR (1,3),X1,X2,X3,X4,X5,X6)
    RETURN
    END

Extracted from os2glue.c:

    long
    icall6 (Fint (**isub)(), void *a, void *b, void *c, void *d, void *e, void *f)
    {
        return (**isub)( a, b, c, d, e, f);
    }
Under the hood on Unix

This section details how dynamic linking is simulated for McIDAS-X. This information may be useful when debugging a navigation or calibration module.

When McIDAS was ported to Unix, a standard dynamic loading feature was not identified. Even shared libraries were implemented differently; some requiring relinking when the shared library was updated. It has been five years since we first did an investigation and we have begun to look into it again.

As an interim measure, we have decided to simulate dynamic linking. This is done by compiling the .DLM files as subroutines, storing them in the library, and then statically linking to the applications. Since the entry point names are the same in each navigation or calibration module, the names must be changed to store them in the library. A preprocessor program, convdlm (in convdlm.fp), automatically modifies the entry point names in a unique way to avoid duplication of names. Below is a skeleton of a calibration module (kbxtest.dlm), along with the output from convdlm, kbxtest1.f.

```

**kbxtest.dlm**

```

1. INTEGER FUNCTION KBXINI (CIN, COUT, IOPT)
2. COMMON/MOTEST/JTYPE, ISOU, IDES, JOPT
3. KBXINI=0
4. RETURN
5. END
6. INTEGER FUNCTION KBXCAL (CALB, IDIR, NVAL, IBAND, IBUF)
7. COMMON/MOTEST/JTYPE, ISOU, IDES, JOPT
8. CALL MAKTAB (ITAB, ISCAL(1), ISCAL(2), ISCAL(3), ISCAL(4))
9. CALL MPXTB (NVAL, ISOU, IDES, IBUF, ITAB)
10. KBXCAL=0
11. RETURN
12. END
13. INTEGER FUNCTION KBOXPT (CFUNC, IIN, IOUT)
14. COMMON/MOTEST/JTYPE, ISOU, IDES, JOPT
15. RETURN
16. END
17. SUBROUTINE MAKTAB (ITAB, INLO, INHI, IBLO, IBHI)
18. RETURN
19. END
20. END
21. END
22. END
23. END
24. END
25. END
26. END
27. END
28. END
29. END
30. END

---

McIDAS Navigation and Calibration Modules

McIDAS Developer/Operator Training

October 24, 1995
kbxtest1.f

1. INTEGER FUNCTION KB1INItest
2. + (CIN, COUT, ILOPT)
3. 4. COMMON/M0TESTtestkb1/
5. + JTYPE, ISOU, IDES, JOPT
6. 7. KB1INItest
8. + =0
9. RETURN
10. END
11. 12. INTEGER FUNCTION KB1CALtest
13. + CALB, IDIR, NVAL, IBAND, IBUF)
14. 15. COMMON/M0TESTtestkb1/
16. + JTYPE, ISOU, IDES, JOPT
17. 18. CALL MAKTABtestkb1
19. + (ITAB, ISCAL(1), ISCAL(2), ISCAL(3), ISCAL(4))
20. CALL MPAXTB(NVAL, ISOU, IDES, IBUF, ITAB)
21. 22. KB1CALtest
23. + =0
24. RETURN
25. END
26. 27. INTEGER FUNCTION KB1OPTtest
28. + (CFUNC, IIN, IOUT)
29. 30. COMMON/M0TESTtestkb1/
31. + JTYPE, ISOU, IDES, JOPT
32. 33. RETURN
34. END
35. 36. SUBROUTINE MAKTABtestkb1
37. + (ITAB, INLO, INHI, IBLO, IBHI)
38. RETURN
39. END

All subroutine, function, and common block names are modified.
The function kbxini becomes kb1inittest. The common block common/m0test/ becomes common/m0testtestkb1/. Note that maktab is modified because it is inline, but the call to mpixtb is not.
Guidelines for writing a module

Here are some things to keep in mind when writing a navigation or calibration module. Most of the restrictions are related to preprocessing the routine with `convd1m`.

- Write the module in Fortran. At this time, `convd1m` only runs against Fortran. To use a C module on Unix, you would have to modify the names manually. C modules work fine in OS/2.

- Write the module in uppercase. When `convd1m` was written, all the modules were in uppercase; navigation and calibration modules originated from the mainframe and ported to OS/2. The only recognized comment line begins with a `C`.

- Do not use the words SUBROUTINE, FUNCTION or COMMON in comment lines or message lines (such as DDEST). Also, in these lines, do not enter the `name` of any subroutine, function, or common block in uppercase.

- Do not use the Fortran ENTRY statement; `convd1m` does not recognize or handle it correctly.

- Do not imbed a function call within another function call if both functions are in the module. `convd1m` will not be able to handle the expansion and will print an error message and then exit. For example, if SUBROUTINE ASUB(K) and FUNCTION BFUNC(J) are both in the .DLM, the following is illegal:

  ```fortran
  CALL ASUB ( BFUNC(10) )
  ```

- Do not allow routines that expect character variables to be passed in (KBXINI, KBXOPT, for example), to declare the variables as CHARACTER(*). The length of the variable is not passed along. So, in KBXINI and KBXOPT, the lengths are known and are so declared as CHARACTER*4.

- Do not output text (SDEST, Fortran WRITE, etc.). This causes problems with ADDE servers that send data through standard output. You can imbed DDEST calls for debugging, but they will only be output with non-ADDE commands.
Run the process **convdlm** manually if you want to examine .f files, as they are automatically deleted during compiling. When compiling .DLMs on Unix, **convdlm** reads the .DLM file and outputs three .f files: kbxttest.dlm becomes kbxttest1.f, kbxttest2.f, and kbxttest3.f. These files are compiled, so any compiler warnings and/or errors refer to these files, which have different line numbers for the statements than the .DLMs. To run **convdlm**, use: **convdlm filename** for example: **convdlm kbxttest.dlm**.
References

*McIDAS Programmer's Manual*
Preliminary issue of Chapters 4, 5 and 6
October 1995

Designing and Implementing Calibration Modules

Presented by
Dave Santek
McIDAS Applications Project Leader

Session 6
McIDAS Developer/Operator Training
October 23-25, 1995
Table of Contents

Overview ................................................. 6-1
    Terminology ....................................... 6-1

Calibration module design ............................. 6-2
    Module requirements ............................... 6-2
    Storing parameters ............................... 6-3
    Performance: computation vs. look-up tables. 6-5

Structure of a calibration module ................... 6-7

Integrating calibration modules into McIDAS ........ 6-9
    McIDAS-OS2 ....................................... 6-9
    McIDAS-X ........................................ 6-9

Sample program ....................................... 6-10

References ............................................ 6-14
Overview

The McIDAS calibration subsystem is designed to be extensible, so that new data or calibration techniques for existing data types can be included. Thus, you can define calibration modules that will allow McIDAS applications to view the data that you prescribe.

This training session presents the issues that you should resolve when designing a calibration module. It also provides a sample module. It is assumed that you have a basic understanding of the McIDAS Area file format and a conceptual understanding of calibration and how it relates to remotely sensed data.

Terminology

The terms defined below are used in this section.

**band**
- spectral band; band 4 for the GOES-8 Imager is 10.7 microns (infrared); see Appendix G of the McIDAS-X Users Guide

**data value**
- 1-, 2-, or 4-byte quantity; sometimes called *pixel*

**physical quantity**
- radiance, temperature, albedo, etc.; sometimes called *unit*

**sensor source number**
- SS number stored in word 3 of the Area directory; 70 is the GOES-8 Imager; see Appendix H of the McIDAS-X Users Guide

**sensor type**
- up to four characters indicating the sensor; it is stored in word 52 of the Area directory; GVAR is the sensor on GOES-8

**slot**
- the number 1, 2 or 3 to allow for loading of up to three dynamic modules

**unit**
- measurement standard, such as Celsius or meters
Calibration module design

All calibration modules are built using the same framework. They must conform to the McIDAS convention for functionality, names of functions, and types of arguments. This standardization allows applications to make use of these modules in a generic, yet powerful way. It is usually not necessary for applications to have any private knowledge of the data with which it is working; the calibration interface provides a means to acquire certain aspects that are common to all.

Although the framework is rigid, you must resolve certain implementation issues. Two key issues, which are described later, are the storing of calibration parameters and performance.

Module requirements

Each calibration module has identical function names and interface, and performs similar operations. The actual algorithm for calibrating the data is hidden from the application, regardless of the calibration type.

All calibration modules contain the following three functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kbxini</td>
<td>initializes and verifies the requested calibration</td>
</tr>
<tr>
<td>kbxcal</td>
<td>calibrates the data</td>
</tr>
<tr>
<td>kbxopt</td>
<td>provides additional operations, which are usually queries from the application</td>
</tr>
</tbody>
</table>

The structure of these functions is designed so that any required ancillary data is passed in as arguments or handled by kbxopt. An exception is the access to the calibration block. The calibration subsystem was created about 10 years ago, before the use of calibration blocks. Currently, calibration blocks are handled by reading the disk from within the module. We are investigating changing this, due to the new ADDE paradigm, which presents a different view of the data to the applications.

All the functions either return the requested data or a bad status. They should never terminate, but rather rely on the exception handling of the application.
Storing parameters

Most of the current McIDAS calibration modules store parameters in these locations:

- Area directory
- calibration block
- line prefix
- LW file
- none

Area directory

The MSAT (Meteosat) calibration parameters are stored in the Area directory. Although the entire image only has three constant parameters, they do change twice per day. The Area directory is not the recommended storage location for your parameters because of the lack of available words in the directory. In retrospect, the parameters should have been placed in a calibration block.

Calibration block

The AAA (GOES-7 Mode AAA), QTIR (Quick AVHRR), PRD (Product), and GVAR (GOES-8, etc.) calibration parameters are stored in the calibration block. This block consists of 128 words and is the most used location for storing calibration parameters. Although the length of this block will be changed to unrestricted, it will remain at 128 words on the mainframe. These values are used for the entire image file. Sections are defined for AAA and GVAR relating to the parameters for the different bands.

Normally, integers or scaled integers are stored so that moving the data to different platforms, such as Unix or OS/2, is not a problem. However, there are potential problems when storing character data. For example, when accessing files that are not native to the platform, byte flipping may have to take place on the integers but not the character data. The problem arises when deciding how to flip the bytes, since a schema for storing calibration parameters is not defined. Currently, 4-byte words are tested to determine if all the bytes are printable characters. This does not always work for large or scaled integers. Be aware of this situation when developing new calibration modules, if problems seem to be platform-dependent. A software solution to this problem is in progress.
Line prefix

VAS (GOES-5 Sounder), AAA (GOES Mode AAA), and TIRO (AVHRR) calibration parameters are stored in the prefix part of the data line. This is the preferred location for image data, where the calibration parameters can change throughout the image. For AAA data, two different channels may alternate through the image; TIRO calibration has a different set of parameters every five lines.

The prefix can be up to 1000 bytes, and is divided into four sections:

- validity code
- documentation
- calibration
- level

You can define both the documentation and calibration sections for specific data. However, you should use only the calibration section for storing calibration parameters; the documentation section is not guaranteed to move with some of the copy commands.

LW file

The GMS (Japanese satellite) and VAS calibration parameters are stored in LW files. The use of these files is decreasing as the definition of the calibration block becomes less restricted. With the previous limit of 128 words in the calibration block, storing large look-up tables in an LW file was preferable to including it as DATA statements in the code.

For VAS calibration, the line-to-line variability in the calibration coefficients required that the look-up tables be pre-generated for better performance. The file VASTBLS, which accounts for all possible look-up tables, is over 6 megabytes.

GMS data is received from the satellite as 1-byte values. For the IR, each value corresponds to a temperature; for the visible, an albedo. Because this table is fixed for GMS-3, and the calibration block is restricted to 128 words, an LW file is the best place to store the calibration information. On the workstation, changes are underway to allow for calibration blocks of any size. This is needed for GMS-5, since the table is no longer fixed.
Some calibration algorithms, such as VISR (GOES 1-byte data) and WSI (WSI Radar), do not require additional information, or the amount is fixed and relatively small.

The VISR calibration was designed for the original GOES satellites, which transmitted its data as 1-byte values. For the visible (even sensor source numbers), the RAW value is the same as the BRIT, and no conversion is necessary. For the IR (odd sensor source numbers), an option for TEMP (temperature) is available.

WSI radar images need only 16 values to represent the data, base map, labels, etc. These images are handled in the code without an external data structure.

**Performance: computation vs. look-up tables**

Most calibration modules in McIDAS generate look-up tables to convert the stored data to some output physical quantity. This is usually preferred over performing the computation for every input data value. For example, a standard McIDAS image frame is about 300,000 pixels. Since most input data is 8 or 10 bits per value, performing 256 or 1024 computations to create a look-up table is much more efficient than doing 300,000.

The table below shows the functions used extensively in calibration modules.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb</td>
<td>moves bytes</td>
</tr>
<tr>
<td>movc</td>
<td>moves bytes</td>
</tr>
<tr>
<td>movw</td>
<td>moves 4-byte words</td>
</tr>
<tr>
<td>movpix</td>
<td>moves bytes with sampling and offsets</td>
</tr>
<tr>
<td>movblk</td>
<td>moves blocks of bytes with sampling and offsets</td>
</tr>
<tr>
<td>mpixel</td>
<td>in place data expand/pack</td>
</tr>
<tr>
<td>mpixtb</td>
<td>in place data expand/pack with lookup table</td>
</tr>
<tr>
<td>maatb</td>
<td>AAA specific mpixtb</td>
</tr>
<tr>
<td>mavhtb</td>
<td>AVHRR specific mpixtb</td>
</tr>
<tr>
<td>mvastb</td>
<td>VAS specific mpixtb</td>
</tr>
<tr>
<td>mgvatb</td>
<td>GVAR specific mpixtb</td>
</tr>
</tbody>
</table>

The first three (movb, movc, movw) move data from one buffer to another. movpix includes parameters for sampling bytes and moving into a buffer in reverse order. movblk has the same functions as movpix, but also operates on blocks of bytes.
mpixel expands and packs the data values in place. If the source of the data is 1 byte, but the application expects it as 4 bytes, mpixel will do it without the need of an additional buffer. mpixtb adds the feature of passing the data through a look-up table.

The last four functions are special implementations of mpixtb. This was done for performance and memory considerations, such as AAA data sent by the satellite as 10-bit data, but stored on disk as 15 bits. Rather than generating a look-up table of 32,768 values (for 15 bits), a table of 1024 values is made for the 10-bit data. maaatb does the bit shifting to take the raw 15-bit data to 10-bit, and then passes it into the look-up table.
Structure of a calibration module

The sample calibration module, on pages 6-10 through 6-13, illustrates the structure of a calibration module. This calibration will accept input data, from 0 to 255, and return values modified by a sine curve. To use this module with existing 1-byte data, run the following McIDAS command:

```
CA Area STYPE=SIN CTYPE=RAW
```

After compiling KBXSIN.DLM and any appropriate applications, McIDAS applications can be run against the data.

The required three functions are present:

- `kbxini [1-72]`
- `kbxcal [74-129]`
- `kbxopt [132-205]`

An additional subroutine, `maktab`, generates the look-up table.

`kbxini` takes it input, usually from `araopt`, and copies it to a local buffer [53]. It then verifies that the calibration requested is valid [59-60].

`kbxcal` is usually not called directly from the application; it is called when required by `redara`. It takes as input the prefix of the line, the Area directory, the number of values to calibrate, the band (if required), and the buffer containing the data. The calibrated data is returned through this same buffer. `kbxcal` checks to see if the look-up table was generated [118]. If not, a call is made to `maktab` [119]. `mpixt` completes the calibration [125] by taking the data, passing it through the look-up table, and expanding or packing the bytes.

`kbxopt` contains additional operations used by applications to query information about the calibration. The KEYS option [182-187] passes in a frame directory block to the calibration module; the number and list of physical quantities are returned. This option was written for the D and IMGPROBE programs, which list out the stored data value converted to appropriate quantities. Because the information returned by the KEYS option was incomplete, the option, `INFO` [191-202], was added to provide scale factors and units. The input for `INFO` is: band number, sensor source number and calibration type.
An important feature to note is that most calibration modules contain code to handle stretch tables generated by the SU command. By calling `kboxopt` with `BRKP` as the option and the name of the stretch table, the calibration module computes a modified brightness value based on the table. You can usually identify the sections of code where this is done by finding the `CALTYP` variable, which is held in COMMON/B RKPN T. In ADDE, this function is done in the client application instead of the calibration module. This code will be removed from the calibration modules at some future date.
Integrating calibration modules into McIDAS

When the calibration module is coded, you must incorporate it into McIDAS for testing and use by placing the source code in the proper directory and running the appropriate McIDAS tools.

McIDAS-OS2

Copy the calibration module source code (our example is KBXSIN.DLM) into the \mcidas\working directory and run the following three commands from the OS/2 command line.

```
F KBXSIN DL CALLIB KB1SIN
F KBXSIN DL CALLIB KB2SIN
F KBXSIN DL CALLIB KB3SIN
```

These commands invoke the F.CMD script and produce KB1SIN.DLL, KB2SIN.DLL, and KB3SIN.DLL in \mcidas\user\code. The calibration of type SIN is immediately available; it is not necessary to recompile applications.

McIDAS-X

Integrating a user-developed calibration module into McIDAS-X is more complicated because of the lack of dynamic linking. Copy the source KBXSIN.DLM into the mcidas/working directory and run the following procedure from the Unix prompt.

```
fx kbxsin dl
```

This procedure generates, compiles, and files in the user library the three source files kbxsin1.f, kbxsin2.f, and kbxsin3.f, each with unique generated names for all function calls and common blocks. Then, you must generate a new kbprep.for with explicit references to the new calibration type by running the two commands below at the Unix prompt.

```
cal_init ~mcidas/mcidas2.1/src/kb*.dlm kbx*.dlm > kbprep.for
fx kbprep 1i
```

You must then recompile all applications that use calibration before they can access the new type. Note that ~mcidas/mcidas2.1/src is the directory where the core McIDAS-X source can be found for version 2.1. Adjust this accordingly for other versions of McIDAS-X.
Sample program

The sample calibration module, KBX SIN.DLM is provided below.

1: INTEGER FUNCTION KBXINI (CIN, COUT, IOPT)
2: *
3: *$ Name:
4: *$ kbxini - Initialize for sine modified calibration
5: *
6: *$ Interface:
7: *$ integer function
8: *$ kbxini( character*4 cin, character*4 cout, integer iopt(*))
9: *
10: *$ Input:
11: *$ cin - input physical quantity ('TEMP', 'BRIT', 'RAW', etc.)
12: *$ cout - output physical quantity
13: *$ iopt -
14: *$ iopt(1) precision of stored data (1, 2 or 4 bytes)
15: *$ iopt(2) spacing of output data (1, 2 or 4 bytes)
16: *$ iopt(3-5) filled by araopt but should not be used
17: *
18: *$ Input and Output:
19: *$ none
20: *
21: *$ Output:
22: *$ none
23: *
24: *$ Return values:
25: *$ -1 - unit conversion not possible
26: *
27: *
28: *$ Remarks:
29: *$ This calibration module will only accept values from 0 to 255
30: *$ and will return values modified by a sine curve. There is no
31: *$ check for input data out of range.
32: *
33: *$ Categories:
34: *$ calibration
35: *
36: CHARACTER*4 CIN
37: CHARACTER*4 COUT
38: INTEGER IOPT(*)
39: *
40: INCLUDE 'areaparm.inc'
41: *
42: INTEGER JTYPE
43: INTEGER ISOU
44: INTEGER IDES
45: INTEGER JOPT(NUMAREAOPTIONS)
46: C --- Store information needed in other functions
47: C
48: C COMMON/MOSIN/JTYPE,ISOU,IDES,JOPT
49: C --- Copy what araopt sent in
50: C
51: *$ CALL MOVW(NUMAREAOPTIONS,IOPT,JOPT)
52: *
53: *$ JTYPE=0
54: *
55: *$ ISOU=IOPT(1) ! length in bytes of input data
56: *
57: *$ IDES=IOPT(2) ! length in bytes to output data
58: *
59: *$ IF(CIN.EQ.'RAW'.AND.COUT.EQ.'SIN') JTYPE=1
60: *$ IF(CIN.EQ.'RAW'.AND.COUT.EQ.'BRIT') JTYPE=2
61: C

Designing and Implementing Calibration Modules
6-10

McIDAS Developer/Operator Training
October 24, 1995
C--- If not one of the 2 cases above are true, error
62: C
63: IF(JTYPE.EQ.0) GO TO 900
64: KBXINI=0
65: RETURN
66:
67:
68: 900 CONTINUE
69: KBXINI = -1
70: RETURN
71: END
72:
73:
74:
75: INTEGER FUNCTION KBXCAL(PREFIX, IDIR, NVAL, IBAND, IBUF)
76: *$ Name:
77: *$ kbxcal - Calibrate data
78: *$ Interface:
79: *$ integer function
80: *$ kbxcal( integer prefix(*), integer idir(*), integer nval,
81: *$ integer iband, integer ibuf(*))
82: *$
83: *$ Input:
84: *$ prefix - prefix part of image line (not needed)
85: *$ idir - Area directory (not needed)
86: *$ nval - number of values to calibrate
87: *$ iband - band number (not needed)
88: *$
89: *$ Input and Output:
90: *$ ibuf - buffer containing data
91: *$
92: *$ Output:
93: *$ none
94: *$
95: *$ Return values:
96: *$ 0 - success
97: *$ -1 - error (not needed)
98: *$
99: *$ Categories:
100: *$ calibration
101:
102: INTEGER PREFIX(*)
103: INCLUDE 'areaparm.inc'
104:
105: INTEGER JTYPE
106: INTEGER ISOU
107: INTEGER IDES
108: INTEGER JOPT(NUMAREAOPTIONS)
109:
110: INTEGER ITAB(256)
111: COMMON/MOSIN/JTYPE,ISOU,IDES,JOPT
112: DATA IFLAG/0/
113:
114: C--- If the calibration type changes, remake the lookup table
115: C
116: IF( JTYPE .NE. IFLAG) THEN
117: CALL MKTAB(JTYPE, ITAB)
118: IFLAG = JTYPE
119: ENDIF
120: C
121: C--- Pass the data IBUF through the lookup table ITAB
122: C
123: CALL MPIXTB(NVAL, ISOU, IDES, IBUF, ITAB)
124: C
125: C
126: KBXCAL=0
127: RETURN
128: END
129: C
130: C
131: INTEGER FUNCTION KBXOPT(CFUNC, IIN, IOUT)
132: *$ Name:
 baiser - Additional operations

Interface:

integer function kbxo (character*4 cfnc, integer iin(*), integer iout(*))

Input:

cfnc - function ("INFO", "KEYS")
iin - for cfnc "KEYS", iin contains frame directory block
for cfnc "INFO"

iin(1) - band number
iin(2) - sensor source number
iin(3) - calibration type ("GVAR", for example)

Input and Output:

iout - for cfnc "KEYS"
iout(1) - number of physical quantities ("TEMP", etc.)
iout(2-n) - list of physical quantities

iout - for cfnc "INFO"
iout(1) - number of physical quantities ("TEMP", etc.)
iout(2-n) - list of physical quantities, units,
and scale factors

Return values:

0 - success
-1 - invalid function

Categories:

character*4 CFUNC
integer IIN(*)
integer IOUT(*)
include 'areaparm.inc'
integer JTYPE
integer ISOU
integer IDES
integer JOPT(NUMAREAOPCIONS)
common/mgsin/jtype, isou, ides, jopt

C--- KEYS option
C
IF( CFUNC .EQ. 'KEYS') THEN
  iout(1) = 3  ! Number of types
  iout(2) = LIT('RAW ')  ! Physical quantities
  iout(3) = LIT('SIN ')
  iout(4) = LIT('BRIT')
ENDIF
C--- INFO option
C
IF( CFUNC .EQ. 'INFO') THEN
  iout(1) = 3  ! Number of types
  iout(2) = LIT('RAW ')  ! Physical quantities
  iout(3) = LIT('SIN ')
  iout(4) = LIT('BRIT')
  iout(5) = LIT('')  ! Units
  iout(6) = LIT('none')
  iout(7) = LIT('')
  iout(8) = 1  ! Scale factors
  iout(9) = 1000
  iout(10) = 1
ENDIF
RETURN
END
SUBROUTINE MAKTAB(JTYPE, ITAB)

* Name: maktab - Make lookup table for sine modified calibration

* Interface:
  subroutine maktab( integer jtype, integer itab(*) )

* Input:
  jtype - calibration type
  1 - sine
  2 - grayscale

* Input and Output:
  none

* Output:
  itab - lookup table of 256 values

* Remarks:
  This routine makes a lookup table by computing the sine for all possible values from 0 to 255 (the range of the input data). Rather than computing the sine directly on the values 0 to 255, it is initially scaled to 0 to 10 which is approximately 3 sine waves (3 * PI = 10)

* Categories:
  calibration

INTEGER ITAB(*)

REAL SINVAL
REAL X

DO 100 I = 1, 256

   X = I - 1
   X = 10. * (X / 255.) ! Normalize and scale to 3 sine waves
   SINVAL = SIN(X)

C--- Output sine value

IF (JTYPE .EQ. 1) THEN
   ITAB(I) = MINT(SINVAL * 1000.) ! Scale sine by 1000
C--- Output gray scale value
ELSE IF (JTYPE .EQ. 2) THEN
   ITAB(I) = MINT(127. + 128 * SINVAL) ! Scale to 0 to 255
ENDIF

CONTINUE
RETURN
END$
References

*McIDAS Programmer's Manual*
Preliminary issue of Chapters 4, 5 and 6
October 1995

*McIDAS Applications Programming Manual*
Issued February 1988; Revised November 1993

Designing and Implementing Navigation Modules

Presented by
Robert Merrill
McIDAS Applications Programmer

Session 7
McIDAS Developer/Operator Training
October 23-25, 1995
# Table of Contents

Overview ................................................................. 7-1

Navigation module requirements ................................. 7-2

Navigation module design ........................................... 7-3

Navigation algorithm .................................................. 7-4
  Coordinate conventions ......................................... 7-4
  Navigation example - the tangent cone projection ....... 7-5
  Algorithm development ......................................... 7-7
  Additional operations .................................... 7-9

Implementing navigation modules ............................... 7-10
  Coding conventions in McIDAS-X ............................. 7-10
  Sample code description ................................... 7-11

Integrating navigation modules into McIDAS ................. 7-13
  McIDAS-OS2 ...................................................... 7-13
  McIDAS-X ......................................................... 7-13

Sample navigation module ....................................... 7-14

Sample application .................................................. 7-25
Overview

The McIDAS navigation subsystem allows you to extend the list of available map projections for data remapping and display by writing new navigation modules. These user-defined navigation modules can provide exactly the right map projection for your data. You can remap imagery directly into the new projection using the REMAP command. Then a blank image with the new navigation type can serve as a background for the grids and point source data that you want to view in the custom projection.

Navigation modules perform these services:

- convert image coordinates to earth coordinates
- convert earth coordinates to image coordinates
- provide special services associated with a particular navigation type

Because all navigation modules provide these services, a ready-made design framework, including naming and argument conventions, is already in place. Constructing a new navigation module, especially for a map projection, is simpler than it may appear. The difficult part is the navigation algorithms because information about the transform equations is often incomplete. Implementing most map projections will be straightforward if you understand the projection in question, recognize the additional relationships needed, and derive them.

Modules for satellite navigation are more difficult because they involve prediction of the satellite’s position in time, operations in both celestial and terrestrial coordinates, three-dimensional vector operations, and possibly an iterative solver for the inverse (earth to image) transform. The overall design principles and approach are the same, however.

This training session will describe the design and implementation of NVXTANC, a navigation module for the tangent cone projection. Although it is a relatively simple navigation module, it has many features common to more complex modules, such as those used for satellite navigation.
Navigation module requirements

All navigation modules must satisfy the following requirements.

- They must validate their inputs.
- They must never crash, regardless of the inputs.
- They must return either a good transformation or an error status.

All navigation modules, including the ones you will write, must conform exactly to the McIDAS conventions for function naming and arguments, both in position and type. These conventions are shown in the Sample navigation module section of this training session.

The system can only select the right navigation type at run time based on the first word of the navigation block or codicil. The selection is done by a call to nvpref. In MCIDAS-OS2, nvpref links the correct navigation module at run time. In MCIDAS-X, nvpref uses an arithmetic IF statement to run the correct navigation module. Either way, the calling sequences for all navigation services must be identical.

A navigation module should never crash, regardless of its inputs, or return an erroneous solution. If the input cannot be transformed, it must return an error status. The application is then responsible for handling the error properly. To ensure this is done, write your navigation modules so that if an error occurs, the module returns outrageous transform results and an error status. During development, this will help you uncover bugs in the applications that use the modules.
Navigation module design

The interface requirements determine the navigation module design. McIDAS navigation modules are polymorphic. This means they have identical entry points and perform similar operations, regardless of navigation type. The type of transformation (the algorithm) and the particular instance of that transformation (parameters) are hidden in the module. The drawing below shows how a navigation module encapsulates its algorithm and parameters while presenting a familiar polymorphic face to the applications.

**Navigation module components and interface**

![Diagram of navigation module components and interface]

All navigation modules contain the four functions below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvxini</td>
<td>initializes the navigation module</td>
</tr>
<tr>
<td>nvxsae</td>
<td>converts image to earth coordinates; used by the E command</td>
</tr>
<tr>
<td>nvx eas</td>
<td>converts earth to image coordinates; used by the PCE command</td>
</tr>
<tr>
<td>nvxopt</td>
<td>performs other special services besides image-earth transforms</td>
</tr>
</tbody>
</table>

The actual navigation algorithms (equations) are coded in the module. The application initializes the module by passing it parameters, which are stored in the module between service calls. They may be altered only by the application via a call to nvxini to reinitialize the module. nvxini and the three navigation services all return a status.
Navigation algorithm

Published map projections typically contain only equations to transform latitude and longitude onto a Cartesian system on the projection surface. You must understand the projection's properties and often must also derive an inverse transformation and equations to convert between McIDAS earth and image coordinates and earth and projection coordinates, respectively, used by the projection.

Coordinate conventions

The primary navigation algorithms convert image lines and elements (L,E) to latitudes and longitudes (Lat,Lon) using nvxsae or forward navigation. They convert (Lat,Lon) to (L,E) using nvxeas or inverse navigation.

Use the McIDAS conventions below for earth and image coordinates.

Latitudes and Longitudes

As shown in the drawing below, latitudes are in degrees positive north of the equator; the range is -90 to +90, inclusive. Longitudes are in degrees positive west of the Greenwich (prime) meridian; the range is -180 to +180. Because the longitude convention differs from standard cartographic practice, and the treatment of the date line is ambiguous, you should be careful to handle longitude consistently.
**Image coordinates**

Image coordinates are in lines and elements, as shown below, and may contain a fractional component. It is convenient to think of \((L,E)\) as defining a right-handed coordinate system with its origin in the upper-left of the image as displayed on the screen.

Image coordinates are the basic coordinate system for data stored in both AREA files and the TV display. Area and TV coordinates can be converted to or from image coordinates using information in the area directory or frame directory, respectively. Navigation modules all operate directly on image coordinates.

![Image Coordinates Diagram]

**Navigation example - the tangent cone projection**

As an example, we will implement a navigation module for a tangent cone projection (Saucier 1983)\(^1\). All we are given is the following:

- a pair of equations for \(R\) and \(\theta\), which are the polar coordinates on the developed projection surface, in terms of \(\psi\) and \(\lambda\).

- a third equation for map scale \(\sigma\) as a function of \(\psi\)

This is typical. The cartographers who develop map projections speak their own language and use their own symbols and conventions. As a McIDAS navigation module developer, you must bridge the gaps. It is generally easier to work in the variables of the projection and convert to and from McIDAS earth coordinates as a pre- or post-processing step.

As shown in the drawings on the next page, the projection’s earth coordinates are \(\psi\) and \(\lambda\), or colatitude and longitude. Colatitude is measured in radians, beginning from 0 at the pole of the projection, which is the apex of the tangent cone. The longitude is also in radians with 0 at the prime meridian, but is positive to the east.

---

The colatitude at which the cone is tangent to the Earth is $\psi_0$, or the standard colatitude. At this colatitude, distance on the imaginary tangent cone exactly matches distance on Earth. Elsewhere, distances on the cone are larger than on the Earth by a factor $\sigma$, which is a function of $\psi$ only.

This is the mathematical price paid for representing the curved earth on a flat surface. This particular projection is conformal, meaning the scale $\sigma$ at a point is independent of direction. This is not true of all projections.

The projection converts earth coordinates to projection coordinates $R$ and $\theta$ on the developed surface. Think of the projection as being made by removing the tangent cone from the Earth, cutting it along the longitude opposite the standard longitude $\lambda_0$, and flattening it. The radius $R$ of a point on the cone is its distance from the apex. The bearing $\theta$ of the point is the angle between the meridian of the standard longitude $\lambda_0$ and a meridian through the point, with counterclockwise $\theta$ positive.

**Navigation Algorithms**

![Diagram of a cone tangent to a sphere and a cone-cut and flattened map](image)

*Designing and Implementing Navigation Modules*  
McIDAS Developer/Operator Training  
October 24, 1995
Algorithm development

Derived tangent cone navigation transforms use the following symbols.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>Earth radius</td>
</tr>
<tr>
<td>$E$</td>
<td>image element</td>
</tr>
<tr>
<td>$E_0$</td>
<td>image element location of the North Pole</td>
</tr>
<tr>
<td>$L$</td>
<td>image line</td>
</tr>
<tr>
<td>$L_0$</td>
<td>image line location of the North Pole</td>
</tr>
<tr>
<td>$\text{Lat}$</td>
<td>latitude; McIDAS convention</td>
</tr>
<tr>
<td>$\text{Lon}$</td>
<td>longitude; McIDAS convention</td>
</tr>
<tr>
<td>$m$</td>
<td>map scale, in km per pixel at standard latitude</td>
</tr>
<tr>
<td>$R$</td>
<td>radius on the projection surface, in km</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>longitude; projection convention</td>
</tr>
<tr>
<td>$\lambda_0$</td>
<td>standard longitude; projection convention</td>
</tr>
<tr>
<td>$\theta$</td>
<td>bearing from standard longitude (and element axis) on the projection surface</td>
</tr>
<tr>
<td>$\psi$</td>
<td>colatitude</td>
</tr>
<tr>
<td>$\psi_0$</td>
<td>standard colatitude</td>
</tr>
</tbody>
</table>

*Note the limiting cases and singularities*

The equations for $R$ and $\theta$ are shown below.

$$R = a \tan \psi_0 \left[ \frac{\tan \frac{\psi}{2}}{\left( \frac{\tan \frac{\psi_0}{2}}{2} \right)^2} \right] \cos \psi_0$$  \hspace{1cm} (1)

$$\theta = \cos \psi_0 (\lambda - \lambda_0)$$  \hspace{1cm} (2)

Inspection reveals that a divide-by-zero will occur for $\psi_0$ of 0. For $\psi_0$ of $\pi/2$, the leading $\tan \psi_0$ in the expression for $R$ will tend to infinity, leading to another singularity. You must recognize and reject these choices of standard colatitude (North Pole and equator, respectively).

For $\psi_0 > \pi/2$, which is a standard colatitude in the Southern Hemisphere, the leading $\tan \psi_0$ term in the expression for $R$ is negative, leading to negative radii. It is unclear if this would be consistent for a tangent cone with its apex over the South Pole. You would also have to reconsider the sign convention for longitude ($\lambda$) for the Southern Hemisphere. For a fully general navigation module, you should resolve these difficulties to support Southern Hemisphere tangent cones. However, in this training session, we will exclude $\psi_0 > \pi/2$. 
You should also note that the valid range of $\theta$ is not $-\pi < \theta \leq \pi$ but $-\pi \cos \psi_0 < \theta \leq \pi \cos \psi_0$. This means that points in the pie-wedge region where the cone was cut and flattened are not navigable and should be rejected as arguments to inverse navigation. Also note that as $\psi \to \pi$, $R \to \infty$, meaning the South Pole is un navigable in practice and should be excluded. The limits on inputs are summarized below.

**Parameters**

- $0 < \psi_0 < \pi/2$  
  standard colatitude is confined to the Northern Hemisphere
- $-\pi/2 < \lambda_0 \leq \pi/2$  
  standard longitude must be a legal value

**Earth coordinates**

- $0 \leq \psi < \pi$  
  all colatitudes are navigable except the South Pole
- $-\pi < \lambda \leq \pi$  
  all longitudes are navigable

**Image coordinates**

- $R \geq 0$  
  no valid earth location maps to $-R$
- $-2\pi \cos(\psi_0) < \theta \leq 2\pi \cos(\psi_0)$  
  area in the split region of the flattened cone is not navigable; exclude it as an input

**Derive the inverse transform**

The next step is to derive the forward transform $(R, \theta)$ to $(\psi, \lambda)$ from the inverse transform provided by algebraic manipulation of (1) and (2), yielding the two equations below.

$$
\psi = 2 \tan^{-1} \left[ \tan \frac{\psi_0}{2} \left( \frac{R}{\tan \psi_0} \right) \frac{1}{\cos \psi_0} \right]
$$

(3)

and

$$
\lambda = \lambda_0 + \frac{\theta}{\cos \psi_0}
$$

(4)
Extend transforms to McIDAS API units

Now write the equations for the conversions between McIDAS and the projection earth coordinates. They are straightforward, involving a degree-radian conversion, the definition of colatitude and latitude, and the differing convention for positive longitude as shown below.

$$\psi = \frac{\pi}{2} - \frac{\pi}{180} \text{Lat} \quad \lambda = -\frac{\pi}{180} \text{Lon}$$  \hspace{1cm} (5)

and

$$\text{Lat} = 90 - \frac{180}{\pi} \psi \quad \text{Lon} = -\frac{180}{\pi} \lambda$$ \hspace{1cm} (6)

To understand the conversion between (R,θ) and screen coordinates, imagine the flattened cone superimposed on your image coordinate system with the standard longitude parallel to the left edge of the screen. You will want to specify the location of the pole on the image (L₀,E₀) and the map scale m (distance on the cone in km per pixel). Although the convention that the standard longitude λ₀ lies along a constant element value could be relaxed by adding another parameter for rotation, this has not been done. Simple trigonometry yields these conversions between (R,θ) and (L,E).

$$L = L_0 + \frac{R \cos \theta}{m} \quad E = E_0 + \frac{R \sin \theta}{m}$$ \hspace{1cm} (7)

and

$$R = m \sqrt{(L - L_0)^2 + (E - E_0)^2}$$ \hspace{1cm} (8)

$$\theta = \tan^{-1}\left[\frac{(E - E_0)}{(L - L_0)}\right]$$

Additional operations

Optional services can involve the computation of anything about the projection that an application needs. The example here is the computation of the scale factor at any point on the Earth. The expression for σ gives the ratio of the map scale at a given ψ to that at ψ₀, so the actual map scale M(ψ) is just mσ(ψ), where m is km per pixel specified at ψ₀.

$$M(\psi) = m \sigma(\psi) = m \left(\frac{\sin \psi_0}{\sin \psi} \right) \left[\frac{\tan \psi}{\tan \psi_0}\right]^{\cos \psi_0}$$ \hspace{1cm} (9)
Implementing navigation modules

To implement your navigation module, you should know the special coding conventions for McIDAS-X and the architecture of the module.

Coding conventions in McIDAS-X

Navigation modules must conform to the following coding conventions to build properly in MCIDAS-X.

- The name declaration for each function must be uppercase:
  INTEGER FUNCTION NVXINI(...)

- The word COMMON in all common block definitions must be in uppercase, and no more than one space must occur between it and the / at the beginning of the common block name. The use of unnamed (blank) common is generally a risky practice; it is not allowed here.

- The words FUNCTION, COMMON, and any of the interface function names NVXINI, NVXSAE, NVXEAS, and NVXOPT should not occur in comments.

These restrictions are necessary because dynamic linking is not presently supported in MCIDAS-X. To emulate dynamic linking to allow more than one navigation module (slot) to be used by an application at a time (as for REMAP), the scripts that build navigation for MCIDAS-X generate three distinct source modules, each with unique interface function names.

Common blocks are also renamed to avoid collisions between modules. The coding conventions above make it possible for the scripts to recognize those names that need to be modified.
Sample code description

Two sample source files are provided at the end of this document:

- sample navigation module
- sample application

Each is described below. Numbers in bold type refer to the numbered lines in the examples.

Sample navigation module

The sample navigation module contains the four routines `nvxin`, `nvxsae`, `nvxcas`, and `nvoxopt` required for all navigation modules.

When called from an application via `nvprep` to initialize the module (first argument `option` value of 1), `nvxin` validates the navigation type [150]. It then converts the parameters to floating point [154-158], checks their validity, and converts them from McIDAS to projection form [166-212]. If successful, it stores them and some intermediate quantities needed by the navigation transform [199-202] in the common block [118-120] and sets the initialization flag [246]. `nvxin` is also used by applications to select the form of the Earth coordinates, latitude-longitude or Cartesian, by specifying a value of 2 for argument `option` and a value of LL or XYZ for the second argument `param`. Lines 224-235 interpret these inputs and either set or clear the flag LatLon, also stored in the common block.

Forward navigation (`nvxsae`) first verifies that the navigation module is initialized [368-371]. It then converts the incoming image coordinates into projection `R` and `θ` [376-380]. Range checking is then applied [385-388] according to the conditions derived earlier (see the Algorithm development section). The actual navigation transform is very short [393-397]. The earth coordinates are then made to conform to the McIDAS convention [403-405] and, if necessary, converted from latitude-longitude to Cartesian coordinates using McIDAS library routine `allxyz` [408].

Inverse navigation (`nvxcas`) follows a similar pattern. The module state is checked [539-542], incoming earth coordinates are converted to McIDAS latitude and longitude, if required by the current navigation option [548-555], and the range is checked [560-573]. When this is done, the latitude and longitude are converted from McIDAS to projection convention [575-590], the transform is applied [598-599], and image coordinates are computed [609-610].
**nvxopt** first verifies that the module is initialized [720-723]. It then examines the name of the special service request [729 and 749]. At present, only SCAL is supported; other options return an error status [753-754]. As in **nvxsae** and **nvxseas**, the SCAL option in **nvxopt** involves an input range check [734-740], conversion from McIDAS to projection form of Earth coordinates [742], and algorithm evaluation [746].

**Sample application**

The sample application is the source code for the McIDAS command MAKTANC, which creates an area with tangent cone navigation. Areas produced with MAKTANC can be used by the REMAP command. The tangent cone map shown earlier was prepared using MAKTANC.

The sample application shows how a navigation block is prepared and inserted into an area. Most of the code fetches user input from the command line and prepares a consistent set of navigation parameters. Five parameters (line and element of pole, standard latitude, standard longitude, and scale) are required. These can either be entered directly using the POLE, SLAT, SLON, and SSCALE keywords (see the help section of the code sample), or computed from the latitude, longitude, and scale at the center of the area being created. Lines 222-247 create the area directory. Lines 253-258 then fill the block with the navigation type and parameters, and lines 260-262 insert the block into the area. Note that the order and scaling of these parameters exactly matches that in the tangent cone initialization module **nvxini**. See lines 17-22 of the sample navigation module.
Integrating navigation modules into McIDAS

When the navigation module is coded, you must incorporate it into McIDAS for testing and use by placing the source code in the proper directory and running the appropriate McIDAS tools.

McIDAS-OS2

Copy the navigation module source code (our example is NVXTANC.DLM) into the \mcidas\working directory and run the following three commands from the OS/2 command line.

F NVXTANC DL NVLLIB NV1TANC
F NVXTANC DL NVLLIB NV2TANC
F NVXTANC DL NVLLIB NV3TANC

These commands invoke the F.CMD script and produce NV1TANC.DLL, NV2TANC.DLL, and NV3TANC.DLL in \mcidas\user\code. The navigation for TANC codicils is immediately available; it is not necessary to recompile applications.

McIDAS-X

Integrating a user-developed navigation module into McIDAS-X is more complicated because of the lack of dynamic linking. Copy the source NVXTANC.DLM into the mcidas/working directory and run the following procedure from the Unix prompt.

fx nvxtanc dl

This procedure generates, compiles, and files in the user library the three source files nvxtanc1.f, nvxtanc2.f, and nvxtanc3.f, each with unique generated names for all function calls and common blocks. Then, you must generate a new nvprep.for with explicit references to the new navigation type by running the two commands below at the Unix prompt.

nav_init -mcidas/mcidas2.1/src/nv*.dlm nv*.dlm > nvprep.for
fx nvprep li

You must then recompile all applications that use navigation before they can access the new type. Note that ~mcidas/mcidas2.1/src is the directory where the core McIDAS-X source can be found for version 2.1. Adjust this accordingly for other versions of McIDAS-X.
Sample navigation module

The sample navigation module, NVXTANC.DLM is provided below.

0001: C THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED.
0002: C
0003: C *** McIDAS Revision History ***
0004: C *** McIDAS Revision History ***
0005: C
0006: *$ Name:
0007: *$ nvxini - Initialize navigation for tangent cone projection
0008: *$
0009: *$ Interface:
0110: *$ integer function
0111: *$ nvxini(integer option, integer param(*))
0112: *$
0113: *$ Input:
0114: *$ option - 1 to set or change projection parameters
0115: *$ option - 2 set output option
0116: *$ param - For option 1:
0117: *$
0118: *$ param( 1) = 'TANC'
0119: *$
0120: *$ param( 2) = image line of pole*10000
0121: *$
0122: *$ param( 3) = image element of pole*10000
0123: *$
0124: *$ param( 4) = km per pixel *10000
0125: *$
0126: *$ param( 5) = standard latitude *10000
0127: *$
0128: *$ param( 6) = standard longitude *10000
0129: *$
0130: *$ for option 2:
0131: *$
0132: *$ param( 1) = 'LL' or 'XYZ'
0133: *$
0134: *$
0135: *$ Input and Output:
0136: *$
0137: *$
0138: *$
0139: *$
0140: *$
0141: *$
0142: *$
0143: *$
0144: *$
0145: *$
0146: *$
0147: *$
0148: *$
0149: *$
0150: *$
0151: *$
0152: *$
0153: *$
0154: *$
0155: *$
0156: *$
0157: *$
0158: *$
0159: *$
0160: *$
0161: *$

0207: *$
0208: *$
0209: *$
0210: *$
0211: *$
0212: *$
0213: *$
0214: *$
0215: *$
0216: *$
0217: *$
0218: *$
0219: *$
0220: *$
0221: *$
0222: *$
0223: *$
0224: *$
0225: *$
0226: *$
0227: *$
0228: *$
0229: *$
0230: *$
0231: *$
0232: *$
0233: *$
0234: *$
0235: *$
0236: *$
0237: *$
0238: *$
0239: *$
0240: *$
0241: *$
0242: *$
0243: *$
0244: *$
0245: *$
0246: *$
0247: *$
0248: *$
0249: *$
0250: *$
0251: *$
0252: C
0253: C
0254: C
0255: C
0256: C
0257: C
0258: C
0259: C
0260: C

0261: *$
0262: *$
0263: *$
0264: *$
0265: *$
0266: *$
0267: *$
0268: *$
0269: *$
0270: *$
0271: *$
0272: *$
0273: *$
0274: *$
0275: *$
0276: *$
0277: *$
0278: *$
0279: *$
0280: *$
0281: *$
0282: *$
0283: *$
0284: *$
0285: *$
0286: *$
0287: *$
0288: *$
0289: *$
0290: *$
0291: *$
0292: *$
0293: *$
0294: *$
0295: *$
0296: *$
0297: *$
0298: *$
0299: *$
0300: *$
0301: *$
0302: *$
0303: *$
0304: *$
0305: *$
0306: *$
0307: *$
0308: *$
0309: *$
0310: *$
0311: *$
0312: *$
0313: *$
0314: *$
0315: *$
0316: *$
0317: *$
0318: *$
0319: *$
0320: *$
0321: *$
0322: *$
0323: *$
0324: *$
0325: *$
0326: *$
0327: *$
0328: *$
0329: *$
0330: *$
0331: *$
0332: *$
0333: *$
0334: *$
0335: *$
0336: *$
0337: *$
0338: *$
0339: *$
0340: *$
0341: *$
0342: *$
0343: *$
0344: *$
0345: *$
0346: *$
0347: *$
0348: *$
0349: *$
0350: *$
0351: *$
0352: C
0353: C
0354: C
0355: C
0356: C
0357: C
0358: C
0359: C
0360: C
0361: C

Designing and Implementing Navigation Modules

McIDAS Developer/Operator Training

7-14

October 24, 1995
implicit NONE

! Interface variables (formal arguments)
integer option ! initialization option
integer param(*) ! navigation parameters or
character*4 navtyp ! codicil type
character*4 outcoord ! output coordinate type
real lat0 ! standard latitude
character*80 cbuf ! text output buffer

! Common block variables and declaration.
ALL CODE BETWEEN THE '//' SEPARATORS MUST BE
DUPLICATED EXACTLY IN EACH NAVIGATION ROUTINE
(A more maintenance-safe version would use ENTRY points
rather than separate functions for the navigation API
but entry points cannot be processed by 'convdim'.)

! Common block contents: projection parameters
real Lin0 ! image line of pole
real Ele0 ! image element of pole
real Scale ! km per unit image
character*4 Lon0 ! coordinate (pixel)
real Colat0 ! standard colatitude

! Common block contents: pre-computed intermediate values
real Coscl ! cosine(Colat0)
real Tanc1 ! tangent(Colat0)
real Tanc2 ! tangent(Colat0/2)
real Mxtheta ! limit of angle from std.
real Lon ! ion on projection surface

! Common block contents: constants
real D2R ! degrees to radians factor
real Pi
real Badreal ! returned when navigation
character*4 Erad ! Earth radius
logical Init ! initialized flag
logical Latlon ! .TRUE. for lat/lon I/O

COMMON/TANC/ Lin0, Ele0, Scale, Lon0, Colat0,
& Coscl, Tanc1, Tanc2, Mxtheta,
& D2R, Pi, Badreal, Erad, Init, Latlon

End of common block variables and declaration.

// Begin initialization process by setting constants.
Erad = 6370. ! This value of Erad is ok
for "low-precision" navigation
where spherical Earth is adequate (Saucier, p. 32)
Pi = acos(-1.)
D2R = Pi / 180.

Badreal = -1.E10 ! obvious unreasonable value for nav transform result

C

// Process initialization options. Only one, initialize
// navigation parameters, is supported in this demo version,
// but a 'hook' is left for an additional option to set the
// output coordinate to something other than lat/lon

if( option.eq.1 ) then
  call DDEST('nvxinip(tanc) option=1',0)
call movvc(param(1),nvtyp)
if( nvtyp.eq.'TANC') then

C // Unpack tangent cone projection parameters

Lin0 = param(2) / 10000.
Ele0 = param(3) / 10000.
Scale = param(4) / 10000.
lat0 = param(5) / 10000.
Lon0 = param(6) / 10000.

write(cbuf,'(''' nvxinip: lat0, Lon0 ' ',2F12.4)')
lat0, Lon0
call DDEST(cbuf,0)

C // apply range checking

if(Scale.le.0. ) then
  call DDEST('nvxinip(tanc) scale is negative',0)
  Init = .FALSE.
  NVXINI = -3
  return
end if

if(lat0.le.0. .or. lat0.ge.90. ) then
  call DDEST('nvxinip(tanc) std lat out of range',0)
  Init = .FALSE.
  NVXINI = -3
  return
end if

if(Lon0.le.-180. .or. Lon0.gt.180. ) then
  call DDEST('nvxinip(tanc) std lon out of range',0)
  Init = .FALSE.
  NVXINI = -3
  return
end if

C // convert degrees to radians and latitude to colat.

C // Account for McIDAS longitude convention

Lon0 = -Lon0 * D2R
Colat0 = Pi/2. - D2R*lat0

write(cbuf,'(''' nvxinip: Colat0, Lon_0 ' ',2F12.4)')
Colat0, Lon0
call DDEST(cbuf,0)

C // Compute intermediate quantities

Coscl = cos(Colat0)
Tanc1 = tan(Colat0)
Tanc12 = tan(Colat0/2.)
MXtheta = Pi*Coscl

write(cbuf,'(''' nvxinip: Coscl, Tanc1', 2F7.4)')
Coscl, Tanc1

Designing and Implementing Navigation Modules
McIDAS Developer/Operator Training
7-16
October 24, 1995
0206:          call DDEST(cbuf,0)
0207:          write(cbuf,'(" nvxini: Tancl2, Mxtheta ", 2F7.4")')
0208:          ! tanc12, Mxtheta
0209:          call DDEST(cbuf,0)
0210:          Latlon = .TRUE.
0211:          ! option=1 but type not 'TANC'
0212:          else
0213:          call DDEST('nvxini(tanc) parameter type bad',0)
0214:          Init = .FALSE.
0215:          NVXINI = -4
0216:          return
0217:          end if
0218:          else if ( option .eq. 2) then
0219:          call movwc(param(1),outcoord)
0220:          if( outcoord.eq. 'LL') then
0221:          Latlon = .TRUE.
0222:          else if( outcoord.eq. 'XYZ') then
0223:          Latlon = .FALSE.
0224:          else
0225:          call DDEST('opt=2 coord '/outcoord//' unknown',0)
0226:          Init = .FALSE.
0227:          NVXINI = -5
0228:          end if
0229:          else
0230:          call DDEST('nvxini(tanc) unknown option ',option)
0231:          NVXINI = -4
0232:          return
0233:          end if
0234:          NVXINI = 0
0235:          Init = .TRUE.
0236:          return
0237:          end

0238:          *$ Name:
0239:          *$ NVXSAE - Compute earth coordinates from image coordinates
0240:          *$ Interface:
0241:          *$ integer function
0242:          *$ NVXSAE( real lin, real ele, real dummy,
0243:          *$   real el1, real e2, real e3 )
0244:          *$
0245:          *$ Input:
0246:          *$ lin  - image line
0247:          *$ ele  - image element
0248:          *$ dummy - (unused)
0249:          *$
0250:          *$ Input and Output:
0251:          *$ none
0252:          *$
0253:          *$ Output:
0254:          *$ el1 - latitude or x
0255:          *$ e2 - longitude or y
0256:          *$ e3 - height or z
0257:          *$
0258:          *$ Return values:
0259:          *$ 0 - success
0260:          *$ -1 - input data physically valid but not navigable
given the specified projection
-6 - module not initialized

Remarks:
The navigation module must first be initialized with
a call to nvxini(). The output form (lat,lon) or (x,y,z)
depends on the last call to nvxini() with option 2.

Categories:

navigation

INTEGER FUNCTION NVXSAE( lin, ele, dummy, el1, e2, e3 )

implicit NONE

C // Interface variables (formal arguments)

real lin ! image line to navigate
real ele ! image element to navigate
real dummy ! (unused argument)
real el1 ! Earth coordinate 1
real e2 ! Earth coordinate 2
real e3 ! Earth coordinate 3

C // Interface variables (formal arguments)

real lat ! latitude (McIDAS convention)
real lon ! longitude (McIDAS convention)
real hgt ! height
real dx ! zonal displacement from pole

C // Interface variables (formal arguments)

real dy ! meridional displacement from pole
real radius ! distance from pole on projection
real theta ! angle from standard longitude on
projection surface
real colat ! colatitude of navigated point

C Common block variables and declaration.

C ALL CODE BETWEEN THE '//////' SEPARATORS MUST BE
duplicated exactly in each navigation routine

C (A more maintenance-safe version would use ENTRY points
rather than separate functions for the navigation API
but entry points cannot be processed by 'condim.')

C // Common block contents: projection parameters

C // Common block contents: pre-computed intermediate values

C // Common block contents: constants

Designing and Implementing Navigation Modules 7-18

McIDAS Developer/Operator Training
October 24, 1995
logical   Init   ! initialized flag
logical   Lation  ! .TRUE. for lat/lon I/O

COMMON/TANC/, Lin0, Ele0, Scale, Lon0, Colat0,
& Coscl, Tanc1, Tanc12, Mxttheta,
& D2R, Pi, Badreal, Erad, Init, Lation

End of common block variables and declaration.

C     // verify initialized module
C     if(.not.Init) then
C       NVXSAE = -6
C     return
C     end if
C
C     // Compute radius and bearing from pole
C     dx = Scale*(Lin-Lin0)
C     dy = Scale*(ele-Ele0)
C     radius = sqrt(dx*dx+dy*dy)
C     theta = atan2(dy,dx)
C
C     // Range check theta to determine if point is navigable
C     if ( theta.le.-Mxttheta .or. theta.gt.Mxttheta ) then
C       NVXSAE = -1
C     return
C     end if
C
C     // Forward navigation: compute longitude and colatitude
C     // from radius and theta
C     lon = Lon0 + theta/Coscl
C     if(lon.le.-Pi) lon = lon + 2.d0*Pi
C     if(lon.gt. Pi) lon = lon - 2.d0*Pi
C     colat = 2.*atan( Tanc12 * (radius/(Erad*Tanc1))**(1./Coscl))
C
C     // Rescale to McIDAS convention (degrees, West positive).
C     // Apply conversion to Cartesian coordinates if 'XYZ' set
C     // as output form. Set return code for success.
C     lon = -lon/D2R
C     lat = 90. - colat/D2R
C     hgt = 0.
C     if(.not.Lation) then
C       call nllxyz(lat,lon,e1,e2,e3)
C     else
C       e1 = lat
C       e2 = lon
C       e3 = 0.
C     end if
C     NVXSAE = 0
C     return
C     end
C
C
0422:  *$ Name:
0424:  *$  nvxeas - Compute image coordinates from earth coordinates
0425:  *$
0426:  *$ Interface:
0427:  *$  integer function
0428:  *$  nvxeas( real el, real e2, real e3,
0429:  *$     real lin, real ele, real dummy)
0430:  *$
0431:  *$ Input:
0432:  *$  el  - latitude or x
0433:  *$  e2  - longitude or y
0434:  *$  e3  - height or z
0435:  *$
0436:  *$ Input and Output:
0437:  *$  none
0438:  *$
0439:  *$ Output:
0440:  *$  lin  - image line
0441:  *$  ele  - image element
0442:  *$  dummy  - (unused)
0443:  *$
0444:  *$ Return values:
0445:  *$  0  - success
0446:  *$  -1  - input data physically valid but not navigable
0447:  *$  -2  - input data exceed physical limits
0448:  *$  -6  - module not initialized
0449:  *$
0450:  *$ Remarks:
0451:  *$  The navigation module must first be initialized with
0452:  *$  a call to nvxin1(). The input form (lat,lon) or (x,y,z)
0453:  *$  depends on the last call to nvxin1() with option 2.
0454:  *$  Input longitude may be in the range -360 to +360;
0455:  *$  values outside this range will not be denavigated.
0456:  *$  Height (hgt) is ignored.
0457:  *$
0458:  *$ Categories:
0459:  *$  navigation
0460:  *$
0461:  INTEGER FUNCTION NVXEAS( e1, e2, e3, lin, ele, dummy)
0462:  implicit NONE
0463:  C  // Interface variables (formal arguments)
0464:  C
0465:  C
0466:  real e1  ! Earth coordinate 1
0467:  real e2  ! Earth coordinate 2
0468:  real e3  ! Earth coordinate 3
0469:  real lin  ! image line to navigate
0470:  real ele  ! image element to navigate
0471:  real dummy  ! (unused argument)
0472:  C  // Local variables
0473:  C
0474:  real lat  ! latitude (McIDAS convention)
0475:  real lon  ! longitude (McIDAS convention)
0476:  real hgt  ! height
0477:  real in_lon  ! input longitude (radians, ! East positive)
0478:  C
0479:  real colat  ! colatitude
0480:  real radius  ! distance from pole on projection
0481:  real theta  ! angle from standard longitude on
0482:  C
0483:  C
0484:  C
0485:  C
0486:  C
0487:  C  !--------------------------------------------------------------
0488:  C
0489:  C
0490:  C
0491:  C
0492:  C
0493:  C  (A more maintenance-safe version would use ENTRY points

Designing and Implementing Navigation Modules
McIDAS Developer/Operator Training
7-20
October 24, 1995
rather than separate functions for the navigation API
but entry points cannot be processed by 'convdlm.'

// Common block contents: projection parameters

real Lin0 ! image line of pole
real Ele0 ! image element of pole
real Scale ! km per unit image
real Lon0 ! coordinate (pixel)
real Colat0 ! standard longitude
real Mxtheta ! standard colatitude

// Common block contents: pre-computed intermediate values

real Coscl ! cosine(Colat0)
real Tancl ! tangent(Colat0)
real Tancl2 ! tangent(Colat0/2)
real Mxtheta ! limit of angle from std.
real ! lon on projection surface

// Common block contents: constants

real D2R ! degrees to radians factor
real Pi !
real Badreal ! returned when navigation cannot be done
real Erad ! Earth radius
logical Init ! initialized flag
logical Latlon ! .TRUE. for lat/lon I/O

COMMON/TANC/ Lin0, Ele0, Scale, Lon0, Colat0,
& Coscl, Tancl, Tancl2, Mxtheta,
& D2R, Pi, Badreal, Erad, Init, Latlon

End of common block variables and declaration.

// verify that module is initialized

if(.not.init) then
   NVXEAS = -6
   return
endif

// Preprocess input values. If mode is 'XYZ' first convert
// from Cartesian to lat/lon. If mode is 'LL' just transcribe
// from arguments.

if(Latlon) then
   lat = e1
   lon = e2
else
   call nxyzll( e1, e2, e3, lat, lon)
   hgt = 0.
endif

if( lat.lt.-90 .or. lat.gt.90.) then
   NVXEAS = -2
   return
endif

if( lon.le.-360 .or. lon.gt.360.) then
   \n}\n// Designing and Implementing Navigation Modules
\n\n\nMcIDAS Developer/Operator Training
October 24, 1995
NVXEQS = -2
return
end if
if( lat.eq.-90. .or. lat.eq.90. ) then
NVXEQS = -1
return
end if
colat = Pi/2. - D2R*lat
in_lon = -D2R*lon
// map longitude into range -Pi to Pi
if(in_lon.le.-Pi) in_lon = in_lon + 2.*Pi
if(in_lon.gt. Pi) in_lon = in_lon - 2.*Pi

// Now trap South Pole. Though a valid latitude,
// tan(colat/2) -> infinity there so it is not navigable
if ( colat.eq.Pi ) then
NVXEQS = -1
return
end if

// Compute radius and theta of point on projection surface.
// Theta is tricky; you have to compute offset relative
// to standard longitude, force that into -pi to +pi range,
// and THEN scale by cos(Colat0)
radius = Erad * Tancl *( tan(colat/2.)/Tancl2 ) ** Coscl
theta = in_lon-Lon0
if(theta.le.-Pi) theta = theta + 2.*Pi
if(theta.gt. Pi) theta = theta - 2.*Pi

theta = Coscl * theta

// Compute line and element
lin = Lin0 + radius*cos(theta)/Scale
ele = Ele0 + radius*sin(theta)/Scale
dummy = 0.

NVXEQS = 0

return
e
end

*$ Name:
nvxopt - Perform supplemental navigation operations
*$ Interface:
integer function
nvxopt(integer option, real xin(*),
real xout(*) )
*$ Input:
option - 'SCAL' compute projection scale
xin(1) - latitude
*$ Input and Output:
none
*$ Output:
xout(1) - km per pixel at given latitude
*$ Return values:
0638: *$  0 - success
0639: *$  -1 - input latitude physically valid, but projection
0640: *$  undefined or scale infinite there
0641: *$  -2 - input latitude exceeds physical limits
0642: *$  -5 - unrecognized option
0643: *$  -6 - module not initialized
0645: *$ Remarks:
0646: *$ The navigation module must first be initialized by
0647: *$ a call to nvxini(). Latitude is in degrees, north positive,
0648: *$ and must lie between -90. and +90.
0649: *$ Categories:
0650: *$ navigation
0652:  INTEGER FUNCTION NVXOPT( option, xin, xout)
0654:  implicit NONE
0656:  // Interface variables (formal arguments)
0655:  integer option ! special service name (character
0656:      ! stored as integer)
0657:  real xin(*) ! input vector
0658:  real xout(*) ! output vector
0663:  // Local variables
0664:  character*4 copt ! special service (character form)
0665:  real colat ! input colatitude
0669:  /////////////////////////////////////////////////////////////////////
0670:  Common block variables and declaration.
0672: (A more maintenance-safe version would use ENTRY points
0673: rather than separate functions for the navigation API
0674: but entry points cannot be processed by 'convdim'.)
0679:  // Common block contents: projection parameters
0681:  real Lin0 ! image line of pole
0682:  real Ele0 ! image element of pole
0683:  real Scale ! km per unit image
0684:  ! coordinate (pixel)
0685:  real Lon0 ! standard longitude
0686:  real Colat0 ! standard colatitude
0687:  // Common block contents: pre-computed intermediate values
0690:  real Coscl ! cosine(Colat0)
0691:  real Tancl ! tangent(Colat0)
0692:  real Tancl2 ! tangent(Colat0/2)
0693:  real Mxtheta ! limit of angle from std.
0694:  ! ion on projection surface
0695:  !
0696:  // Common block contents: constants
0697:  real D2R ! degrees to radians factor
0698:  real Pi ! returned when navigation
0699:  ! cannot be done
0700:  real Badreal ! initialized flag
0701:  !.TRUE. for lat/lon I/O
0702:  real Erad
0703:  logical Init
0704:  logical Latlon
0705:  COMMON/TANC/ Lin0, Ele0, Scale, Lon0, Colat0,
0706: &  Coscl, Tancl, Tancl2, Mxtheta,
& D2R, Pi, Badreal, Erad, Init, Latlon
0712: C End of common block variables and declaration.
0713: C //-----------------------------------------------------------------------------------/  
0714:  
0715:  
0716: xout(1) = Badreal  
0717:  
0718: C // verify initialized module  
0719:  
0720: if(.not.init) then  
0721:     NVXOPT = -6  
0722:     return  
0723: end if  
0724:  
0725: C // Extract and interpret the option  
0726:  
0727: call mowc(option,copt)  
0728:  
0729: if(copt.eq.'SCAL') then  
0730:  
0731: C // Compute colatitude and make sure it is  
0732: C // physically possible and navigable  
0733:  
0734: if ( xin(1).gt.90. .or. xin(1).lt.-90. ) then  
0735:     NVXOPT = -2  
0736:     return  
0737: else if ( xin(1).eq.90. .or. xin(1).eq.-90. ) then  
0738:     NVXOPT = -1  
0739:     return  
0740: end if  
0741:  
0742: colat = Pi/2. - D2R*xin(1)  
0743:  
0744: C // Now compute actual scale for this colatitude  
0745:  
0746: xout(1) = scale  
0747:     * (sin(Colat0)**2/tan(Colat2)**2)/sin(Colat)  
0748:  
0749: C else if(copt.eq.'????')  
0750: C // Add code for additional options here  
0751:  
0752:  
0753: NVXOPT = -5  
0754:     return  
0755: end if  
0756:  
0757: NVXOPT = 0  
0758:  
0759: return  
0760: end
A sample application, MAKTANC.PGM, which creates an area with attached TANC navigation, is provided below.

```
C THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED.
C *** McIDAS Revision History ***
C *** McIDAS Revision History ***
C ? MAKTANC -- Create test area with tangent cone navigation
C MAKTANC area lat lon <keywords>
C Parameters:
C area area number to create
C lat latitude at center (def compute from POLE=)
C lon longitude at center (def compute from POLE=)
C Keywords:
C POLE lin ele Image location of North Pole
C (def compute from lat,lon)
C SLAT standard latitude (def=lat)
C SLON standard longitude (def=lon)
C SSCALE scale (km per pixel) at standard latitude
C LSSCALE scale (km per pixel) at center latitude
C Remarks:
C The standard latitude must be between 0 and 90, exclusive.
C --------
C subroutine main0
C implicit NONE
C // Parameters
C integer NKEYS, MXANUM, NVWDS, DIRSIGZ
C parameter (NKEYS=5, MXANUM=9999, NVWDS=128)
C parameter (DIRSIGZ=64)
C integer FFIXLEN, CALLLEN, LEVLEN
C parameter (FFIXLEN=0, CALLLEN=0, LEVLEN=0)
C double precision PI, DR, A
C parameter (A=6370.0)
C // Local variables
C character*12 keys(NKEYS) ! keywords
C logical cpoint ! center point specified ?
C integer slat_type ! standard latitude source
C integer slon_type ! standard longitude source
C integer aranum ! area number
C integer nvblk(NVWDS) ! navigation block
C integer adir(DIRSIGZ) ! area directory
C integer i ! loop index
C double precision clat ! center latitude
C double precision clon ! center longitude
C double precision slat ! standard latitude
C double precision slon ! standard longitude
C double precision sscale ! scale at std latitude
C double precision lscale ! scale at center latitude
C double precision ratio ! scale ratio center:std
C double precision lin_0 ! image line of pole
C double precision ele_0 ! image element of pole
C double precision lin_c ! line of center
C double precision ele_c ! element of center
```
double precision psi_0 ! standard colatitude
0062:  double precision psi ! colatitude
0063:  double precision lam_0 ! standard longitude
0064:  double precision lam ! standard longitude
0065:  double precision radius ! radius on projection
0066:  double precision theta ! bearing on projection
0067:
0068:
0069:
0070:
0071:  C // External functions
0072:
0073:  character*12 cfi ! integer to string
0074:  integer luc ! User Common peek
0075:  integer lit ! integer from character*4
0076:  integer mccmkey ! validate keywords
0077:  integer mccmdnum ! # values with keyword
0078:  integer mccmdll ! fetch latitude/longitude
0079:  integer mccmdbl ! fetch double precision
0080:  integer mccmdint ! fetch integer
0081:
0082:  C // Initialization -----------------------------------------------
0083:
0084:  data keys//'P.OLE','SLA.T','SLO.N','SS.CALE','LS.CALE'//
0085:
0086:
0087:  PI = dacos(-1.0d0)
0088:  D2R = PI/180.0d0
0089:
0090:
0091:  C // Validate key words -------------------------------------------
0092:
0093:  if( mccmkey(NKEYS, keys) .lt. 0 ) return
0094:
0095:  if( mccmdnum(keys(4)).gt.0.and.mccmdnum(keys(5)).gt.0) then
0096:      call edest('Please use either SS.CALE or LS.CALE '//
0097:                       'to set map scale',0)
0098:      return
0099:  end if
0100:
0101:
0102:  C // Fetch command line arguments ---------------------------------
0103:
0104:  C // If area already exists, shut down now. If not, create
0105:  C // an area with the specified number
0106:
0107:  if( mccmdint(' ', 1, 'Number of area to create',
0108:      $ 1, 1, MXANUM, aranum).ne.1100) then
0109:      call edest('You must specify area number to create',0)
0110:      return
0111:      end if
0112:
0113:      call readd( aranum, adir )
0114:      if(adir(1).eq.0) then
0115:         call edest('Area '//cfi(aranum)//' already exists',0)
0116:         call edest('Please delete it or use another number',0)
0117:         return
0118:      end if
0119:
0120:  C // Fetch center latitude and longitude, if available
0121:
0122:  if( ( mccmdll(' ', 2, 'Center latitude',
0123:      $ 0.0d0, -90.0d0, 90.0d0, clat).gt.1000 ) .and.
0124:      $ ( mccmdll(' ', 3, 'Center longitude',
0125:      $ 0.0d0, -180.0d0, 180.0d0, clon).gt.1000 ) ) then
0126:     cpoint = .TRUE.
0127:  else
0128:     cpoint = .FALSE.
0129:  end if
0130:
Designing and Implementing Navigation Modules
McIDAS Developer/Operator Training
7-26
October 24, 1995
// Fetch standard latitude and longitude. If center point
// was entered, use it as default. Otherwise, require that
// a value be entered. Determine this by comparing 'cpoint'
// flag and the return codes from mccmddll(). If cpoint is
// not set, actual values (not the defaults) must have been
// fetched for SLAT and SLON

slat_type = mccmddll(keys(2), 1, 'Standard latitude',
clat, 0.d0, 90.0d0, slat)
slon_type = mccmddll(keys(3), 1, 'Standard longitude',
clon, -180.d0, 180.d0, slon)

if( .not.cpoint .and.
(slat_type.ne.1550 .or. slon_type.ne.1550) ) then
  call edest('You must specify either a center point'/
' or SLAT= and SLON=',0)
else
  return
end if
if( slat.le.0.d0 .or. slat.ge.90.d0 ) then
  call edest('SLAT must be BETWEEN 0 and 90',0)
if(cpoint) then
  call edest('If you want a center point outside',0)
if(cpoint .and. mccmddll(keys(5), 1, 1555)
end if
return
end if
psi_0 = PI/2.d0 - D2R*slat

// Fetch scale. A default is available for the absolute
// (standard latitude) scale. If local scale is specified,
if(mccmddll(keys(4), 1, 'Scale (km/pixel at std lat)'),
1.d0, 0.001d0, 1000.d0, sscale).lt.0) return
if(cpoint .and. mccmddll(keys(5), 1, 1568)
end if
'Scale at center latitude', 1.d0, 0.001d0, 1000.d0,
lscale).eq.1200) then

// Compute sscale from lscale, using standard latitude
psi = PI/2.d0 - D2R*clat
ratio = (tan(psi/2.d0)/tan(psi_0/2.d0))**cos(psi_0) *
sin(psi_0) / sin(psi)
sscale = lscale*ratio

// If a center point was not specified, fetch the location
// of the pole. Otherwise, compute it
if( .not.cpoint ) then
  if( mccmddll(keys(1), 1, 'Image line of pole',0.d0,
1.d0, 0.d0, lin_0, l.t.0 ) return
  if( mccmddll(keys(1), 2, 'Image element of pole',0.d0,
1.d0, 0.d0, ele_0, l.t.0 ) return
  else
    psi = PI/2.d0 - D2R*clat
    lam = - D2R*clon
    lam_0 = - D2R*slon
    radius = A * tan(psi_0) *
(tan(psi/2.d0)/tan(psi_0/2.d0))**cos(psi_0)
    theta = cos(psi_0)*(lam-lam_0) -
    lin_c = dble(luc(11)) / 2.d0

McIDAS Developer/Operator Training
October 24, 1995
Designing and Implementing Navigation Modules
7-27
0205:     ele_c = dble(luc(12)) / 2.0
0206:     lin_0 = lin_c - radius*cos(theta)/sscale
0209:     ele_0 = ele_c - radius*sin(theta)/sscale
0210:
0211:     end if
0212:
0213:
0214:     // Now make the area  -----------------------------
0215:
0216:     // Fill in the area directory and create the area
0217:
0218:     do i=1,DIRSIZ
0219:         adir(1) = 0
0220:     end do
0221:
0222:     adir( 1) = 0     ! existence flag
0223:     adir( 2) = 4     ! format (interleaved)
0224:     adir( 3) = 1     ! satellite ID (test area)
0225:     call getday(adir(4)) ! today’s date
0226:     adir( 5) = 0     ! hour 0
0227:     adir( 6) = 1     ! image line upper left
0228:     adir( 7) = 1     ! image element upper left
0229:     adir( 9) = luc(11) ! # lines (TV size)
0230:     adir(10) = luc(12) ! # elements (TV size)
0231:     adir(11) = 1     ! bytes per element
0232:     adir(12) = 1     ! line resolution
0233:     adir(13) = 1     ! element resolution
0234:     adir(14) = 1     ! number of bands
0235:     adir(15) = 0     ! y-z prefix (0 for image)
0236:     adir(16) = luc(1) ! project number logged on
0237:     call getday(adir(17)) ! creation date
0238:     call gettim(adir(18)) ! creation time
0239:     adir(19) = 1     ! set band 1 (least bit)
0240:     adir(34) = 4*DIRSIZ + 4*NVWDS ! nav block end (bytes)
0241:     adir(35) = 4*DIRSIZ ! nav block start (bytes)
0242:     adir(36) = 0     ! validity code
0243:     adir(49) = PFXLEN ! prefix length, bytes
0244:     adir(50) = CALLEN ! prefix cal length, bytes
0245:     adir(51) = LEVLEN ! prefix lev length, bytes
0246:     adir(52) = lit(’VISR’) ! sensor type
0247:     adir(53) = lit(’BRIT’) ! calibration type
0248:
0249:     call makara (aranum, adir )
0250:
0250:     call makara (aranum, adir )
0251:     // Create and insert the navigation block
0252:
0253:     nvblk( 1) = lit(’TANC’) ! Tangent Cone nav type
0254:     nvblk( 2) = nint(10000*lin_0) ! image line of pole
0255:     nvblk( 3) = nint(10000*ele_0) ! image element of pole
0256:     nvblk( 4) = nint(10000*sscale) ! scale km/pixel at slat
0257:     nvblk( 5) = nint(10000*slat) ! standard latitude
0258:     nvblk( 6) = nint(10000*slon) ! standard longitude
0259:
0260:     call araput(aranum,4*DIRSIZ,4*NVWDS,nvblk)
0261:     call stamp(aranum)
0262:     call cleara(aranum)
0263:
0264:
0265:     call sdest(’MAKTANC done!’,0)
0266:
0267:     return
0268:     end
Developing Local Decoders in McIDAS-XCD

Presented by
John Pyeatt
MUG Program Manager

Session 8
McIDAS Developer/Operator Training
October 23-25, 1995
# Table of Contents

Overview .............................................. 8-1
Terminology ......................................... 8-1
Ingestors currently supported in McIDAS-XCD ...... 8-2
Decoders currently supported in McIDAS-XCD ...... 8-2
Decoders currently under development .......... 8-3

What does a decoder typically do? ................. 8-4

Questions to ask before writing a decoder ........ 8-7

Other McIDAS-XCD subsystems .................. 8-18
  Gross error checking system ..................... 8-18
  Configuration file system ....................... 8-19
  Decoder status display ......................... 8-20
  Station monitoring .............................. 8-21
  DECTEST ........................................ 8-21

Compiling and linking ............................. 8-22

Integrating a local decoder in McIDAS-XCD .... 8-23
  Building the decoder and putting the data in MD files ........ 8-23
  Automating the decoder process ................ 8-27

Developing local decoders for your site ........ 8-29

Useful functions in McIDAS-XCD ................. 8-30

Sample decoder ................................... 8-33
Overview

This training session will provide McIDAS software developers with useful information for writing and maintaining locally developed decoders in conjunction with the McIDAS-XCD software package.

The Nested Grid Model's trajectory forecast reports are used as examples for writing data decoders. You will find a sample decoder for this data type at the end of this section. Values placed in brackets '[ ]' throughout this section reference specific lines of source code provided in the sample decoder.

Terminology

The following terms are used throughout this section.

- **data block**: text data containing a WMO header and a data section
- **data monitor**: McIDAS command that periodically tests to see if new data has arrived that may be important for a specific decoder
- **data section**: portion of text containing independent observations
- **DDS**: Domestic Data Service
- **decoder**: software that parses data from one format into a common format for use by another process such as a plotter or lister, or software that further manipulates data
- **HRS**: High Resolution data Service
- **IDS**: International Data Service
- **ingestor**: process that listens to data received by a communications port and reformats the information for further processing
- **NGM**: National Meteorological Center Nested Grid Model
- **observation**: one complete, independent report about the state of a given group of fields in a specific time or time range
**parsing**  decomposition of an observation into its most elementary parts

**PPS**  Public Products Service

**STARTXCD**  mother task to the entire McIDAS-XCD ingestor/decoder package

**status display**  X Windows application that informs an operator of the current state of the McIDAS-XCD decoder/ingestor system

**token**  smallest entity to which an observation may be parsed

**WMO header**  WMO formatted string containing a product code, product number, originating station and day/time stamp; for example: FOUS14 KWBC 231200

---

### Ingestors currently supported in McIDAS-XCD

- National Weather Service Family of Services: DDS, IDS, PPS, HRS
- National Weather Service AFOS
- AFGWC (Tinker AFB)

---

### Decoders currently supported in McIDAS-XCD

- Surface Hourly SAO and METAR
- RAOB (TEMP): TTAA, TTBB, TTDD, PPAA and PPBB
- Ship and Buoy
- FOUS14
- SYNOP
- AIREP/PIREP
- POES Navigation
- Severe Weather Watch
- GRIB (for HRS circuit)
Decoders currently under development

- Severe Weather by County Federal Information Processing Standards (FIPS)
  - Flood Watch
  - Severe Thunderstorm Watch
  - Tornado Watch
  - Flood Warning
  - Severe Thunderstorm Warning
  - Tornado Warning
- New SAO/METAR Format
- New Aerodrome Forecast (TAF) format
- Expanded data support for GRIB format
What does a decoder typically do?

A decoder goes through approximately seven steps when processing a data type. These steps may include the following:

1. isolating the observation
2. extracting the observation's location
3. extracting the observation time
4. parsing the observation into small enough components so that the information can be interpreted by the software
5. decoding the parsed reports
6. passing the decoded information to the appropriate subsystem for availability to the user
7. writing information to the status display

Each step is described below.

Isolating an observation

When writing decoders that process text data, one of the most difficult tasks is to isolate an individual observation from the data block. Many data types contain observations from several locations in one data block. Determining where one observation ends and the next begins can be a frustrating task. Ideally, a starting character or termination character separates observations.

A starting character is usually the record separator (Hex 1E). If you scan through a data block and encounter a hex 1E, you can be quite certain the characters that follow are a new observation.

If no record separator exists at the beginning of an observation, look for a character indicating the end of the observation. This character may vary depending on the data type. SAO or METAR reports will most likely have an equals sign (=) at the end. Terminal forecasts may be followed by two periods (..).
Be careful. Many overseas reports don't have separation characters and you cannot ignore this problem. If some type of contingent plan is not in the code, you may find erroneous values in your final dataset. For example, several times during its development for the McIDAS-XCD package, the SYNOP decoder produced reports of 400 inches of snow in parts of Iraq. This occurred because the decoder was not prepared for observations coming in without record separators or termination characters.

Extracting the observation's location

Once you isolate an observation, you must determine its origin, such as station ID for METAR reports or station block number for SYNOP reports. This is simple if certain flags are present in the report to help you isolate the section of the data containing the station origin. It may get more complicated with reports that are more free-format.

When you have the origin, you can scan a list of known stations looking for latitude, longitude and elevation information. For some data types, you must determine the latitude and longitude of the observation directly from the report. This is straightforward for reports such as DRIBUs or AIREPs, but can be difficult for a PIREP, which contains no definitive reporting format (or worse, multiple reporting formats).

Once you find the station ID, the decoder can determine if it should continue processing. If you don't have enough meta-data about the station, (latitude, longitude, state, etc.) there is no need to continue processing. In this case, you should file this station in the new stations list with the ID monitoring system [A565] and move on to the next report.

Extracting the observation time

Extracting the observation time is an easy task. The format is rigid, with simple checks to perform [A434-A464]. The only difficulty is when you must use the time report to validate the date of the observation. For example, if a real-time decoder is processing DRIBU reports at 1845 UTC today and an observation comes in with a reporting time of 2130 UTC, your decoder must be robust enough to recognize that the report is really from yesterday. You can use the McIDAS-XCD function mcfittim to simplify this logic.

The observation time is often used to determine where in a data structure an observation should be filed. For example, in MD files, the time determines the appropriate row number for filing data.
Parsing the observation

Once the location and time are known, the decoder breaks the report into individual tokens based on the reporting format. The method of parsing varies among data types. Typically, the three formats are: tabular, formatted and plain text. These will be discussed later in this session.

Decoding the parsed reports

Decoding occurs once the raw text is parsed. This is where all the real work takes place. It is a simple process for tabular data [A1099-A1167]. The only time tabular reports are problematic is when a signal goes bad. Problems with formatted or plain text reports are usually due to human error, such as typos, incorrectly placed data, or meteorologically incorrect values. A decoder must be smart enough not to abort, and to recover successfully from reporting errors. The gross error checking system, described later, is the final place to resolve problems in reports [A705-A709].

Passing the decoded information to the appropriate subsystem

After the information in an observation is fully interpreted by the decoder, it is usually stored in some type of disk file format that users can access. In McIDAS, the most common storage format is MD or grid files. After decoding the report, the software must perform a transformation on the data to put it in a format that is acceptable for the data file structure being used. This transformation may include converting units of the report, changing values from floating point to scaled integers, or converting character strings into string literals [A671-A716].

Writing information to the status display

The final step performed by a decoder is to update the status display [A792-A800]. This display provides the mechanism for operations to monitor the system. Information of interest to the operations staff might include: the time data was last processed by this decoder; the last MD file, row and column written to; the last grid file and grid number written to; or other information unique to this data type.
Questions to ask before writing a decoder

Before you write your decoders, answer the questions below.

**What are the characteristics of the data?**
1. Is the format of the data to be decoded binary or text?
2. If text, is its format tabular, formatted or plain?
3. Does the information come in continually or sporadically?

**What will users do with the data when it is decoded?**
4. Will the decoded information be used by one or many users?
5. Will the user want to plot/contour the data? List it? Notify?
6. Do you have enough resources?

Each of these questions is discussed below. Knowing the characteristics and the intended use of the data will help you determine the portions of the McIDAS-XCD package you can use to process the raw data.

The flow chart on the next page will point you towards some of the modules in the McIDAS-XCD library that you will need.
McIDAS-XCD library modules for acquiring and decoding raw data

What is the format of the raw data?

Binary
- m0grabyt
- mcgbytes
- mcsbytes

Text

Formatted Text
- m0parobs

Tabular
- mceextrln

What is the text format type?

Plain Language
- m0dcsplt
- m0grbval

As needed
- mcodbin
- mcgetidx

Will processing be automatic or as needed?

Automatic
- m0nxtidx
Is the format of the data to be decoded binary or text?

**Binary**

Binary data is the easiest format to decode. A binary format has very rigid standards, so there is little chance for human error. If something is wrong with the signal, it is usually not subtle. The most important concern when writing decoders for binary data is to fireproof the decoder from a bad signal. Fireproofing includes protection against array overwrites and obvious invalid meteorological values.

If the data is a binary stream received through a communications port, you must activate the INGEBIN ingester for the signal so McIDAS-XCD can decode it. To activate a binary data ingester in McIDAS-XCD you must perform the three steps below.

1. Create a configuration file containing the information specific for the circuit. This information includes the baud rate, the communications port used, and the destination file in which to store the data. If you use the configuration file for the HRS circuit (HRS.CFG) as a template, you will modify the values for IBAUD, OBAUD, PORT and SPOOL.

2. Add the ingester to the list of ingestors that McIDAS-XCD monitors. From the McIDAS command window, enter the command below, replacing `name` with the circuit name and `config` with the configuration file containing the communications information.

   ```
   CIRCUIT ADD name CONFIG=config INGESTOR=INGEBIN
   ```

3. Activate the ingester. From the McIDAS command window, enter the command below.

   ```
   CIRCUIT SET name ACTIVE
   ```

   Every 30 seconds, STARTXCD checks to see if processes were activated, inactivated or aborted. Since the new circuit was just activated, STARTXCD will start up the ingester for you and the ingester will file the data into the spool file designated in the configuration file.

Once the data is placed in the spool file by INGEBIN, you can access it with the McIDAS-XCD function `m0grabyn`. This is a very useful interface for processing real-time binary data.

If the signal contains packed data, use the McIDAS-XCD functions `Mcupackbit` and `mcgbytes` to extract packed values.
Text

If the data is an ASCII stream, decoding gets more complicated. The interfaces for text data in McIDAS-XCD expect the data to begin with a WMO header followed by \( n \) lines of text.

The form for the WMO header is: \( ppcc## \ orig \ ddhhmm \)

where:

- \( pp \) product code
- \( cc \) country code
- \( ## \) numeric code
- \( orig \) station origin
- \( dd \) day of the month
- \( hhmm \) time stamp

For example: FOUS14 KWBC 241725

For this training session, assume the data is already filed in the appropriate raw format.
If text, is it tabular, formatted or plain?

Tabular text

If the text is a computer generated table, such as FOUS14, it is relatively easy to decode. Use the McIDAS-XCD function mcextrln to extract information [A1103-A1104], as shown below.

```
1 2 3
column number: 12345678901234567890123456789012345678
sample string = '68 79 SNOW 45.3'
character*12 string
double precision fltval
integer intval

ok = mcextrln(line,1,2,string,strsta,intval,intsta,fltval,fltsta)
intval returns 68
fltval returns 68.0
string returns '68'
all statuses return success

ok = mcextrln(line,33,36,string,strsta,intval,intsta,fltval,fltsta)
string returns 'SNOW'
strsta returns success
intsta and fltsta return failure

ok = mcextrln(line,32,35,string,strsta,intval,intsta,fltval,fltsta)
fltval returns 45.3
string returns '45.3'
intsta returns failure
```

Below is an example of tabular text.

```
FOUS1-KWBC 181200
TRAJECTORY PCST

181200Z 181800Z 190000Z 190600Z 191200Z
LATLONPP LATLONPP LATLONPP LATLONPP LATLONPP
BIL 446169672 445248642 446126631 449107659 449107659
    2.7 -22.3 -9
850 446129813 465108826 462093842 456086841 467086841
    10.0 -4.0
SFC 466128819 451084422 462092848 456085846 456086846
    8.8 -3.0
RAP 700 443148572 445115569 447083640 445054671 445054671
    6.6 -19.7 -10
850 476112810 473083384 467060846 455040855 455040855
    11.2 -3.1
SFC 478104877 475077895 468059705 456038910 456038910
    902 12.3 .2
BIS 700 483169547 480136575 480098605 477053650 477053650
    2.3 -27.4 -14
850 495141739 494109777 493076795 483039623 483039623
    10.4 -5.1
SFC 490122865 492090900 491065914 482034930 482034930
    944 14.9 1.1
```
**Formatted text**

If the text is formatted, such as SYNOP or RAOB reports, the McIDAS-XCD function `m0parobs` will parse the observation into tokens that can then be broken down into meteorological data. The source line contains the following:

```plaintext
'72645 11/// AUTO 10244 20222'
```

```plaintext
parameter (maxgrp = 100)
character*8 creprt(maxgrp)
integer report(maxgrp)
ok = m0parobs(msg,28,maxgrp,report,creprt,ngroup)
```

`ngroup returns 5`

<table>
<thead>
<tr>
<th>Group #</th>
<th>creprt</th>
<th>report</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'72645'</td>
<td>72645</td>
</tr>
<tr>
<td>2</td>
<td>'11///'</td>
<td>11000</td>
</tr>
<tr>
<td>3</td>
<td>'AUTO'</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>'10244'</td>
<td>10244</td>
</tr>
<tr>
<td>5</td>
<td>'20222'</td>
<td>20222</td>
</tr>
</tbody>
</table>

When `m0parobs` is finished, you can begin extracting parameters. For example, if you determine that the fourth group is the temperature field in Celsius, the extracting code will be:

```plaintext
if (creprt(4)(3:5) ne. '///') temp = float(mod(report(4),1000)) / 10.0
```

Below is an example of formatted text.

```
SXUS23 WKBC 182200 RRA 95230 2244
CMAN 18224
BLIA2 46/// 2307 10114 40171 92230 333 91207 555 11006 22006=
POTA2 46/// 2308 10123 30150 40164 92230 333 91221 555 11008 22008=
```

**Plain text**

Plain text, such as SAO reports, is the most difficult data to decode. Plain text reports do not have a rigid format or data indication flags. These decoders are the most complicated due to the nature of the format. The McIDAS-XCD functions `m0desplit` and `m0grbval` can help. `m0desplit` tokenizes a string into its basic components and `m0grbval` gives you the values in the tokenized string [A408-A447].
Example:

'MSN SA 1155 CLR 10 207/75/65'

include 'xcd.inc'
integer terms(2)
data terms/59,61/  ! termination characters '=' and ';
numgrp = 0
ok = m0dcsp1t(80,numgrp,2,terms)

numgrp returns 15

<table>
<thead>
<tr>
<th>Group #</th>
<th>Value</th>
<th>Type</th>
<th>Numch</th>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MSN</td>
<td>ACHAR</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>blank</td>
<td>ABLANK</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>SA</td>
<td>ACHAR</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>blank</td>
<td>ABLANK</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>1155</td>
<td>ADIGIT</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>blank</td>
<td>ABLANK</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>CLR</td>
<td>ACHAR</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>blank</td>
<td>ABLANK</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>ADIGIT</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>blank</td>
<td>ABLANK</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>11</td>
<td>207</td>
<td>ADIGIT</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>/</td>
<td>ASLASH</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>75</td>
<td>ADIGIT</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>14</td>
<td>/</td>
<td>ASLASH</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>15</td>
<td>65</td>
<td>ADIGIT</td>
<td>2</td>
<td>27</td>
</tr>
</tbody>
</table>

Other Type values include the following.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>terminator character</td>
</tr>
<tr>
<td>APUNC</td>
<td>;:</td>
</tr>
<tr>
<td>PLSMNS</td>
<td>+ -</td>
</tr>
<tr>
<td>ARECS</td>
<td>record separator (0x1e)</td>
</tr>
<tr>
<td>AXCLAM</td>
<td>!</td>
</tr>
<tr>
<td>AAMPER</td>
<td>@</td>
</tr>
<tr>
<td>ALOGNT</td>
<td>^</td>
</tr>
<tr>
<td>AOTHER</td>
<td>any other character</td>
</tr>
</tbody>
</table>

Below is an example of plain text.

SAUS80 KWBC 182200
ILM SA 2150 50 SCT 250 -SCT 7 112/89/74/1507/986/ MDT CU W
LGA SA 2150 250 SCT 13 132/89/60/0512/992/ CU N
LND SA 2150 300 -BKN 70 147/78/18/3212/015/ CU N-NE
MFE SA 2150 40 SCT E250 BKN 10 85/75/0911/986/TCU N-E
MIA SA 2150 30 SCT M50 BKN 250 OVC 7 120/90/75/2210/988/ CB N MOVG S
MSN SA 2150 28 SCT M36 BKN 7 144/85/73/1208/997
OFK SA 2150 CLR 15 073/96/72/1712/980/ CB DSNT W-NW MOVG NE
Retrieving Text Data

Data Block Index
1 - circuit source
2 - number of bytes in data block
3 - starting location in *.XCD file
4 - time stamp (hhmmss)
5 - WMO header ex. 'FOUS'
6 - WMO product number
7 - WMO station origin ex. 'KWBC'
8 - AFOS/AMIPS product code
9 - AFOS/AMIPS station origin
10 - AFOS/FIL station origin
16 - FAA catalog number

**FO95300.IDX**

<table>
<thead>
<tr>
<th>224</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>160</td>
</tr>
<tr>
<td>176</td>
</tr>
<tr>
<td>192</td>
</tr>
<tr>
<td>208</td>
</tr>
<tr>
<td>224</td>
</tr>
</tbody>
</table>
If data is continuous, such as surface hourlies, RAOBs, and MDRs, you should consider processing it in an automatic mode. This method requires no user intervention to ensure data is decoded.

To process data automatically, you must create data monitors similar those distributed in McIDAS-XCD. Using m0nxtidx [C344-C347], these data monitors scan a predefined set of index files, checking to see if they have been updated by the ingestors. If they have, the data monitor wakes up, loads the new index block, and, if it is one the data monitor is interested in, calls an appropriate decoder [C433-C439]. One data monitor can run many decoders. If you plan to have numerous local decoders, SSEC recommends embedding them in your own data monitors and activating them with the DECINFO command.

If your data is sporadic, it may be enough to run the decoder on an as-needed basis. For example, to plot projected hurricane storm tracks, which are not high volume reports, you can access the raw text directly each time a plot is requested instead of creating a special filing format just for hurricane tracks.

The interface for accessing sporadic data is very different from automatic processing. In the automatic mode, a data monitor, using m0nxtidx, scans to see if new data has arrived. In the as-needed mode, the application, using mcgetidx [B39], starts with the most recent data and works backward in time. Each call to this function grabs the next oldest index block for the data you request. See the diagram on the adjacent page. Once you retrieve the new index block, you can load the text and call your decoder.

The structure of the index block returned by mcgetidx and m0nxtidx is as follows:

<table>
<thead>
<tr>
<th>Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>circuit source</td>
</tr>
<tr>
<td>2</td>
<td>number of bytes in the data block</td>
</tr>
<tr>
<td>3</td>
<td>starting location in the .XCD file</td>
</tr>
<tr>
<td>4</td>
<td>time stamp, hhmmss</td>
</tr>
<tr>
<td>5</td>
<td>WMO header, for example FOUS</td>
</tr>
<tr>
<td>6</td>
<td>WMO product number</td>
</tr>
<tr>
<td>7</td>
<td>WMO station origin, for example KWBC</td>
</tr>
<tr>
<td>8</td>
<td>AFOS/AWIPS product code, when available</td>
</tr>
<tr>
<td>9</td>
<td>AFOS/AWIPS station origin, when available</td>
</tr>
<tr>
<td>10</td>
<td>AFOS/PIL station origin, when available</td>
</tr>
<tr>
<td>16</td>
<td>FAA catalog number, when available</td>
</tr>
</tbody>
</table>

Once you have the appropriate index block, you load the data into a character array buffer with mclltdatb and continue decoding.

Note: Binary data should be processed in the automatic mode.
Will the decoded information be used by one or many users?

If only one user will access the data, you may not need to create a data monitor for specific requests. Processing the data in an as-needed mode should be sufficient. If multiple users will access the data, consider the automatic mode to prevent redundant processing and wasting of resources.

What will the user do with the data?

If users want to plot or contour data, the data should be decoded and placed in McIDAS-supported data formats such as MD [A783] or grid files. Then users can display or list it with many of the core McIDAS commands.

When you build a decoder, do not tie that decoder to a specific filing format. The decoder should extract the information from the data stream into a known memory structure. Then you can write an ancillary routine that converts the memory structure to the format for the file system. Thus, if you decide to file a certain type of data in a different file structure, the only new software you have to write is a routine converting the memory structure to the new file format.

If timeliness of text retrieval is important for observational reports, you may want to decode the observation and file it in the McIDAS-XCD Rapid Access (RA) text format. This file structure allows the user to formulate the request for individual stations or groups of stations and retrieve the data quickly. Putting raw text into this format is a two-step process.

1. Register the filing format with the BILDTEXT command, which sets up the pointer files for observational data. Enter the type of data you will store, the length of time to store it, and the format of the station ID. BILDTEXT initializes the file based on this input.

2. In the decoder, open the RA file using mctxttop [A367]. Fill in two data blocks with pertinent information about the observation, and call mctxtwrt [A740-A773] to write the data to an RA file. An example is provided in the program at the end of this section.

Once the data is in an RA file, you can use the command OBLIST to access the data quickly. Typically, you will build a macro on top of OBLIST to access the data type. The McIDAS-XCD commands SAO, RAOB, FT and TAF are examples of macros that call OBLIST.

Sometimes, the only thing a user needs to know from a data type is if a particular event took place. In this case, nothing needs to be filed, although an alert should be sent to the user. As an example, operations should be notified when an administrative message is received.
A tremendous amount of data passes through a McIDAS-XCD server workstation. In a typical ingest set up with DDS, PPS and IDS, you can expect about 100 Mb per day before the data is put in a special format such as MD or grid files. When you add McIDAS data files to this list, you quickly expand the need for disk storage for these machines.

To calculate the amount of disk space (in bytes) needed for an MD file, use this formula:

\[(NR \times NC \times NDKEYS + NC \times NCKEYS + NR \times NRKEYS) \times 4\]

where:
- \(NR\) number of rows in the MD file
- \(NC\) number of columns in the MD file
- \(NDKEYS\) number of keys in the MD file’s data section
- \(NCKEYS\) number of keys in the MD file’s column header
- \(NRKEYS\) number of keys in the MD file’s row header

To calculate the amount of disk space needed for grids, use this formula:

\[((NR \times NC \times 4) + 256) \times NG\]

where:
- \(NR\) number of rows in the grid
- \(NC\) number of columns in the grid
- \(NG\) number of grids in the file

The HRS circuit transmits another 70 to 90 Mb per day. Once you decode and file the HRS signal, the amount of needed disk space grows rapidly, depending on which grid formats are decoded. Some grids sent by NCEP, for example, may be up to 160 Kb per grid, and approximately 500 grids are sent in this format.

SSEC recommends that your McIDAS-XCD workstation has a minimum of 2 Gb of hard disk for the typical ingest/decode configuration.
Other McIDAS-XCD subsystems

Many of the processes done in decoders are the same, regardless of the data type. This section describes some of the common subsystems in McIDAS-XCD that you can integrate into your site's local decoder system.

Gross error checking system

McIDAS-XCD performs gross error checks, discarding values that are not meteorologically possible. The McIDAS-XCD functions megrssel and m0grserr perform this task [A708-A709]. Below is a table of the current parameters and thresholds used for gross error checking.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Name</th>
<th>Units</th>
<th>Level</th>
<th>Min. value</th>
<th>Max. value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure</td>
<td>P</td>
<td>mb</td>
<td>All</td>
<td>0</td>
<td>1100</td>
</tr>
<tr>
<td>Sea level pressure</td>
<td>SLP</td>
<td>mb</td>
<td>All</td>
<td>800</td>
<td>1100</td>
</tr>
<tr>
<td>Precipitation</td>
<td>PCP</td>
<td>M</td>
<td>All</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Snow depth</td>
<td>SNOW</td>
<td>M</td>
<td>All</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Sea surface temp</td>
<td>SST</td>
<td>K</td>
<td>All</td>
<td>265</td>
<td>315</td>
</tr>
<tr>
<td>Temperature</td>
<td>T</td>
<td>K</td>
<td>All</td>
<td>188</td>
<td>340</td>
</tr>
<tr>
<td>Dew point</td>
<td>TD</td>
<td>K</td>
<td>All</td>
<td>188</td>
<td>310</td>
</tr>
<tr>
<td>Wind direction</td>
<td>DIR</td>
<td>Deg</td>
<td>All</td>
<td>0</td>
<td>360</td>
</tr>
<tr>
<td>Wind speed</td>
<td>SPD</td>
<td>MPS</td>
<td>All</td>
<td>0</td>
<td>200</td>
</tr>
<tr>
<td>Zonal wind</td>
<td>U</td>
<td>MPS</td>
<td>All</td>
<td>0</td>
<td>200</td>
</tr>
<tr>
<td>Meridional wind</td>
<td>V</td>
<td>MPS</td>
<td>All</td>
<td>0</td>
<td>200</td>
</tr>
<tr>
<td>Visibility</td>
<td>VIS</td>
<td>M</td>
<td>All</td>
<td>0</td>
<td>300000</td>
</tr>
<tr>
<td>Cloud height</td>
<td>CLD</td>
<td>M</td>
<td>All</td>
<td>0</td>
<td>27000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>MSL</td>
<td>-500</td>
<td>500</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>SFC</td>
<td>-500</td>
<td>no maximum</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>1000</td>
<td>-500</td>
<td>500</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>925</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>850</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>700</td>
<td>2500</td>
<td>4000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>500</td>
<td>4000</td>
<td>7000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>400</td>
<td>6000</td>
<td>9000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>300</td>
<td>8000</td>
<td>11000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>250</td>
<td>9000</td>
<td>13000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>200</td>
<td>10000</td>
<td>14000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>150</td>
<td>11000</td>
<td>15000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>100</td>
<td>12000</td>
<td>18000</td>
</tr>
<tr>
<td>Height</td>
<td>Z</td>
<td>M</td>
<td>TRO</td>
<td>0</td>
<td>no maximum</td>
</tr>
</tbody>
</table>
Configuration file system

Some McIDAS-XCD subsystems require numerous settings for proper configuration. Because it is impractical to enter these fields from a McIDAS command line, a configuration script language and interface was developed. Users can write scripts with a text editor and the subsystem will get the information it needs from the text file instead of the command line. This mechanism is currently used for configuring communications ports and defining settings for decoders.

The syntax of the configuration script language is simple. It is designed to get specific values based on keywords and positional parameters. The McIDAS-XCD routines used to interface with this subsystem are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mcgtcfgstr</td>
<td>retrieves a value as a string from the file</td>
<td>C</td>
</tr>
<tr>
<td>mcgtcstr</td>
<td>retrieves a value as a string from the file</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>Mcgtcfgint</td>
<td>retrieves a value as an integer from the file</td>
<td>C</td>
</tr>
<tr>
<td>mcgtcint</td>
<td>retrieves a value as an integer from a file</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>Mcgtcdbl</td>
<td>retrieves a value as a double from a file</td>
<td>C</td>
</tr>
<tr>
<td>mcgtcdbl</td>
<td>retrieves a value as a double from a file</td>
<td>FORTRAN</td>
</tr>
</tbody>
</table>

The contents of a sample file are shown below.

```
# Cross reference list
# The cross reference list allows values to be accessed with
# multiple keyword names

:FLAGS [01]MD
:FLAGS [02]NR
:FLAGS [03]NC

MD=101     # first MD file in real-time range
NR=24      # number of rows to make for MD file
NC=500     # number of columns to make for MD file
WMO=FOUS FOUE  # list of WMO headers to decode
MINPRD=51  # minimum WMO product number to decode
MAXPRD=57  # maximum WMO product number to decode
DTIME=17.0 # number of hours to scan back in time to locate data
```

Below is sample code used to extract information.

```python
integer intval
double precision fitval
character*80 error , string
character*12 file

file = 'example.cfg'
ok = mcgtcstr (file , 'WMO' , 1 , string , error)
    string will contain 'FOUS'
ok = mcgtcdbl (file , 'DTIME' , 1 , fitval , error)
    fitval will contain 17.0
ok = mcgtcint (file , 'FLAGS[01]' , 1 , intval , error)
    intval will contain 101
```

The McIDAS-XCD decoders have a useful interface into the configuration scripts through the function m0dcinfo [C187-C195].
The McIDAS-XCD status display monitors the status of the ingestors and decoders. Operators can check the status display by running the STAT command in McIDAS, or statdisp from the Unix prompt. The functions for reading from and writing to the status display are m0rsdcd [A375] and m0wsdcd [A800], respectively. The positions in the memory structure written to the status display are found in xcd.inc, and are shown below.

<table>
<thead>
<tr>
<th>#</th>
<th>Decoder</th>
<th>Time</th>
<th>Beptr</th>
<th>Lasptr</th>
<th>Gridf</th>
<th>Grid MD</th>
<th>Grid Row</th>
<th>Grid Col</th>
<th>Text</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BBTASK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Name of the decoder, 8 characters maximum</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BBTIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Time stamp, hhmms</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BBBPTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Beginning index location being processed, usually set by m0ntidx</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BBLPTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Last index location to process, usually set by m0ntidx</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>BBMD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MD file number being processed</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>BBGRDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Grid file number being processed</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BBRD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MD file row number being written to</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BBGRID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Grid number being written to</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>BBBCOL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MD file column number being written to</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>BBTEXT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>General text, 12 characters maximum</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>BBIDXNM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Index file being processed, usually set by m0ntidx</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>BBON</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Status flag indicating the decoder is on</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>BBDAY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Julian day stamp</td>
<td></td>
</tr>
</tbody>
</table>

The information in the status display is stored on disk. McIDAS-XCD decoders read from/write to the file ~oper/mcidas/data/DECOSTAT.DAT.

If you write local decoders that use the status display, SSEC recommends writing your status to a different file so you won’t have to modify the file with each upgrade. To display local decoder status, assign a file name to the environment variable XCD disp file and start a second version of statdisp.

The structure for writing data to the status display is provided in the include file xcd.inc. Portions of this information may be written to the status display by the index block access routine m0nidx, but most information is provided by the decoder.

The typical way to write data to the status display is to fill the array bullbd with the appropriate values and to call the writing routine m0wsdcd [A792-A800], as shown in the example below.
INCLUDE 'xcd.inc'
IMPLICIT INTEGER (A-Z)

md=102
row=3
col=21
decnum=4

bullbd(BBON)= 1
bullbd(BBDAY)= yyddd
bullbd(BBTIME)= now
bullbd(BBMD)= md
bullbd(BBROW)= row
bullbd(BBCOL)= col

call movcw('TRJDEC', bullbd(BBTASK))
call movcw('TRAJECTORY', bullbd(BBTEXT))
ok = m0wsdcd ('LOCALSTA.DAT', 'DEC', decnum, bullbd, BBSIZE, jstat)

Once the call to m0wsdcd is complete, the status display looks like this:

## Decoder Time Begptr Endptr MD Row Col Text
4 TRJDEC 063423 160000 160160 102 3 21 TRAJECTORY

Station monitoring

The station tables delivered with McIDAS-XCD don't always contain all stations available for a data type. They may also contain obsolete stations no longer reporting a certain data type. McIDAS-XCD lets you monitor incoming stations in a decoder with the m0idnew function [A565]. This function keeps a running count of the number of times a station reports, and files the last day and time a station reported.

Use the command IDMON to display the station status. To add a station to a decoder type, enter: IDU EDIT stdid SWITCH=YES DEC=decname replacing stdid with the station to modify and decname with the name of the decoder to add to the station.

If you add a station to a decoder that is currently filing in an RA file (rafite), you must add that station separately with the BILDTEXT ADD command. For example: BILDTEXT ADD stdid rafite.

DECTEST

The command DECTEST is included in the McIDAS-XCD package. As a developer, you can link your text decoder into DECTEST during development to simulate how the decoder will work in a real-time data monitor. It is a good way to expose problems while they are easy to locate.
Compiling and linking

To build decoders in the McIDAS-XCD environment, you typically write decoders as functions placed in a library and call these decoders from the McIDAS command line or scheduler. Decoder software is usually built using the McIDAS library archive script mcar and the McIDAS compilation script mccomp.

To compile the decoder function, m0trjdec.for, and place the object in the library libmylib.a, perform these two steps.

1. Compile the function m0trjdec.for.
   
   mccomp -I. -I/-mcidas/inc -c m0trjdec.for

2. Put the object code created in step 1 in the library libmylib.a.

   mcar libmylib.a m0trjdec.o

Now, compile the data monitor, dmlocal.pgm; link it with the appropriate libraries; and create the McIDAS executable, dmlocal.mx.

3. Compile the data monitor, dmlocal.pgm.

   mccomp -I. -I/-mcidas/inc -c dmlocal.pgm

4. Link the data monitor to the appropriate libraries and generate the McIDAS executable dmlocal.mx.

   mccomp -mcidas/lib/main.o dmlocal.o -L. -L/-mcidas/lib
   -L/-oper/mcidas/lib -lmylib -lxcdbl -lxcd -lmcidas -lx11
   -o dmlocal.mx
Integrating a local decoder in McIDAS-XCD

This section provides an example of how to decode and file NGM Trajectory Forecasts. These forecasts are sent twice daily in conjunction with the NGM model runs. This is not a high volume dataset, as the NCEP supplies trajectory forecasts for only about 40 stations. The reports are sent in WMO headers FOUS50-57. The decoder provided at the end of this section can be implemented to run either as-needed or automatically. The steps for making the decoder run automatically are in the last part of the exercise.

The raw data format is as follows:

<table>
<thead>
<tr>
<th>FOUS51</th>
<th>KWBC 181200</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAJECTORY FCST</td>
<td></td>
</tr>
<tr>
<td>181200Z</td>
<td>181800Z</td>
</tr>
<tr>
<td>LATLONPPP</td>
<td>LATLONPPP</td>
</tr>
<tr>
<td>BIL</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>850</td>
</tr>
<tr>
<td></td>
<td>SFC</td>
</tr>
<tr>
<td></td>
<td>LND</td>
</tr>
<tr>
<td></td>
<td>SFC</td>
</tr>
</tbody>
</table>

Part 1: Building the decoder and putting the data into MD files

Perform the six steps below to build the pieces of the decoder and put the data into MD files.

1. Build the schema template and register the schema. Since this is point source data at fixed locations, place the location-dependent information in the column header and the time-dependent information in the row header.

By default, you will want 10 rows for the MD file because there are five forecast periods per model run and two model runs per day.

The row header will contain the following:

- **DAY** Julian day of the model run that generated this data
- **TIME** time of the model run that generated this
Trajectory MD file structure

Column Header
ID
ST
ZS

ID = MSN
ST = WI
ZS = 265

Row Header
DAY
TIME
VDAY
VTIM

Data Header
MOD
T
TD
LAT
LON
P
LEV

Sample MD file structure

DAY = 95017
TIME = 120000
VDAY = 95017
VTIM = 120000

T = 285.26
TD = 283.26
T = 273.1
TD = 268.4
LAT = 43.1
LON = 89.4
P = 696
LEV = 700

T = 278.1
TD = 271.2
LAT = 44.1
LON = 87.1
P = 831
LEV = 850

T = 294.1
TD = 277.4
LAT = 41.1
LON = 88.3
P = 988
LEV = SFC

DAY = 95017
TIME = 120000
VDAY = 95017
VTIM = 180000
You will need one column for each station reporting trajectories. 
The column header will contain the following:

ID          station ID
ST          state
ZS          station elevation

The data header will contain the following:

MOD         modification flag
3 REPEAT     groups (SFC, 850 and 700)
T            temperature at the station
TD           dew point at the station
LAT          latitude of the parcel
LON          longitude of the parcel
P            pressure of the parcel
LEV          pressure level of the parcel (SFC, 850 or 700)

Once the schema is designed appropriately it is registered with the
LSCHE command.

See the sample trajectory MD file structure on the adjacent page.

2. Write a decoder that parses out the information into a known
structure. Remember to adhere to the standard calling sequence
[A0-A47] used by the core decoders if you want to test the decoder
using DECTEST. If the decoder will run in automatic mode, use
the configuration script interface of the function m0dinfo. The
standard calling sequence looks like this:

integer function m0trjdec(wmohdr, block, nlines, circuit, julday,
timdec, flags, cflags)

The input values are defined below.

wmohdr c**(*) line containing the WMO header
          for example: FOUS51 KWBC 231200
block c**(*) array containing the data portion
nlines integer number of lines in block that this
data section uses
circuit c**(*) data source; seldom used
julday integer Julian day for which the data is valid,
yyddd
timdec integer time of the data filing, hhmmss
flags integer array containing the integer values
          for this decoder; typically they will be:
          1 - error output flag (1 = active)
          2 - ID monitoring key
          3 - decoder number for the status display
          4 - base MD or grid file range
          5 - number of rows in the MD file
          6 - number of columns in the MD file
          7 - rapid text filing flag
cflags c(*) array containing the characters strings
for this decoder; typically they will be:
1 - error file name
2 - old station ID file name
3 - new station ID file name
4 - ID table name to use
5 - master station list to use
6 - pointer file for the RA format

3. Create the MD file, if it does not exist, and initialize its row and
column headers [A809-A1014]. Building the row headers is easy
[A979-A1000] because all the information is in the decoding
process. Building the column headers is more difficult because the
only information you have is the station ID, and you also need the
state and station elevation.

The McIDAS-XCD function mobildid [A959] builds a station ID
table from parameters that you suggest. Since you need the station
ID, state and station elevation for the column header, send your
request to mobildid to build a file with these parameters [A1002-
A1009]. Once the ID table is built, you can build the column
headers for the MD file.

4. When the data is decoded, find the column number in the MD file
where the station will be filed. Use the function m0loccol to find
the appropriate column number; you must provide the station ID
and MD file number [A637-A640]. This function may appear
cumbersome, but it was designed to work with an unlimited
number of MD files simultaneously. This is useful when you have
multiple decoders activated by the same data monitor.

5. Convert the known decoded trajectory structure to the format that
MDO expects and file the data in the MD file. The conversion
process may include unit conversions (Celsius to Kelvin), type
conversion (floating point to scaled integer), or some type of
symbol conversion.

6. Write the output to the decoder status display using m0wsdced
[A792-A800], and process the next observation to decode.
Part 2: Automating the decoder process

Use the three steps below to make the decoder/filer an automatic process and connect it to McIDAS-XCD.

1. Create a data monitor. The easiest way to do this is to start with an existing one. If you use dmmisc.pgm as the template for dmlocal.pgm [C0-C540] and you use the standard decoder calling sequence enumerated in step 2 below, you will only have to change about 10 lines of code to install your decoder.

   For these trajectory products you will change:
   - `numdec` to 1 [C47]
   - `task` to DMLOCAL [C53]
   - `decnam` to TRJDEC [C157]

   The actual call to the decoder will look like this [C435-C439]:

   ```
   if (decnam(dec) .eq. 'TRJDEC') then
     decok = m0trjdec(cblk(1),cblk(2),
     & nlines-1,circit,yyddd,time,flags(1,dec),cflags(1,dec))
   endif
   ```

2. Set up a configuration file to tell the system where to look for data, the MD file range in which to put the data, the number of rows and columns in the MD file, and the name of the station ID table.

   Assume that you want the data filed in MD files 101 to 110 and the MD files are 10 rows by 600 columns. The data comes in as WMO headers FOUS50-57. Your configuration file, TRJDEC.CFG, might look like the one below.
# TRJDEC.CFG - Configuration file for the Trajectory forecasts
# --------------------- Cross reference List ---------------------
: FLAGS[01] ERRORFLG
: FLAGS[02] IDMNONFLG
: FLAGS[03] DISPLAYNUM
: FLAGS[04] MDF
: FLAGS[05] NROWS
: FLAGS[06] NCOLS

: CFLAGS[01] ERRORFILE
: CFLAGS[02] OLDIDFILE
: CFLAGS[03] NEWIDFILE
: CFLAGS[04] IDTABLE
: CFLAGS[05] MASTERFILE

# --------------------- End Of Cross Reference List ---------------------
# - You can modify any of the fields below -
# decoder description
DESCRIPTION="FOUS50-57 Decoder"

# which indices to search for this decoder
INDEX=PO

# which specific WMO headers to activate the decoder for
WMO=FOUS
MINPRD=50
MAXPRD=57

# which specific station origins to activate the decoder for
ORIGIN=KWBC

ERRORFLG=0 # error output flag set to 1 to activate
ERRORFILE=FO50DEC.ERR # error file name
IDMNONFLG=0 # station id monitoring activation flag
OLDIDFILE=OLDFO50.IDM # set to 1 or 3 to monitor new stations
# set to 2 or 3 to monitor old stations
NEWIDFILE=NEWFO50.IDM
DISPLAYNUM=1
MDF=101 # decoder number on status display
NROWS=10 # first real-time MD file number to use for decoder
NCOLS=600 # number of rows to make for MD file
IDTABLE=FO50DEC.IDT # number of columns to make for MD file
MASTERFILE=MASTERID.DAT # ID file to build when creating MD file

3. Configure the system so STARTXCD activates the data monitor
DMLOCAL and starts the decoder TRJDEC. Enter the three
commands below from the McIDAS command line, logged on as
oper.

DECINFO ADD DMLOCAL DEC=TRJDEC
DECINFO EDIT DMLOCAL TRJDEC ACTIVE
CONFIG=TRJDEC.CFG

DECINFO SET DMLOCAL ACTIVE

The next time STARTXCD samples the system configuration, it
will see that DMLOCAL is active and will try to start it.
Developing local decoders for your site

Below is a list of local decoders that you may want to develop at your site.

**Binary**
- BUFR format
- Lightning

**Text**

<table>
<thead>
<tr>
<th>Description</th>
<th>WMO Header</th>
<th>Text Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hurricane forecast positions</td>
<td>WT</td>
<td>Plain</td>
</tr>
<tr>
<td>Model output text</td>
<td>FO</td>
<td>Tabular</td>
</tr>
<tr>
<td>BATHY (bathythermal obs)</td>
<td>SO</td>
<td>Formatted</td>
</tr>
<tr>
<td>WAVEOB (spectral wave obs)</td>
<td>SX</td>
<td>Formatted</td>
</tr>
<tr>
<td>C-MAN (automated pier obs)</td>
<td>SX</td>
<td>Formatted</td>
</tr>
<tr>
<td>WINTEM (aviation wind/temp forecast)</td>
<td>FB</td>
<td>Tabular</td>
</tr>
<tr>
<td>ARFOR (area forecast for aviation)</td>
<td>FA</td>
<td>Tabular</td>
</tr>
<tr>
<td>SARAD (satellite clear radiance obs)</td>
<td>TR</td>
<td>Formatted</td>
</tr>
<tr>
<td>SATEM (satellite remote upper air soundings)</td>
<td>TH</td>
<td>Formatted</td>
</tr>
<tr>
<td>Frontal analysis</td>
<td>AS</td>
<td>Formatted</td>
</tr>
<tr>
<td>Forecasted frontal analysis</td>
<td>FS</td>
<td>Formatted</td>
</tr>
<tr>
<td>Winds aloft forecast</td>
<td>FD</td>
<td>Tabular</td>
</tr>
<tr>
<td>Satellite derived cloud information</td>
<td>TB</td>
<td>Tabular</td>
</tr>
<tr>
<td>Daily climate summaries</td>
<td>CS</td>
<td>Plain</td>
</tr>
<tr>
<td>Dropsonde</td>
<td>UZ</td>
<td>Formatted</td>
</tr>
<tr>
<td>SFLOC (Report of geographic location of atmospherics)</td>
<td>SF</td>
<td>Formatted</td>
</tr>
<tr>
<td>River and rainfall observations</td>
<td>SR</td>
<td>Plain</td>
</tr>
<tr>
<td>Aircraft recon data</td>
<td>UR</td>
<td>Formatted</td>
</tr>
<tr>
<td>Coded city forecasts</td>
<td>FP</td>
<td>Formatted</td>
</tr>
<tr>
<td>Convective outlook</td>
<td>AC</td>
<td>Plain</td>
</tr>
<tr>
<td>Coded upper air forecasts</td>
<td>FU</td>
<td>Formatted</td>
</tr>
</tbody>
</table>
**Useful functions in McIDAS-XCD**

The library functions described below are provided in the McIDAS-XCD package in two separate libraries. Functions called by the McIDAS-XCD user commands reside in the library `~mcidas/lib/libxcdcli.a`. The source code associated with `libxcdcli.a` is located in `~mcidas/xcd1.X/src`. Functions specific to the server portion of the package are in `~oper/mcidas/lib/libxcd.a`. The source code associated with `libxcd.a` is located in `~oper/mcidas/xcd1.X/src`.

### Ingestors

<table>
<thead>
<tr>
<th>Source file</th>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m0filblk.for</td>
<td>libxcd.a</td>
<td>files a text data block in the native McIDAS-XCD format</td>
</tr>
<tr>
<td>M0pt_utils.c</td>
<td>libxcd.a</td>
<td>group of functions that configures and reads from communications ports</td>
</tr>
</tbody>
</table>

### Data retrieval

<table>
<thead>
<tr>
<th>Source file</th>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcfngrd.for</td>
<td>libxcdcli.a</td>
<td>returns a list of grids in a grid file given search conditions</td>
</tr>
<tr>
<td>mcgetidx.for</td>
<td>libxcdcli.a</td>
<td>retrieves the index block directory given search conditions</td>
</tr>
<tr>
<td>mclddatb.for</td>
<td>libxcdcli.a</td>
<td>loads the raw text of a data block</td>
</tr>
<tr>
<td>mlcdbin.for</td>
<td>libxcdcli.a</td>
<td>loads the list of text index files</td>
</tr>
<tr>
<td>mctxtopn.for</td>
<td>libxcdcli.a</td>
<td>opens an RA text file</td>
</tr>
<tr>
<td>mctxxred.for</td>
<td>libxcdcli.a</td>
<td>reads an observation from an RA file given search conditions</td>
</tr>
<tr>
<td>Mctrgdfile.c</td>
<td>libxcdcli.a</td>
<td>retrieves the list of real-time grid files given search conditions</td>
</tr>
<tr>
<td>Mctrmmodels.c</td>
<td>libxcdcli.a</td>
<td>retrieves the list of real-time grid files</td>
</tr>
<tr>
<td>M0getspbyt.c</td>
<td>libxcd.a</td>
<td>retrieves bytes from a spool file</td>
</tr>
<tr>
<td>m0spnam.for</td>
<td>libxcdcli.a</td>
<td>creates the *.XCD file name for a given day and source</td>
</tr>
<tr>
<td>m0txget.for</td>
<td>libxcdcli.a</td>
<td>retrieves an observation from an RA file</td>
</tr>
<tr>
<td>m0txput.for</td>
<td>libxcd.a</td>
<td>writes an observation to an RA file</td>
</tr>
</tbody>
</table>
### Station IDs

<table>
<thead>
<tr>
<th>Source file</th>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcellest.for</td>
<td>libxcdcli.a</td>
<td>returns a list of stations interactively from the cursor</td>
</tr>
<tr>
<td>mcid2idn.for</td>
<td>libxcdcli.a</td>
<td>converts station ID to station block number</td>
</tr>
<tr>
<td>mclidids.for</td>
<td>libxcdcli.a</td>
<td>loads a list of station IDs from a group list</td>
</tr>
<tr>
<td>m0bildid.for</td>
<td>libxcdcli.a</td>
<td>builds a station ID table based on selection criteria</td>
</tr>
<tr>
<td>m0idnew.for</td>
<td>libxcd.a</td>
<td>monitors new and old station acquisition tables</td>
</tr>
</tbody>
</table>

### String parsing

<table>
<thead>
<tr>
<th>Source file</th>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcextrln.for</td>
<td>libxcd.a</td>
<td>retrieves parameters from a fixed format string</td>
</tr>
<tr>
<td>Mcgtpstrg.c</td>
<td>libxcd.a</td>
<td>parses a formatted string that contains field separators</td>
</tr>
<tr>
<td>mcsenblk.for</td>
<td>libxcdcli.a</td>
<td>scans an entire data block looking for string matches</td>
</tr>
<tr>
<td>m0dcsplt.for</td>
<td>libxcdcli.a</td>
<td>tokenizes a string into its basic components</td>
</tr>
<tr>
<td>m0grbval.for</td>
<td>libxcdcli.a</td>
<td>retrieves a component of a m0dcspltED tokenized string</td>
</tr>
<tr>
<td>m0parobs.for</td>
<td>libxcd.a</td>
<td>parses a SYNOPTIC style text string</td>
</tr>
</tbody>
</table>
### General utilities

<table>
<thead>
<tr>
<th>Source file</th>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mccidfit.for</td>
<td>libxcd.a</td>
<td>returns a reasonable default contour interval for a given parameter/level</td>
</tr>
<tr>
<td>mcfiltim.for</td>
<td>libxcd.a</td>
<td>returns the nominal time of an observation</td>
</tr>
<tr>
<td>Mcibmfloat.c</td>
<td>libxcd.a</td>
<td>converts IBM floating point representation to machine native format</td>
</tr>
<tr>
<td>Mcisbitset.c</td>
<td>libxcd.a</td>
<td>tells whether a bit in a buffer is set</td>
</tr>
<tr>
<td>Mcpackbit.c</td>
<td>libxcd.a</td>
<td>packs bits into a buffer</td>
</tr>
<tr>
<td>Mcupackbit.c</td>
<td>libxcd.a</td>
<td>unpacks bits from a buffer</td>
</tr>
<tr>
<td>mcydd2ch.for</td>
<td>libxcd.a</td>
<td>converts Julian day to a variety of formats</td>
</tr>
<tr>
<td>m0af2wmo.for</td>
<td>libxcd.a</td>
<td>converts AFOS PILs to WMO headers</td>
</tr>
<tr>
<td>m0grserr.forl</td>
<td>libxcd.a</td>
<td>performs gross error checks on known meteorological parameters</td>
</tr>
<tr>
<td>m0isrung_.c</td>
<td>libxcd.a</td>
<td>tells whether a Unix PID is active</td>
</tr>
<tr>
<td>m0rscd.for</td>
<td>libxcd.a</td>
<td>reads from a status display</td>
</tr>
<tr>
<td>m0wscd.for</td>
<td>libxcd.a</td>
<td>writes to a status display</td>
</tr>
<tr>
<td>M0unixuser.c</td>
<td>libxcd.a</td>
<td>returns the user ID of the session</td>
</tr>
</tbody>
</table>
Sample decoder

A 0: c $ Name:
A 1: c $ m0trjdec - decode trajectory forecasts from FOUS51-57 into
A 2: c $ TRAJ schema MD file
A 3: c $ Interface:
A 4: c $ integer function
A 5: c $ m0trjdec(character*(*) wmo, character*(*) cblk(*),
A 6: c $ integer nlines, integer jlday, integer timdec,
A 7: c $ character*(*) circuit, integer flags(*),
A 8: c $ character*(*) cflags(*))
A 9: c $ Input:
A 10: c $ wmo - wmo header of data block
A 11: c $ cblk - array containing raw text data
A 12: c $ nlines - number of lines in cblk used
A 13: c $ jlday - julian day of the data
A 14: c $ timdec - time stamp from the data (hmmss)
A 15: c $ circuit - source of the data
A 16: c $ flags - array of integer flags
A 17: c $ 3 - decoder display number
A 18: c $ 4 - base md file in range
A 19: c $ 5 - number of rows for the file
A 20: c $ 6 - number of columns
A 21: c $ cflags - array of character strings
A 22: c $ 3 - file name to use to contain
A 23: c $ 4 - the list of new stations
A 24: c $ 5 - station id file to use
A 25: c $ Input and Output:
A 26: c $ Output:
A 27: c $ Return values:
A 28: c $ 0 - success
A 29: c $ Remarks:
A 30: c $ The RA text file for this file format is initialized by
A 31: c $ the keyin.
A 32: c $ BILDTXT INIT FO50.RAP FO50.RAT 600 3 C4 5 12 80 FOUS14 X 1
A 33: c $ The resultant MD file schema used is the TRAJ schema.
A 34: c $ This decoder is intended as an example only. It was built
A 35: c $ to demonstrate many of the features at the disposal of the
A 36: c $ developer in writing local decoders.
A 37: c $ integer function m0trjdec(wmo,cblk,nlines,jlday,timdec,
A 38: c $ & circuit,flags,cflags)
A 39: c $ include 'xcd.inc'
A 40: c $ implicit none
A 41: c $ maxlev - maximum number of trajectory levels
A 42: c $ maxper - maximum number of forecast periods
A 43: c $ maxsta - maximum number of stations
A 44: c $ maxids - maximum number of stations in this data block
A 45: c $ ndkeys - number of data keys in md file
A 46: c $ integer maxlev
A 47: c $ integer maxper
A 48: c $ integer maxsta
<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:</td>
<td>integer maxids</td>
</tr>
<tr>
<td>64:</td>
<td>integer ndkeys</td>
</tr>
<tr>
<td>65:</td>
<td>parameter (maxlev = 3 , maxper = 5 , maxsta = 600)</td>
</tr>
<tr>
<td>66:</td>
<td>parameter (maxids = 30 , ndkeys = 19)</td>
</tr>
<tr>
<td>67:</td>
<td>character(<em>) cbk(</em>) , cflags(*) , circit , wmo</td>
</tr>
<tr>
<td>68:</td>
<td>integer flag(*)</td>
</tr>
<tr>
<td>70:</td>
<td>c--- daylist - array containing recent days of the month</td>
</tr>
<tr>
<td>71:</td>
<td>c--- yyddd - julian days associated with recent days of the month</td>
</tr>
<tr>
<td>72:</td>
<td>c--- integer daylist(4)</td>
</tr>
<tr>
<td>73:</td>
<td>integer yyddd(4)</td>
</tr>
<tr>
<td>74:</td>
<td>integer idinfo(6)</td>
</tr>
<tr>
<td>75:</td>
<td>c--- vday - list of forecast valid julian days</td>
</tr>
<tr>
<td>76:</td>
<td>integer vday(maxper)</td>
</tr>
<tr>
<td>77:</td>
<td>c--- vtime - list of forecast valid times (hhmmss)</td>
</tr>
<tr>
<td>78:</td>
<td>integer vtime(maxper)</td>
</tr>
<tr>
<td>79:</td>
<td>c--- pre - list of pressure levels of parcels</td>
</tr>
<tr>
<td>80:</td>
<td>integer pre(maxper , maxlev , maxids)</td>
</tr>
<tr>
<td>81:</td>
<td>c--- lat - list of latitudes of parcels</td>
</tr>
<tr>
<td>82:</td>
<td>c--- lon - list of longitudes of parcels</td>
</tr>
<tr>
<td>83:</td>
<td>integer lat(maxper , maxlev , maxids)</td>
</tr>
<tr>
<td>84:</td>
<td>integer lon(maxper , maxlev , maxids)</td>
</tr>
<tr>
<td>85:</td>
<td>c--- linloc - location in data block where each ob is found</td>
</tr>
<tr>
<td>86:</td>
<td>c--- for level and station</td>
</tr>
<tr>
<td>87:</td>
<td>integer linloc(maxlev , maxids)</td>
</tr>
<tr>
<td>88:</td>
<td>c--- bcol - beginning column for a forecast</td>
</tr>
<tr>
<td>89:</td>
<td>integer bcol(maxper)</td>
</tr>
<tr>
<td>90:</td>
<td>c--- ecol - ending column for a forecast</td>
</tr>
<tr>
<td>91:</td>
<td>integer ecol(maxper)</td>
</tr>
<tr>
<td>92:</td>
<td>c--- tlat - temporary latitude array for storing maxper latitudes for one level</td>
</tr>
<tr>
<td>93:</td>
<td>c--- tlon - temporary longitude array for storing maxper longitude for one level</td>
</tr>
<tr>
<td>94:</td>
<td>c--- tpref - temporary pressure array for storing maxper pressure for one level</td>
</tr>
<tr>
<td>95:</td>
<td>integer tlat(maxper)</td>
</tr>
<tr>
<td>96:</td>
<td>integer tlon(maxper)</td>
</tr>
<tr>
<td>97:</td>
<td>integer tpref(maxper)</td>
</tr>
<tr>
<td>98:</td>
<td>integer staton - list of all possible stations that could file this type of report</td>
</tr>
<tr>
<td>99:</td>
<td>integer slat(maxsta)</td>
</tr>
<tr>
<td>100:</td>
<td>integer stlon(maxsta)</td>
</tr>
<tr>
<td>101:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>102:</td>
<td>integer slat(maxsta)</td>
</tr>
<tr>
<td>103:</td>
<td>integer slat(maxsta)</td>
</tr>
<tr>
<td>104:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>105:</td>
<td>integer stlon(maxsta)</td>
</tr>
<tr>
<td>106:</td>
<td>integer stlon(maxsta)</td>
</tr>
<tr>
<td>107:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>108:</td>
<td>integer slat(maxsta)</td>
</tr>
<tr>
<td>109:</td>
<td>integer slat(maxsta)</td>
</tr>
<tr>
<td>110:</td>
<td>integer slat(maxsta)</td>
</tr>
<tr>
<td>111:</td>
<td>integer slat(maxsta)</td>
</tr>
<tr>
<td>112:</td>
<td>integer slat(maxsta)</td>
</tr>
<tr>
<td>113:</td>
<td>integer stlon(maxsta)</td>
</tr>
<tr>
<td>114:</td>
<td>integer stlon(maxsta)</td>
</tr>
<tr>
<td>115:</td>
<td>integer stlon(maxsta)</td>
</tr>
<tr>
<td>116:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>117:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>118:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>119:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>120:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>121:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>122:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>123:</td>
<td>integer staton(maxsta)</td>
</tr>
<tr>
<td>124:</td>
<td>c--- line - temporary array used for cracking raw text</td>
</tr>
<tr>
<td>125:</td>
<td>integer line(20)</td>
</tr>
<tr>
<td>126:</td>
<td>mdbufase - base md file number</td>
</tr>
<tr>
<td>127:</td>
<td>mdrow - md row number to write to</td>
</tr>
<tr>
<td>128:</td>
<td>mdcol - md column number to write to</td>
</tr>
<tr>
<td>129:</td>
<td>mdrow - number of rows to make for md file</td>
</tr>
<tr>
<td>130:</td>
<td>mdcol - number of columns to make for md file</td>
</tr>
<tr>
<td>131:</td>
<td>c--- record - array containing decoded data section for writing to md file</td>
</tr>
</tbody>
</table>
A 134:  integer mdbase
A 135:  integer mdrow
A 136:  integer mdcoll
A 137:  integer nr
A 138:  integer nc
A 139:  integer record(ndkeys)
A 140:  
A 141:  c---  mlocs - array containing md file positions for all
A 142:  c---  of the parameters in the data section
A 143:  integer mlocs(ndkeys)
A 144:  
A 145:  c---  midtb - array containing list of stations in column
A 146:  c---  headers for m0loccol
A 147:  integer midtb(maxsta)
A 148:  
A 149:  c---  blkdom - block domain used for RA
A 150:  c---  timdom - time domain used for RA
A 151:  c---  ptrhed - pointer header for RA
A 152:  c---  idtab - station id table for RA
A 153:  integer blkdom(4)
A 154:  integer timdom(8)
A 155:  integer ptrhed(THSIZE)
A 156:  integer idtab(maxsta)
A 157:  
A 158:  c---  t - array containing all temperature forecasts for
A 159:  c---  the final period
A 160:  c---  td - array containing all dew point forecasts for
A 161:  c---  the final period
A 162:  double precision t(maxlev,maxids)
A 163:  double precision td(maxlev,maxids)
A 164:  
A 165:  c---  tt - temporary temperature storage
A 166:  c---  ttd - temporary dew point storage
A 167:  double precision tt
A 168:  double precision ttd
A 169:  
A 170:  c---  cvalue - temporary string value used in mcgrpval
A 171:  character*12 cvalue
A 172:  
A 173:  c---  stafl - file containing list of all possible stations
A 174:  character*12 stafl
A 175:  
A 176:  c---  cid - array of decoded station ids
A 177:  character*12 cid(maxids)
A 178:  
A 179:  c---  ctemp - temporary string
A 180:  character*12 ctemp
A 181:  
A 182:  c---  ctemp4 - temporary string
A 183:  character*4 ctemp4
A 184:  
A 185:  c---  cline - temporary string used for debug messages
A 186:  character*80 cline
A 187:  
A 188:  c---  ptrfl - pointer file for RA
A 189:  character*12 ptrfl
A 190:  
A 191:  c---  lvls - array containing observation levels for each
A 192:  c---  station
A 193:  character*4 lvls(maxlev,maxids)
A 194:  
A 195:  c---  tlv1 - temporary storage variable for observation level
A 196:  character*4 tlv1
A 197:  
A 198:  c---  clit - function declaration
A 199:  c---  rit - function declaration
A 200:  c---  incday - function declaration
A 201:  c---  dectrj - function declaration
A 202:  c---  maken - function declaration
A 203:  c---  mod - function declaration
A 204:  c---  mctxtopn- function declaration
A 205: c--- mcgrssl- function declaration
A 206: c--- mclodids- function declaration
A 207: c--- m0dcsplt- function declaration
A 208: c--- m0loccol- function declaration
A 209: c--- mctxtwrt- function declaration
A 210: c
A 211: character*4 clit
A 212: integer lit
A 213: integer incday
A 214: integer mctxttopn
A 215: integer mcgrsscl
A 216: integer mclodids
A 217: integer m0dcsplt
A 218: integer dectrj
A 219: integer makmdf
A 220: integer m0loccol
A 221: integer mctxtwrt
A 222: integer mdo
A 223: c---
A 224: nlines - number of lines used in cblk
A 225: integer nlines
A 226: c
A 227: c--- julday - julian day the data represents
A 228: c--- timdec - current time
A 229: integer julday
A 230: integer timdec
A 231: c
A 232: c--- firstcl - flag indicating this is the first call to dectrj
A 233: integer firstcl
A 234: c
A 235: c--- opnrnap - return value from mctxttopn
A 236: integer opnrnap
A 237: c
A 238: c--- i,j,k - loop counters
A 239: integer i
A 240: integer j
A 241: integer k
A 242: c
A 243: c--- decnum - decoder number to write to on status display
A 244: integer decnum
A 245: c
A 246: c--- month - month number for julday
A 247: c--- year - year for julday
A 248: c
A 249: integer month
A 250: integer year
A 251: c
A 252: c--- numrnap - number of stations stored in RA file
A 253: integer numrnap
A 254: c
A 255: c--- stat - status returned from m0wsdcd and m0rsdcd
A 256: c
A 257: integer stat
A 258: c
A 259: c ok - function return value
A 260: integer ok
A 261: c
A 262: c--- numlod - number of stations loaded from m0loccol
A 263: integer numlod
A 264: c
A 265: c--- linnum - internal line number counter
A 266: integer linnum
A 267: c
A 268: c--- numgrp - number of groups tokenized by m0dcsplt
A 269: integer numgrp
A 270: c
A 271: c--- dum - dummy variable
A 272: c--- dum1 - dummy variable
A 273: c--- dum2 - dummy variable
A 274: c--- dum3 - dummy variable
A 275: c--- dum4 - dummy variable
A 276: c--- dum5 - dummy variable
A 277: integer dum
A 278: integer dum1
A 279: integer dum2
A 280: integer dum3
A 281: integer dum4
A 282: integer dum5
A 283: c--- value - value returned from m0grbval
A 285: integer value
A 286: c--- group - group counter from m0dcsp1t
A 288: integer group
A 289: c--- numper - number of forecast periods
A 291: integer numper
A 292: c--- tday - temporary day storage
A 294: integer tday
A 295: c--- numid - counter for number of stations decoded
A 297: integer numid
A 298: c--- nostat - flag indicating if a station id had been found
A 300: integer nostat
A 301: c--- clat - station latitude
A 303: c--- clon - station longitude
A 304: integer clat
A 305: integer clon
A 306: c--- lev - pointer to the level 1-SFC 2-850 3-700
A 308: integer lev
A 309: c--- id - station id
A 311: integer id
A 312: c--- mdfile - mdfile number to make
A 314: integer mdfile
A 315: c--- mdlist - list of mdf files to use for m0loccol
A 317: integer mdlist
A 318: c--- pt - pointer describing where in 'record' array to
A 320: c--- store decoded data
A 321: integer pt
A 322: c--- ln - line number to use for filing
A 324: integer ln
A 325: c--- pres - type value used for storing in RA file
A 327: integer pres
A 328: c--- generate the md key locations for the data section of
A 334: c--- the observation
A 335: c--- do 1 i = 1, ndkeys
A 336: A 337: mdlocs(i) = i + 7
A 338: 1 continue
A 339: c--- assign values for decoder number, base md file, number
A 341: c--- of rows and columns to make md file based on input from
A 342: c--- data monitor
A 343: c--- decnum = flags(3)
A 344: mdbase = flags(4)
A 345: nr = flags(5)
A 346: nc = flags(6)
A 348: calculate the day of the month to julian day conversions
A 350: this is necessary because the report retrieved from the
A 351: raw text is day of month but we need julain day for filing.
A 352: the 4 values stored will be value from julday and the
A 353: 3 following days.
A 354: yyydd(1) = julday
A 356: do 5 i = 1, 4
A 357: call yymdmy(yydd(i),daylst(i),month,year)
A 358: if (i .lt. 4) yyydd(i+1) = incday(yydd(i),1)
A 359: continue
A 360: initialize the arrays for the RA files if this is the
A 362: first call to the decoder
A 363: ptrfil = cflags(6)
A 365: if (firstcl .eq. 0) then
A 366: firstcl = 1
A 367: opnrap = mctxttopn(ptrfil, ptrhed, maxsta, idtab, numrap)
A 368: endif
A 369: m0trjdec = 0
A 370: stafil = cflags(5)
A 372: get the values currently stored in the status display
A 374: call m0radsod(' ',decsnum,'DECO',bullbd,BBSIZE,stat)
A 376: acquire the list of stations/lat/lons possible for
A 378: this data type, this is used to determine station lat/lon
A 379:
A 380: ok = melodids(stafil,maxsta,staton,slat,slon,numlmd)
A 381: call ddest('number of stations loaded ','numlmd')
A 382:
A 383: b------------------------------------------------------------
A 384: c--- * the actual decoding begins here
A 385: b------------------------------------------------------------
A 386:
A 387: c--- we will look for a line that contains 6 digits
A 388: c--- immediately followed by a single character 'Z'. this will
A 389: c--- be the valid forecast times of the trajectory data.
A 390: ex.: 1412002
A 391: linnum = 0
A 393: 200 continue
A 395: grab one line at a time until you reach the end of
A 397: the data block
A 399: linnum = linnum + 1
A 400: if (linnum .le. nlines) then
A 401: tokenize the line into decodable groups
A 403: call ddest(cblk(linnum),0)
A 405: numgrp = 0
A 407: call movc(80,cblk,(linnum-1)*80,line,0)
A 409: ok = m0dcspit(80,numgrp,0,dum)
A 410: print debug messages showing how the string was
A 411: tokenized otherwise loop 210 is frivolous
A 413: do 210 i = 1, numgrp
A 414: call m0grval(cvalue, value, i)
A 415: write(cline,FMT='(4(a2,i5,1x),a12)'
A 416: & 'gp',i,'pt=',point(i),'tp=',type(i),
A 417: & 'nc=',numch(i),cvalue
A 418: call ddest(cline,0)
A 419: continue
A 420: c--- this is where you actually scan for 6 digits
A 422: c--- followed immediately by the character Z.
A 423:
A 424: group = 0
A 425: numper = 0
A 426: 220 continue
A 427:
A 428: c--- scan through the tokenized groups of the line until
A 429: c--- you have no more groups to check for this line
A 430:
A 431: group = group + 1
A 432: if (group .i.e. numgrp) then
A 433:
A 434: c--- if this group is NOT 6 digits immediately followed
A 435: c--- by a single character, it cannot be the valid
A 436: c--- forecast time, so go check the next group
A 437:
A 438: if (type(group) .ne. ADIGIT .or.
A 439: & numch(group) .ne. 6 .or.
A 440: & type(group+1) .ne. ACHAR .or.
A 441: & numch(group+1) .ne. 1) goto 220
A 442:
A 443: c--- if you have made it to here, you know you have
A 444: c--- a forecast period
A 445:
A 446: numper = numper + 1
A 447: call woodtval(cvalue , value , group)
A 448:
A 449: c--- extract the valid date and convert to julian
A 450:
A 451: tday = value / 10000
A 452: do 230 j = 1 , 4
A 453: if (daylst(j) .eq. tday) then
A 454: tday = yydld(j)
A 455: goto 235
A 456: endif
A 457: 230 continue
A 458: 235 continue
A 459:
A 460: vday(numper) = tday
A 461:
A 462: c--- extract the valid time and convert to hhmmss
A 463:
A 464: vtime(numper) = mod(value , 10000) * 100
A 465:
A 466: c--- if the number of periods is less than maxper,
A 467: c--- continue extracting periods from the line
A 468:
A 469: if (numper .lt. maxper) goto 220
A 470:
A 471: c--- if you have made it to here, you have all the
A 472: c--- periods, so go to the next point in the decoding
A 473:
A 474: goto 290
A 475:
A 476: endif
A 477:
A 478: c--- if you have made it to here, you haven't found the
A 479: c--- forecast periods yet. go back and grab a new line.
A 480:
A 481: goto 200
A 482:
A 483: else
A 484: call sdest('unable to find valid forecast labels',0)
A 485: goto 999
A 486: endif
A 487:
A 488: 290 continue
A 489:
A 490: c--- if you have made it to here, you have all the forecast
A 491: c--- dates and times in 'vday' and 'vtime'.

McIDAS Developer/Operator Training
October 24, 1995
Developing Local Decoders in McIDAS-XCD
8-39
A A 492: c--- loop 295 is printing the valid times that were decoded
A A 494: do 295 i = 1 , maxper
A A 496: write(cline,FMT='(a4,i12,1x,a5,i5,1x,a5,i6.6)')
A A 497: & 'per=',i,'vday=',vday(i),',vtime(i),',vtim(i)
A A 498: call ddest(cline,0)
A A 499: 295 continue
A A 500:
A A 501: c--- now that we have the forecast information, we will scan
A A 502: c--- each line for decodable data. Since this format only
A A 503: c--- sends the station id once regardless the number of levels
A A 504: c--- forecasted we will assume that lines that do not contain
A A 505: c--- station ids have the same station id as the previous line
A A 506:
A A 507: numid = 0
A A 508: nostat = 0
A A 509: linnum = linnum + 1
A A 510: 300 continue
A A 511:
A A 512: c--- loop through the remaining number of lines until you
A A 513: c--- reach the end of the data block
A A 514:
A A 515: linnum = linnum + 1
A A 516: if (linnum .ie. nlines)then
A A 517:
A A 518: c--- initialize the output arrays to missing value codes
A A 519:
A A 520: call zmiss(maxper , tlat , MISS)
A A 521: call zmiss(maxper , tlon , MISS)
A A 522: call zmiss(maxper , tpre , MISS)
A A 523: tt = -9999.d0
A A 524: ttd = -9999.d0
A A 525:
A A 526: c--- decode the current line
A A 527:
A A 528: ok = dectrt(cblk(linnum),ctemp,tlat,tlon,tpre,tvl,t,ttd)
A A 529:
A A 530: c--- if we haven't found a station id yet and this line
A A 531: c--- does not include a station id then go grab the next
A A 532: c--- line and start over.
A A 533:
A A 534: if (nostat .eq. 1 .and. ok .ne. 1)goto 300
A A 535: nostat = 0
A A 536:
A A 537: c--- if the decoding detected a station id you must find
A A 538: c--- the lat/lon of the station
A A 539:
A A 540: if (ok .eq. 1)then
A A 541: A A 542: do 315 i = 1 , numlod
A A 543:
A A 544: c--- if a station match is found, assign the appropriate
A A 545: c--- lat/lon, increment the number of stations found, and
A A 546: c--- reset the level counter for the new station.
A A 547:
A A 548: ctemp4 = clit(statlon(i))
A A 549: if (ctemp(1:3) .eq. ctemp4(1:3))then
A A 550: numid = numid + 1
A A 551: cid(numid) = ctemp
A A 552: clat = slat(i)
A A 553: clon = slon(i)
A A 554: lev = 0
A A 555: goto 316
A A 556: endif
A A 557:
A A 558: 315 continue
A A 559:
A A 560: c--- if the station was not found and the flag is set
A A 561: c--- at the station to the monitor list
A A 562: A
A 563: if (flags(2) .eq. 1) then
A 564:   id = lit(citmp4)
A 565: call m0idnew(id,cflags(3),1,'NEW ')
A 566: endif
A 567: nostat = 1
A 568: goto 300
A 569: endif
A 571: if you have made it to here you have successfully decoded
A 572: the entire observation. clat/clon contain the station
A 573: lat/lon's you are currently processing
A 574: 316 continue
A 575: A 576: move the station lat/lon into the final forecast period's
A 577: c---- in the temporary array
A 578: tlat(5) = clat
A 579: tlon(5) = clon
A 580: c---- increment the level number being viewed and moved the
A 581: data from the temporary arrays into the permanent ones
A 582: c---- the value for lev will be used to indicate where in the
A 583: repeat section of the md file to store this report
A 584: c---- if (tlvl .eq. 'SFC') then
A 585: lev = 1
A 586: elseif (tlvl .eq. '850') then
A 587: lev = 2
A 588: else
A 589: lev = 3
A 590: endif
A 591: c---- moved the data just decoded from temporary into permanent
A 592: c---- storage arrays.
A 593: t(lev,numid) = tt
A 594: td(lev,numid) = ttd
A 595: linloc(lev,numid) = linnum
A 596: lvs(lev,numid) = tlvl
A 597: call movw(maxper,tlat,linloc(1,lev,numid))
A 598: call movw(maxper,tlon,linnum(1,lev,numid))
A 599: call movw(maxper,tpre,pre(1,lev,numid))
A 600: A 601: go back up looking for more stations
A 602: return
A 603: goto 300
A 604: endif
A 605: c---- if you have made it to here all of the important arrays
A 606: have been filled so you can commence filing the reports
A 607: c---- first check to see if the MD file you will be filing to
A 608: c---- exists. if it doesn't build the row/column headers
A 609: c---- mdfile = mdbase + mod(julday,10)
A 610: if (mdfile .eq. mdbase) mdfile = mdfile + 10
A 611: ok = makmdf(mdfile,nr,nc,julday)
A 612: c---- **************************************************************
A 613: c---- * file decoded reports in MD and RA files
A 614: c---- **************************************************************
A 615: A 616: c---- loop through all of the stations that were decoded
A 617: c---- do 400 i = 1 , numid
A 618: write(cline,FMT='(a3,a4)' )'id=',cid(i)
A 619: call ddest(cline,0)
locate the appropriate column number to file data in

if (mdcol .le. 0) then
    goto 400
endif

goto 400

figure out the appropriate row number
if the first VTIM is 12Z that means this is the
output from the 12z model run so begin filing
after the 5 00Z rows

mdrow = 0
if (vtime(1) .gt. 0) mdrow = 5

record(1) = 0
do 440 k = 1 , maxper

mdrow = mdrow + 1

loop through the forecast periods calculating the
appropriate row number and filling up the record
array for the md file. This section also files
in the RA file if appropriate.
do 420 j = 1 , maxlev

fill the md data output array

the storage format for the output array 'record' is:
1 - MOD flag
2 - temperature from SFC (only reported for last period)
3 - dew point from SFC (only reported for last period)
4 - latitude from SFC
5 - longitude from SFC
6 - pressure level SFC
7 - ending pressure SFC
8 - temperature from 850
9 - dew point from 850
10 - latitude from 850
11 - longitude from 850
12 - pressure level 850
13 - ending pressure 850
14 - temperature from 700
15 - dew point from 700
16 - latitude from 700
17 - longitude from 700
18 - pressure level 700
19 - ending pressure 700

convert the temp and dewpoint to kelvin and put
in the appropriate record location for the final
forecast period.

pt = 1 + (j - 1) * 6
record(pt+1) = MISS
record(pt+2) = MISS
if (k .eq. maxper) then
    t(j,i) = 273.16 + t(j,i)
    td(j,i) = 273.16 + td(j,i)
    record(pt+1) = int(t(j,i) * 100.0d0)
    record(pt+2) = int(td(j,i) * 100.0d0)
    perform gross error checks on the temperature
    and dew point
A 707:     ok = mcgrsscl(record(pt+1),2,MISS,'T ',',K ',0)
A 708:     ok = mcgrsscl(record(pt+2),2,MISS,'TD ',',K ',0)
A 710:
A 711:     endif
A 712:
A 713:     record(pt+3) = lat(k,j,i)
A 714:     record(pt+4) = lon(k,j,i)
A 715:     record(pt+5) = pres(k,j,i)
A 716:     record(pt+6) = lit(lvls(j,i))
A 717:
A 718:     if the RA text file was successfully opened
A 719:     file the portion of the observation having
A 720:     c--
A 721:     c--
A 722:     c--
A 723:     c--
A 724:     if (opnrap.ge.0)then
A 725:     build the domain within the data block that
A 726:     should be filed
A 727:     c--
A 728:     since this is a bit of a flaky output, we will
A 729:     build a temporary string which is what we will
A 730:     actually file
A 731:     ln = linloc(j,i)
A 732:     cline = cid(i)(1:4)/cblok(ln)(6:9)
A 733:     &
A 734:     //cblok(ln)(bcol(k):ecol(k))
A 735:     blkdom contains the list of what section of the
A 736:     raw observation stored in 'cline' is to be
A 737:     c--
A 738:     included. This includes row/col information,
A 739:     not to be confused with the MD row and col
A 740:     blkdom(1) = 1
A 741:     blkdom(2) = blkdom(1)
A 742:     blkdom(3) = 1
A 743:     blkdom(4) = 80
A 744:
A 745:     build the time domain information.
A 746:     c--
A 747:     we store 1001 for SPC level.
A 748:
A 749:     pres = 1001
A 750:     if (lvls(j,i) .eq. '700')then
A 751:     pres = 700
A 752:     elseif (lvls(j,i) .eq. '850')then
A 753:     pres = 850
A 754:     endif
A 755:     timdom stores pertinent meta-data about the
A 756:     c--
A 757:     c--
A 758:     valid day/time
A 759:     c--
A 760:     c--
A 761:     timdom(1) = pres
A 762:     timdom(2) = 1
A 763:     timdom(3) = vday(k)
A 764:     timdom(4) = vtime(k)
A 765:     timdom(5) = timdom(4)
A 766:     timdom(6) = timdom(3)
A 767:     timdom(7) = timdom(4)
A 768:     timdom(8) = timdom(5)
A 769:     id = lit(cid(i))
A 770:     write the observation to the RA file
A 771:
A 772:     ok = mctxtwrt(ptrfil,ptrhed,cline,blkdom,
A 773:     &
A 774:     call ddest(cline(1:60),ok)
A 775:     endif
A 776:
A 777:     420 continue
at this point the entire data record should be filled so we can write the output to the md file

ok = mdo(mdfile, mdrow, mdcol, ndkeys, mdlocs, record)
write(cline, FMT='(a4,i2,1x,2(a2,i4,1x))')
'md0=', ok, "r=", mdrow, 'c=', mdcol

update the buffer for the status display and output it to the window

bullbd(BBON) = 1
call gettim(bullbd(BBTIME))
call getday(bullbd(BBDAY))
bullbd(BBMD) = mdfile
bullbd(BBROW) = mdrow
bullbd(BBCOL) = mdcol
call movcv('TRAJECT ',bullbd(BBTASK))
call movcv(' ',bullbd(BBTEXT))
call mswdcd(' ', decnum, 'DECO', bullbd, BBSIZE, stat)

continue
return
end

makmdf - make the md file with the necessary row and column headers
integer function makmdf(integer mdfile, integer nr, integer nc, integer julday)

input:
mdfile md file number to build
number of rows to make md file
number of columns to make md file
julday julian day md file is valid for

return values:
0 - md file already exists
1 - md file made successfully
<0 - error while making md file

integer function makmdf(mdfile, nr, nc, julday)

implicit none

include 'xcd.inc'

nkeys - number of keys to get from master station list
maxsta - maximum number of stations that can be stored in the column headers

integer nkeys
integer maxsta
parameter (nkeys = 3, maxsta = 600)

nrkeys - number of keys in the row header
nc - number of keys in the column header

integer nrkeys
integer nc
parameter (nrkeys = 4, nc = 3)

mdfile - md file to be built
nr - number of rows to make the md file
nc - number of columns to make the md file

integer mdfile
integer nr
integer nc
A 851: c--- julday  - julian day this md file represents
A 852: c--- record  - storage array for output to the md file
A 853: integer julday
A 854: integer record(nrkeys+nckeys)
A 855: c--- title  - array containing the title for the md file
A 856: character(8)
A 857: integer title(8)
A 858: c--- rklocs  - row key locations in the md file
A 859: c--- cklocs  - columns key locations in the md file
A 860: integer rklocs(nrkeys)
A 861: integer cklocs(nckeys)
A 862: c--- filnam  - temp string containing lw filename of md file
A 863: character*12 filnam
A 864: c--- idfile  - id file to use to build station list and
A 865: c--- column headers from
A 866: character*12 idfile
A 867: c--- ckeys  - list of keys to get from station id list
A 868: character*4 ckeys(nrkeys)
A 869: c--- cdate  - string to contain date for title of md file
A 870: character*40 cdate
A 871: c--- cfile  - string containing md file title
A 872: character*40 cfile
A 873: c--- filtmp  - integer array name of md file
A 874: integer filtmp(3)
A 875: c--- finc  - forecast increment between md file rows
A 876: integer finc
A 877: c--- ids  - array containing complete list of station ids
A 878: c--- to include in column headers
A 879: c--- ele  - array containing complete list of elevations
A 880: c--- to include in column headers
A 881: c--- st  - array containing complete list of states to
A 882: c--- include in column headers
A 883: integer ids(maxsta)
A 884: integer ele(maxsta)
A 885: integer st(maxsta)
A 886: c--- lwfile  - function declaration
A 887: c--- lit  - function declaration
A 888: c--- mcydd2ch- function declaration
A 889: c--- mddmake - function declaration
A 890: c--- mobildid- function declaration
A 891: c--- mdo  - function declaration
A 892: c--- mcinchr - function declaration
A 893: c--- lw  - function declaration
A 894: integer lwfile
A 895: integer lit
A 896: integer mcydd2ch
A 897: integer mddmake
A 898: integer mobildid
A 899: integer mdo
A 900: integer mcinchr
A 901: integer lw
A 902: c--- schema  - md schema name
A 903: c--- ok  - function return value
A 904: c--- nsta  - number of stations returned from mobildid
A 905: c--- start  - starting word in station id table to begin
A 906: c--- i    - loop counter
A 907: c--- loading station information
A 908: c--- integer schema
A 909: integer ok
A 910: integer nsta
A 911: integer schema
A 912: integer ok
A 913: integer nsta
A 923: integer start
A 924: integer i
A 925:
A 926: equivalence (filnam, filtmp)
A 927: data rklocs/1,2,3,4/, cklocs/5,6,7/
A 928:
A 929: c--- build the file name
A 930:
A 931: call mdbname(mdfile, filtmp)
A 932:
A 933: c--- check to see if the file already exists
A 934:
A 935: makmdf = 0
A 936: if (lwfile(filnam) .eq. 1) goto 999
A 937:
A 938: c--- if you have made it to here, the file doesn't exist so
A 939: c--- create the md file with row and column headers
A 940:
A 941: makmdf = -1
A 942:
A 943: c--- build the md file title and make the md file
A 944: schema = lit('TRAJ')
A 945: ok = mydd2ch(julday, 4, cdate)
A 946: cttl = 'NGM Traj Fcst: '//cdate
A 947: call movcw(cttl, title)
A 948: ok = mdbname(mdfile, schema, 0, nr, nc, julday, title)
A 949: if (ok .lt. 0) goto 999
A 950:
A 951: c--- first we must build a station list. For this exercise
A 952: c--- we will use the station ids used for the FOUS14 decoder
A 953:
A 954: idfile = 'FOUS51.IDT'
A 955: ckeys(1) = 'CID1'
A 956: ckeys(2) = 'ST '
A 957: ckeys(3) = 'ELE1'
A 958:
A 959: nsta = m0bilda(' ',1,idfile,ckeyes,nkeys,'DDS ',0,'FOUS14',
& 1,1,5)
A 960:
A 961:
A 962: c--- check for errors in building station id table
A 963:
A 964: if (nsta .le. 0) goto 999
A 965:
A 966: makmdf = makmdf - 1
A 967: if (nsta .gt. maxsta) goto 999
A 968:
A 969: c--- load the station list into the appropriate arrays
A 970:
A 971: makmdf = makmdf - 1
A 972: start = 1024
A 973: if (lw(idfile, start, nsta, ids) .lt. 0) goto 999
A 974: if (lw(idfile, 1 * nsta + start, nsta, st) .lt. 0) goto 999
A 975: if (lw(idfile, 2 * nsta + start, nsta, ele) .lt. 0) goto 999
A 976:
A 977: c--- build the row headers
A 978:
A 979: record(1) = julday
A 980: record(2) = 0
A 981: finc = 0
A 982: do 10 i = 1, nr
A 983:
A 984: c--- if i = 6 that means that we are starting to build the
A 985: c--- row headers for the 12Z model run
A 986:
A 987: if (i .eq. 6) then
A 988: record(2) = 120000
A 989: finc = 0
A 990: endif
A 991:
A 992: c--- increment the forecast time
A 993:
A 994: ok = mcinchr(record(1),record(2),finc,record(3),record(4))
A 995: c--- write to the row header
A 997: A 998: ck = mdo(mdfile, i, 0, nrkeys, rklocs, record)
A 999: A 1000: 10 continue
A 1001: A 1002: c--- build the column headers
A 1003: A 1004: do 100 i = 1, nsta
A 1005: record(1) = ids(i)
A 1006: record(2) = st(i)
A 1007: record(3) = ele(i)
A 1008: ck = mdo(mdfile, 0, i, nckeys, cklocs, record)
A 1009: 100 continue
A 1010: A 1011: makmdf = 1
A 1012: 999 continue
A 1013: return
A 1014: end
A 1015: A 1016: c $ dectraj - decodes a line of the trajectory forecast
A 1017: A 1018: c $ integer function dectraj(line,cid,lat,lon,
A 1019: pre,level,t,td)
A 1020: c $ input:
A 1021: A 1022: c $ line c(*)- line to be decoded
A 1023: A 1024: c $ output:
A 1025: c $ cid c(*)- station id
A 1026: c $ lat i(*)- array of decoded latitudes
A 1027: c $ lon i(*)- array of decoded longitudes
A 1028: c $ pre i(*)- array of pressures
A 1029: c $ t dp - final temperature
A 1030: c $ td dp - final dewpoint
A 1031: c $ return values:
A 1032: A 1033: 0 - success, no station id on this line
A 1034: A 1035: 1 - success, station id on this line
A 1036: c--- numper - number of periods in the for aaaa000pp
A 1037: integer numper
A 1038: A 1039: parameter (numper = 4)
A 1040: A 1041: c--- line - line to decode
A 1042: c--- cid - station id decoded (if found)
A 1043: c--- level - level decoded (SFC, 850, 700)
A 1044: A 1045: character*4 line
A 1046: character*4 cid
A 1047: c--- level
A 1048: A 1049: c---fltval - temporary floating point variable
A 1050: c---t - temperature decoded from report
A 1051: c---td - dew point decoded from report
A 1052: A 1053: double precision fltval
A 1054: double precision t
A 1055: double precision td
A 1056: A 1057: c--- ctemp - temporary character string
A 1058: character*12 ctemp
A 1059: A 1060: c--- lat - array of decoded latitudes
A 1061: c--- lon - array of decoded longitudes
A 1062: c--- pre - array of decoded pressures
A 1063: A 1064: integer lat(*)
A 1065: integer lon(*)
A 1066: integer pre(*)
A 1066: c--- latcol1 - column number where latitudes begin
A 1067: c--- loncol - column number where longitudes begin
A 1068: c--- pcol - column number where pressures begin
A 1069:
A 1070: integer latcol(numer)
A 1071: integer loncol(numer)
A 1072: integer pcol(numer)
A 1073:
A 1074: integer ok
A 1075: integer strsta
A 1076: integer intval
A 1077: integer intsta
A 1078: integer fitsta
A 1079: integer moextrln
A 1080: integer i
A 1081:
A 1082: data latcol/10,20,30,40/
A 1083: data loncol/13,23,33,43/
A 1084: data pcol /16,26,36,46/
A 1085:
A 1086: dectrj = 0
A 1087:
A 1088: c--- extract the station id if it exists from column 2-4
A 1089:
A 1090: ok = moextrln(line,2,4,cid,strsta,intval,intsta,fitsta)
A 1091:
A 1092: c--- if the line contains a station id, return 1
A 1093:
A 1094: if (cid .ne. ' ')dectrj = 1
A 1095:
A 1096: c--- loop 10 scans through the line decoding the first 4
A 1097: c--- forecast periods
A 1098:
A 1099: do 10 i = 1 , numer
A 1100: c--- extract the latitude
A 1101:
A 1102: ok = moextrln(line,latcol(i),latcol(i)+2,ctemp,strsta,
A 1103: & intval,intsta,fitval,fitsta)
A 1104: lat(i) = intval * 1000
A 1105:
A 1106:
A 1107: c--- extract the longitude, we may have to convert it
A 1108: c--- if the value extracted is less than 600 it implies
A 1109: c--- that the value is actually proceeded by 100.
A 1110: c--- example:
A 1111: c--- if the value is 765 it implies a value of 76.5
A 1112: c--- if the value is 116 that implies a value of 111.6
A 1113:
A 1114: ok = moextrln(line,loncol(i),loncol(i)+2,ctemp,strsta,
A 1115: & intval,intsta,fitval,fitsta)
A 1116:
A 1117: if (intval .ge. 600) then
A 1118: lon(i) = intval * 1000
A 1119: else
A 1120: lon(i) = 1000000 + (intval * 1000)
A 1121: endif
A 1122:
A 1123: c--- extract the pressure
A 1124:
A 1125: ok = moextrln(line,pcol(i),pcol(i)+2,ctemp,strsta,
A 1126: & intval,intsta,fitval,fitsta)
A 1127:
A 1128: c--- if the value extracted is less that 100 assume that
A 1129: c--- the data is actually for a level above 1000mb
A 1130:
A 1131: if (intval .lt. 100)intval = intval + 1000
A 1132: pre(i) = intval
A 1133:
A 1134: continue
A 1135:
A 1136: c--- now decode the final state of the parcel. put the
A 1137: c--- results in position 5 of the lat, lon, and pre arrays.
A 1138: c--- extract the ending parcel level
A 1140: A 1141: ok = mceextrln(line,6,8,ctemp,strsta,
A 1142: & intval,intsta,fltval,fltsta)
A 1143: A 1144: level = ctemp
A 1145: A 1146: c--- if this is not the SFC level put this value in pre(5)
A 1147: c--- otherwise read the pressure value from SFC location
A 1148: A 1149: if (ctemp .eq. 'SFC') then
A 1150: A 1151: ok = mceextrln(line,50,52,ctemp,strsta,intval,intsta,
A 1152: & fltval,fltsta)
A 1153: A 1154: endif
A 1155: A 1156: pre(5) = intval
A 1157: A 1158: c--- extract the temperature
A 1159: A 1160: ok = mceextrln(line,53,57,ctemp,strsta,intval,intsta,
A 1161: & t,fltsta)
A 1162: A 1163: c--- extract the dewpoint
A 1164: A 1165: ok = mceextrln(line,59,63,ctemp,strsta,intval,intsta,
A 1166: & td,fltsta)
A 1167: A 1168: 999 continue
A 1169: A 1170: end

Source code used to run m0trjdec from the command line

B 0: c $ trjdec - demonstration command for trajectory decoder
B 1: c $ this command runs the trajectory forecast decoder m0trjdec
B 2: c $ to file information in an MD file format and RA format
B 3: c $ for quick access.
B 4: c
B 5: subroutine main0
B 6: implicit integer (a-z)
B 7: parameter (maxlin = 200 , idxsiz = 16)
B 8: integer idxblk(idxsiz) , flags(8)
B 9: character*4 header , clit , src , origin , cindex , wmo
B 10: character*80 cline , file , cblk(maxlin)
B 11: character*12 cflags(8)
B 12: double precision dtme
B 13: data cindex/'FO'/
B 14: data header/'FOUS'/
B 15: data minprd/50/ , maxprd/57/
B 16: data dtme/13.90/
B 17: B 18: call getday(today)
B 19: call gettim(now)
B 20: yyddd = ikwp('DAY',1,today)
B 21: time = ikwp('TIME',1,now)
B 22: B 23: c--- set the appropriate values for the cflags array
B 24: B 25: cflags(3) = 'NEWPO50.IDM'
B 26: cflags(5) = 'PO14DEC.IDT'
B 27: cflags(6) = 'PO50.RAP'
B 28: flags(2) = 1
B 29: flags(3) = 12
B 30: flags(4) = 100
B 31: flags(5) = 10
B 32: flags(6) = 600
B 33: flag = 0
B 34:  found  = 0
B 35:  100 continue
B 36:
B 37:  c---
B 38:    get the next available index block
B 39:  gidx = mcgetidx(yyddd,time,dtime,1,cindex,idxblk,flag)
B 40:
B 41:  c---
B 42:    if there was an error or we have no more data
B 43:  if (gidx .le. 0)goto 999
B 44:
B 45:  src = clit(idxblk(1))
B 46:  wmo = clit(idxblk(5))
B 47:  origin = clit(idxblk(7))
B 48:
B 49:    write(cline,FMT='(a4,1x,i5,1x,i9,1x,i6,1x,a4,1x,i12,1x,a4)')
B 50:  & src,idxblk(2),idxblk(3),idxblk(4),wmo,idxblk(6),origin
B 51:    call ddest(cline,0)
B 52:
B 53:  c---
B 54:    make certain the wmo header products match
B 55:  if (wmo .ne. header)goto 100
B 56:  if (idxblk(6) .lt. minprd .or.
B 57:    & idxblk(6) .gt. maxprd)goto 100
B 58:
B 59:    found = found + 1
B 60:
B 61:  c---
B 62:    if you make it to here, you know you have the data you
B 63:    are interested in, so load the text and decode it
B 64:
B 65:    call m0spiln(idxblk(3) , src , yyddd , file) ptr)
B 66:    ok = mc1ddat(file , ptr , idxblk(2),maxlin*80,80,cblk)
B 67:
B 68:    do 200 i = 1 , nlines
B 69:      call ddest(cblk(i),0)
B 70:  200 continue
B 71:    call ddest('----------',0)
B 72:
B 73:    ok = m0trjdec(cblk(1),cblk(2),nlines-1,yyddd,time,'' ,
B 74:    & flags,cflags)
B 75:
B 76:    if (found .lt. (maxprd-minprd+1))goto 100
B 77:
B 78:  999 continue
B 79:    call edest('done',0)
B 80:  return
B 81: end

Source code used to run m0trjdec from a data monitor,
dmlocal.pgm
into the data structure for decinfo.dat and
cannot be altered.
mxiflg - the maximum number of integer decoding flags
that can be set for a particular decoder. this
value is wired into the data structure for
decinfo.dat and cannot be altered.
mxcfIg - the maximum number of character decoding flags
that can be set for a particular decoder. this
value is wired into the data structure for
decinfo.dat and cannot be altered.
maxwmo - the maximum number of wmo headers
that can be set for a particular decoder. this
value is wired into the data structure for
decinfo.dat and cannot be altered.
maxorg - the maximum number of station origins
that can be set for a particular decoder.
byprln - number of bytes per line

If you build your own decoding task using this one as a
template, the only things that have to be changed are
'numdec' - the number of decoders this task processes
'task' - the data monitor name that is running
decnam' - the decoders to be processed for this task
And the actual decoder calls themselves

PARAMETER (idxsize = 16)
PARAMETER (maxlin = 1000 , numdec = 1 , byprln = 80)
PARAMETER (maxidx = 8 , mxiflg = 16 , mxcflg = 8 , maxwmo = 20)
PARAMETER (maxorg = 128)
PARAMETER (maxbyt = maxlin * byprln)
CHARACTER*12 task , decnam(numdec) , cuser
PARAMETER (task = 'DMLLOCAL')
c wmo is a list of the maxwmo character portions
c c numdec decoders
c CHARACTER*4 wmo(maxwmo,numdec)
c c orglst is a list of the maxorg station origins
c c that are acceptable for each of the
c c numdec decoders
CHARACTER*4 orglst(maxorg,numdec)
c c namidx is a list of the maxidx indices that are
c c to be processed for the numdec decoders
CHARACTER*4 namidx(maxidx,numdec)
c c descrp is a list of the decoder titles
CHARACTER*80 descrp(numdec)
c c cflags is a list of the mxcflg character
c c string flags used by the numdec decoders
CHARACTER*12 cflags(mxcflg,numdec)
c c cidxfl is a list of the mxcflg index file names
used by the numdec decoders
CHARACTER*12 cidxfl(maxidx,numdec)
c c cbklk is the character array that will contain
the data to be decoded
C 89: CHARACTER*80 cbilk(maxlin)
C 90: c --- begin ptr and lasptr are the beginning and ending
C 91: c --- index pointer locations processed for each of
C 92: c --- the maxidx indices for each numdec decoders.
C 93: c --- begin ptr is initialized to 0 if the task is just
C 94: c --- starting or if the day has changed. lasptr is
C 95: c --- initialized to -1.
C 96: c --- calfig is the flag indicating initial
C 97: c --- processing procedures, see m0nxtidx
C 98: c ---
C 99: C 100: INTEGER begptr(maxidx,numdec), lasptr(maxidx,numdec)
C 101: INTEGER calfig(maxidx,numdec)
C 102: c --- flags is a list of mxiflg integer flags used
C 103: c --- by the numdec decoders
C 104: c ---
C 105: INTEGER flags(mxiflg,numdec)
C 106: c --- numwmo is the number of wmo headers defined
C 107: c --- for the numdec decoders
C 108: c ---
C 109: INTEGER numwmo(numdec)
C 110: c --- numorg is the number of station origins
C 111: c --- defined for the numdec decoders
C 112: c ---
C 113: INTEGER numorg(numdec)
C 114: c --- numidx is the number of indices that are
C 115: c --- actually defined for the numdec decoders
C 116: c ---
C 117: INTEGER numidx(numdec)
C 118: c ---
C 119: c --- decsta is the decoder status flag for the
C 120: c --- numdec decoders
C 121: c ---
C 122: INTEGER decsta(numdec)
C 123: c ---
C 124: c --- minprd and maxprd are the minimum and
C 125: c --- maximum product numbers that are to be
C 126: c --- decoded for the numdec decoders
C 127: c ---
C 128: INTEGER minprd(numwmo,numdec)
C 129: c ---
C 130: c ---
C 131: INTEGER maxprd(numwmo,numdec)
C 132: c ---
C 133: c ---
C 134: c ---
C 135: c ---
C 136: c ---
C 137: c ---
C 138: c ---
C 139: c ---
C 140: c ---
C 141: c ---
C 142: c ---
C 143: c ---
C 144: INTEGER idxblk(idxsiz)
C 145: c ---
C 146: CHARACTER*12 cfu, ched, cspool, cbull1, cbull2, cdnum,
C 147: c ---
C 148: c ---
C 149: CHARACTER*80 cline
C 150: c ---
C 151: CHARACTER*4 curt, cwm, corgin, clit, afos, afostn,
C 152: c ---
C 153: c ---
C 154: LOGICAL difday
C 155: DATA s1not/0/
C 156: c ---
C 157: DATA decnam /*TRJDEC*/
C 158: c ---
C 159: c ---
C 160: c --- setting a debugging keyword


```
C 161:  dbgflg = ikwp('DEB',1,0)
C 162:  c--- if mxstrt is set to -1 then this decoding
C 164:  c--- task will run indifinitely. otherwise
C 165:  c--- it will end normally after the routine
C 166:  c--- sleep has been called mxstrt times
C 167:  mxstrt = ikwp('RESTART',1,500)
C 169:  call getday(yyddd)
C 171:  cerror = task
C 173:  len = min0(nchars(cerror,ib,ie),8)
C 174:  cerror = cerror(1:len)//'.'ERR'
C 175:  c--- make certain the correct unix login is used
C 177:  ok = m0oprchk(cuser,1)
C 179:  if (ok .eq. 0) goto 2000
C 180:  numoff = 0
C 182:  do 100 dec = 1, numdec
C 184:  c--- get the decoder configuration information
C 186:  active = m0dcinfo(FDCINF, task, decnam(dec),
C 188:  & tsksta, decsta(dec),
C 189:  & maxidx, numidx(dec), namidx(1,dec),
C 190:  & mxcf0g, cflags(1,dec),
C 191:  & mxiflg, flags(1,dec),
C 192:  & maxwmo, numwmo(dec), wmo(1,dec),
C 193:  & minprd(1,dec), maxprd(1,dec),
C 194:  & maxorg, numorg(dec), orglst(1,dec), descrp(dec))
C 195:  if (tsksta .lt. 0) then
C 197:  call edest('Data Monitor '//task//'is inactive '//cerror,
C 198:  & tsksta)
C 199:  goto 2000
C 200:  endif
C 201:  c--- output any errors generated by m0dcinfo
C 203:  if (active .eq. -1) then
C 205:  call edest('Unable to find '//task//' and '//decnam(dec)
C 206:  & '// in '//FDCINF,0)
C 207:  goto 2000
C 208:  endif
C 209:  c--- if no indices are defined, the decoder does
C 211:  c--- not know where to look for data, so
C 212:  c--- exit out
C 213:  if (numidx(dec) .le. 0) then
C 215:  call edest('No indices defined for '//decnam(dec),0)
C 216:  goto 2000
C 217:  endif
C 218:  c--- if the decoder is currently labeled inactive
C 220:  c--- notify the starter and exit out if no decoders
C 221:  c--- are labeled active
C 222:  if (decsta(dec) .lt. 0) then
C 224:  c--- if (decsta(dec) .lt. 0) then
C 225:  c--- numoff = numoff + 1
C 226:  call edest(decnam(dec)//' is labeled inactive: '
C 227:  & '//decnrp(dec),0)
C 228:  if (numoff .eq. numdec) goto 2000
C 229:  elseif (decsta(dec) .eq. 0) then
C 231:  call edest('No decoder found in '//task//' called '}
```

McIDAS Developer/Operator Training

October 24, 1995

Developing Local Decoders in McIDAS-XCD

8-53
& goto 2000

else
call sdest(task//' Starting: '//descrip(dec),0)
endif
c--- initialize the things that are index/decoder
c--- dependent
do 110 idx = 1, numidx(dec)
callflg(idx,dec) = -2
begptx(idx,dec) = 0
lasptr(idx,dec) = -1
init(idx,dec) = 0
chead = namidx(idx,dec)
call m01dxnam(chead,yyddd,' ,cidxfl(idx,dec))
110 continue

c--- add startup information to the error

c--- message file
call m01xadc(' ,flags(3,dec),'DECO',bullbd,BBSIZE,kstat)
cbull1 = cfu(bullbd(BBBPTR))
cbull2 = cfu(bullbd(BBLPTR))
cdnum = cfu(flags(3,dec))
cline = 'Started '/(scheme(dec)'/cbull1(1:10)
& //cbull2(1:10)/cdnum
call mcerror(1,cerror,cline)

100 continue

c--- difday - flag set to true if the data-day
c--- currently being processed is different from

c--- the system day. this was required to insure
c--- that all the text from one spool file is

c--- processed before going on to the next spool

c--- file.
c---
difday = .false.
c--- statement #5 is only accessed if no new data

c--- has come in in a while or if the task has been

c--- asleep.
c---
c--- rstart is a counter used to determine when

c--- this task is to restart

c---
rstart = 0

5 continue

call getday(curday)
c

c--- if the system day is different from the data-day

c--- set the difday flag to true
c

c--- if (yyddd .ne. curday)difday = .true.
c

c--- loop 200 scans through each of the defined

c--- decoders
c

do 200 dec = 1, numdec
c

c--- if (mod(dbgsflg,2) .eq. 1) then
c

call gettim(time)
c
c--- ctemp = cfj(time)
c
c--- cline = task//' processing for '//decnam(dec)//'
c
c--- //ctemp(7:12)
ctemp = cfu(yyddd)
cline(49:) = ctemp
ctemp = cfu(curday)
cline(56:) = ctemp
call sdest(cline,0)

endif

if the decoder is labeled as active

if (decsa(dec) .ge. 0) then

loop 300 scans through each of the defined
indices used by each decoder
do 300 idx = 1, numidx(dec)
call gettime(time)

if the task has just started, check to make
 certain that the index lw file exists
(i.e. data of the type you are interested has
been ingested). if not go to the next index file

if (init(idx,dec) .eq. 0) then
lwf = lwfile(cidxfl(idx,dec))
if (lw .eq. 0) goto 300
init(idx,dec) = 1

endif

statement #400 starts the main processing
loop. this section gets the next 4 word
index block, and if it is a new block,
processes the data to completion
continue

next = m0nxtidx(cidxfl(idx,dec),yyddd,time,
      idxbk, cspool.ptr, flags(3,dec), bpl, 
      begptr(idx,dec), lasptr(idx,dec),
      calflg(idx,dec))

if (idxbk(8) .ne. MISS) afos = clit(idxbk(8))
afosn = 
if (idxbk(9) .ne. MISS) afosn = clit(idxbk(9))
aforg = 
if (idxbk(10) .ne. MISS) aforg = clit(idxbk(10))
faa = 0
if (idxbk(16) .ne. MISS) faa = idxbk(16)

if next .lt. 0 goto 300 for and look at the
next index/decoder

if (mod(dbgflg/2,2) .eq. 1) then
write(cline,490)task,cidxfl(idx,dec),
begptr(idx,dec),lasptr(idx,dec),
cspool,nbytes,cwmo,prodct,corgin,next
format(2(al12,1x),2(i16,1x),al12,1x,i16,1x,
a4,1x,12,1x,a4,1x,13)
call sdest(cline,0)
endif

if (next .lt. 0) goto 300
endif

if a new data block was found, reset
the processing counter to 0

stlnot = 0

now we will do a check on
this data block to be sure we really want to
load it.

if specific wmo headers are to be decoded
check to make certain that this might be
a correct block.

if (numwmo(dec) .gt. 0 .or. numorg(dec) .gt. 0) then
endif

compare the wmo header, (ex. 'fou0') and
the product number with the list of
acceptable values

ok = m0hedchk(cwmo, prodct, corgin,
    1, 1, 1,
    1, numwmo(dec), wmo(1,dec),
    minprd(1,dec), maxprd(1,dec),
    numorg(dec), orglist(1,dec)
)

if this is not a correct header go grab the
next index block

if (ok .lt. 0) goto 400
endif

if the index block is ok, load the data block
to cblk and do actual decoding

ok = m0ldatb(cspool, ptr, nbytes, maxbyt, byprln, cblk)

if the data block was successfully loaded,
decode the data

if (ok .gt. 0) then
endif

if (mod(dbgflg/4,2) .eq. 1) then
endif

if (decnam(dec) .eq. 'TRJDEC ') then
endif

decok = m0trjdec(cblk(1), cblk(2), nlines-1,
circuit, yydd, time,
    flags(1,dec), cflags(1,dec))

endif

go back up and see if any more index blocks
can be processed before moving onto the next
index

goto 400
endif
C 449: 300    continue
C 450: endif
C 452: c 200 continue
C 454: c 700 continue
C 456: statement #700 checks the following:
C 457: c 1) if the system day is different from C 458: c the data-day update the data-day to
C 459: c the system day and build the new index C 460: c file name.
C 461: c 2) increment the counter stlnot by 1. if C 462: c stlnot is .gt. 1 that means that monxidx C 463: c has been called twice in a row without C 464: c receiving any new data. if this occurs, C 465: c set stlnot back to zero and go to sleep C 466: c 3) else go back to the top and continue C 467: c processing
C 468: c if (difday .and. stlnot .gt. 0) then
C 469: yyydd = curday
C 470: difday = .false.
C 471: c reinitialize all of the index/decoder
C 472: c dependent information for the new day
C 473: do 710 dec = 1 , numdec
C 474: do 720 idx = 1 , numidx(dec)
C 475: init(idx,dec) = 0
C 476: calfig(idx,dec) = -1
C 477: beginptr(idx,dec) = 0
C 478: lasptr(idx,dec) = -1
C 479: chead = namidx(idx,dec)
C 480: call midxnam(chead, yyydd, circuit, cidxf1(idx,dec))
C 481: call
C 482: 720 continue
C 483: 710 continue
C 484: endif
C 485: stlnot = stlnot + 1
C 486: c if stlnot is .gt. 1 that means that the C 487: cycle of loops 200 and 300 has been through C 488: 2 complete times without finding anything C 489: new to decode so go to sleep
C 490: if (stlnot .gt. 1) then
C 491: stlnot = 0
C 492: goto 1000
C 493: endif
C 494: goto 5
C 495: c statement 1000 puts system to sleep for C 496: about 30 seconds
C 497: c check to see if the system is shutting down
C 498: if (luc(194) .ne. 0) goto 2000
C 499: c check to see if we should end the command
C 521: c--- instead of sleeping
C 522:
C 523:   if (mxstrt .gt. 0 .and. rstart .ge. mxstrt) goto 2000
C 524:
C 525:   rstart = rstart + 1
C 526:
C 527:   if (mod(dbgflg,2) .eq. 1) call sdest(task//' is sleeping',0)
C 528:
C 529:   call sleep(30000)
C 530:
C 531:   c--- after waking up. go back to the top and try
C 532:   c--- processing again.
C 533:
C 534:   goto 5
C 535:2000 continue
C 536:
C 537: call mcerror(i,errno,task//'- Done')
C 538: call edest('Done',0)
C 539: return
C 540: end
Moving to a Distributed System

Presented by
Dee Wade
McIDAS Operations Manager

Session 9
McIDAS Developer/Operator Training
October 23-25, 1995
Table of Contents

Overview .................................................. 9-1
SSEC’s McIDAS system ................................. 9-1
Problems encountered ................................. 9-3
Future goals .............................................. 9-5
Overview

This session will update you on SSEC's status on moving to a distributed system. It will also explain some of the problems SSEC has encountered during our transition and the goals we have set for the near future.

SSEC's McIDAS system

The SSEC McIDAS system currently consists of the following:

- an IBM 4381-T92 mainframe that ingests and serves satellite data from GOES-8, GOES-9 and Meteosat-5, and receives conventional data from the Unix -XCD relay workstation
- several Unix workstations that handle a variety of tasks

The diagram below shows this configuration. The mainframe provides users with applications that have not been ported to Unix or OS/2.
The Unix McIDAS network consists of the following workstations:

<table>
<thead>
<tr>
<th>Workstations</th>
<th>SSEC names</th>
</tr>
</thead>
<tbody>
<tr>
<td>two IBM RISC/6000 3BTs</td>
<td>spock and bones</td>
</tr>
<tr>
<td>one HP 725/75</td>
<td>kirk</td>
</tr>
<tr>
<td>two SUN SPARCstation 10s</td>
<td>wxdata and r2d2</td>
</tr>
<tr>
<td>one IBM RISC/6000 58H</td>
<td>gold</td>
</tr>
</tbody>
</table>

The two 3BTs run McIDAS-XSD, ingesting and serving the POES relay data: GAC, LAC and HRPT.

The HP receives and serves GOES-7 data.

The SUN workstations run McIDAS-XCD. wxdata is the primary -XCD workstation; it also receives and serves GMS data from Australia. r2d2 is a hot spare for -XCD, and is also used for product generation and serving.

The RISC/6000 58H serves in-house users who don’t have the computing power on their desks. It is also the main source for tape output. gold has two 4mm DAT drives, an 8mm Exabyte drive, a 9trk drive, and a 3490E drive. All Unix tape processing is done on this workstation.

We quickly found that we didn’t have enough disk space. All operation’s Unix workstations now have 9 GB disk drives.
Limited floor space

As we began moving towards a distributed system, one of the first problems we encountered was where to put the workstations. Because limited floor space didn’t allow us to add desks or tables, we had to build vertically. Complicating the search for appropriate furniture was the requirement that we handle multiple vendor hardware. Not only did the furniture have to be flexible, since the size and shape of the workstations and monitors varied from vendor to vendor, but the shelves had to support a great deal of weight. We chose Ergotron because of their pricing and flexibility. The picture below shows the Ergotron units.

Each unit holds six workstations: three on the bottom shelf and three on the middle shelf. The top shelf is used for peripherals. So far, this has worked well, but we don’t have six workstations on a unit yet. This will limit the operator’s workspace.
Operator training

Another problem we encountered was the training of our operations staff. The Unix environment is very different from the mainframe. The staff was not familiar with the Unix software or the new hardware. We will continue our training until the McIDAS operators are completely familiar with the new environment.

Increased staff

Operating a distributed Unix environment has forced an increase in the operations staff numbers. We added two people with strong Unix backgrounds to assist the McIDAS operators by writing scripts and programs, and performing some administrative duties. Unix system administrators are also needed to set up and maintain the workstations and network. SSEC's system administrator is responsible for all Unix systems in the Center, not just those in the McIDAS area. It quickly became apparent that McIDAS Operations needed its own system administrator.

Root access

In the McIDAS mainframe environment, the user oper often had power that other users did not. In a Unix environment, the user root has complete control over the system. This is usually reserved for the system administrator. We decided that the operator should not run as root. We use root only when necessary: when setting up accounts, for example. Each site must decide who has root access.

Shared files and peripherals

Multiple workstations must share common files and the use of peripherals. Operators should not be required to maintain a list of user initials and project numbers on each workstation, but every workstation needs access to this data. Each workstation should not require a separate tape drive for backups or saving data.

To solve these problems, we decided to NFS mount shared files, locate all tape drives on one workstation, and connect the workstations with an FDDI interface (100Mb/sec vs. 10Mb/sec of ethernet). We used the same approach with printing.
**Future goals**

**Have the option to turn off the mainframe**

Our goal is to make all mainframe functions available on Unix by the end of 1996. We can then turn off our mainframe if we choose. This means we must have reliable -XSD GVAR and Meteosat ingestors. Both of these satellite types are currently under development. Since SSEC maintains the GOES archive, we must also be able to process the archive data via -XSD. Finally, we must resolve our problems with tape processing, which Unix does not readily address, and the porting of user commands.

**Operate from a consolidated console**

A distributed system means the operator must monitor multiple workstations. Operators need a consolidated console that provides a condensed status of processing on all workstations and sends warning messages when a workstation has trouble. The operator must also be able to get in-depth information about each workstation. We will be working on this console in 1996.

**Produce the Distributed Operations Manual**

We hope to have the first version of the *Distributed Operations Manual* available in the spring of 1996. Because we are still learning to run a distributed system, this will be a living document. If you have suggestions about what should be in this manual, please let me know.

I can be reached at (608) 263-0527 or via e-mail at deew@ssec.wisc.edu
McIDAS-XSD
Operations

Presented by

Dana Davis - McIDAS Operator
Jerrold Robaidek - Operations Programmer

Session 10
McIDAS Developer/Operator Training
October 23-25, 1995
# Table of Contents

Overview .................................................. 10-1  
Terminology ................................................. 10-1  

McIDAS-XSD ingest system ................................. 10-2  
Satellite signals that McIDAS-XSD ingests .............. 10-2  
Satellite signal ingestors under development .......... 10-2  
McIDAS-XSD vs. -MVS ingestors .......................... 10-3  

McIDAS-XSD data flow ...................................... 10-5  
Image data flow ............................................. 10-5  
Control processes and information flow ................. 10-6  

McIDAS-XSD setup ........................................... 10-12  
Initializing schedule files ............................... 10-12  
Modifying your startup environment ..................... 10-13  
Obtaining navigation ...................................... 10-13  
Setting up satellite schedules ........................... 10-14  
Setting up event schedules ............................... 10-15  

Starting and stopping McIDAS-XSD ....................... 10-16  
Geostationary satellite ingest ............................ 10-16  
Polar orbiter ingest ........................................ 10-17  
Stopping McIDAS-XSD ....................................... 10-17  

Miscellaneous utilities ..................................... 10-18  

Troubleshooting ............................................. 10-21
Overview

This training session will provide basic information about McIDAS-XSD (McIDAS-X Satellite Data). After this session, you should be able to:

- start and stop McIDAS-XSD
- recognize and use the control and display windows
- make modifications to your configuration file, xsd.cfg
- recognize aborted processes and restart them
- set up an event schedule

Terminology

The following terms are used throughout this section.

cron          Unix daemon that runs commands at specified times

crontab       Unix command for submitting entries to the cron daemon

daemon        a Unix process that operates continuously and unattended to perform a service

DDS           Domestic Data Service

full resolution buffer  a portion of shared memory where the ingd process stores full resolution data to make it available for other processes

PDL            Processor Data Load

rcp            remote copy protocol

SAS            Satellite-data Acquisition System

sector buffer  the portion of shared memory where the ingd process gets data

shared memory  a block of memory accessible to more than one process

$XSD           Unix environment variable set as the home directory path of McIDAS-XSD
McIDAS-XSD ingest system

The McIDAS-XSD ingest system allows Unix-based workstations to directly receive and process satellite data. With McIDAS-XSD, you can schedule the creation of real-time areas, schedule post processing of those areas, control the Satellite-data Acquisition System (SAS), and monitor system operation.

Satellite signals that McIDAS-XSD ingests

McIDAS-XSD ingests the five types of satellite data below. SSEC currently ingests only GOES-7 and POES-Relay data with McIDAS-XSD.

- GOES AAA
- POES-Relay
- POES-Flyover
- GMS
- DMSP

Satellite signal ingestors under development

- METEOSAT
- GVAR
McIDAS-XSD vs. -MVS ingestors

Besides being designed for Unix-based workstations, McIDAS-XSD has several other differences from the McIDAS-MVS ingestor.

**Ingest process**

On the -MVS ingestor, the ingestor card does most of the work, including averaging, sampling and resolution reduction. McIDAS-XSD has several software processes to accomplish these tasks.

**Operational commands**

Many of the McIDAS-MVS operational commands for generating satellite schedules are also available in McIDAS-XSD. However there are important differences. The satellite scheduler commands in McIDAS-MVS (SSKx and GEOx) are entered at the McIDAS command line. The McIDAS-XSD scheduler commands (polx and geox) are entered at the Unix prompt.

**Independent operation**

McIDAS-XSD does not need an active McIDAS session. The McIDAS-MVS ingestor, however, requires that McIDAS be running.

**Operator intervention**

In McIDAS-XSD, the monitoring of incoming data is an automated process. The same task in McIDAS-MVS requires operator intervention.
McIDAS-XSD data flow

McIDAS-XSD uses several processes when ingesting satellite data. The data used, and the information exchanged by these processes follow many paths. This section examines these processes and the flow of data and information through McIDAS-XSD, as shown in the flow chart on the adjacent page.

Image data flow

Image data from the satellite is received by the antenna and passed to the receiving electronics and PSK demodulator. From the demodulator, it is passed to the SSEC ingessor card, or SAS (Satellite-data Acquisition System).

The SAS is a stand-alone unit that collects satellite data and provides it to another system upon request via an ethernet connection. The SSEC ingest card, along with the software device driver, provides that same data to a RS6000-based ingest system. Because the board is inserted into a Micro-channel slot on the RS6000, an external communication link is not required.

The communication process (cm) receives the data from the SAS or SSEC ingessor card and informs xsdm. xsdm then directs cm where in the sector buffer section of shared memory to put the data. The ingest process (ingd) takes the data from the sector buffer and creates a full resolution buffer section of shared memory. The full resolution buffer contains full resolution image data in a format that enables the product generator to use the data. The product generator (pgd) gets the data from the full resolution buffer and creates products (digital areas) on disk according to the requests in the satellite scheduler.
Control processes and information flow

**xsdm**

The **xsdm** process is the master control for the entire McIDAS-XSD system. This process is started at the Unix prompt by typing **xsdm**. The following functions are performed by **xsdm**:

- reserves the use of the SAS
- manages all program-to-program communication
- allocates ingester resources
- sends antenna positioning information to the SAS for polar HRPT flyover ingests
- starts all other McIDAS-XSD processes

**cm**

The communications process, **cm**, is the first process started by **xsdm**. It establishes communication pathways that allow **xsdm** to pass commands and to receive the satellite signals from the SAS. When the SAS begins sending data, a second **cm** process is started. This **cm** process receives the satellite signal and writes the data to the sector buffer.

Information is also exchanged by **xsdm** and **cm** to control the rest of the McIDAS-XSD system. The SAS receives antenna table information and control information from **cm**, and returns the antenna position and SAS status information. The SAS controls the antenna movements by sending antenna position information to the antenna.
ehd

The event handler process is ehd. It receives messages from other processes and broadcasts the messages throughout the McIDAS-XSD system. The event handler also evaluates user-defined expression and command pair entries from the event scheduler. If an expression evaluates as true, ehd runs the command. The user can create these expression and command pairs with the ehe command.

The ingest process sends messages to the event handler for these events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Message contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>image begin and end time</td>
<td>image date and time, sensor source number and type, scan number, scan date and time, start block, satellite mode</td>
</tr>
<tr>
<td>navigation codicil filed</td>
<td>scan count, slot where the data directory is filed, scan date and time, image date and time, sensor source number and type</td>
</tr>
<tr>
<td>calibration codicil filed</td>
<td>scan count, scan date and time, image date and time, slot of the data directory, bandmap, sensor source number and type</td>
</tr>
<tr>
<td>PDL</td>
<td>satellite mode, scan date and time, image date and time, pdlls, pdllats, pdlbands, pdlspins (if Dwell), pdlnum</td>
</tr>
</tbody>
</table>

ui

The process, ui, is the user interface process for xsdm. This process sends user-entered commands back to xsdm. All processes in McIDAS-XSD use the ui window for error and other text output.

The user interface window is an xterm window with an Xsd: prompt, as shown above. From this prompt, all processes in McIDAS-XSD can be started or stopped. This is also where other ui commands are entered, such as acquire SAS-address or start satellite.
**watchd**

The `watchd` process is known as the watch dog process. Information regarding the status of all other McIDAS-XSD processes (except xsdm) is exchanged between `watchd` and `xsdm`. If a process fails, `watchd` notifies the operator by displaying a yellow popup window on the screen.

Once data is detected, the following processes are started:

- ingd
- pgd
- statusd
- textdisp
- mond
- ant_track
- jd

**ingd**

The ingestor process, known as `ingd`, reads the raw data from the sector buffer, and generates full resolution data. This data is later used by other processes in McIDAS-XSD. Status information is sent by `ingd` to the Bulletin Board shared memory segment.

**pgd**

The product generator process, `pgd`, generates McIDAS areas from the full resolution buffer. It creates products based on user-defined entries. Area products may be single- or multi-banded, or full or reduced resolution imagery. Status information is sent to the Bulletin Board and product generation information to the event handler.

**statusd**

This is the status process that reads the ingestor and product generator status from the Bulletin Board shared memory segment. The status process then provides this information to `textdisp` for display in the status window.
textdisp

This process creates and displays status messages. The McIDAS-XSD Status window, shown below, is generated by the textdisp process. It lists information about the product generator (pgd) and ingest process (ingd). Status information from ingd and pgd are obtained by statusd from the Bulletin Board shared memory segment and passed to textdisp.

mon

The monitor process, mond, creates a window and displays data from the full resolution shared memory buffer. Other image information can also be found in this window such as date, time and element offset.

The McIDAS-XSD Monitor window, shown below, displays a portion of the current image being ingested. If the ingest process flags a bad line of data, a color line (shown as a gray line below) is displayed on the window. The text line at the bottom of the window displays the current scan line, date, time, and the type of data being displayed.

Note: For geostationary satellites, clicking the left mouse button on the McIDAS-XSD monitor window toggles the display between visible and infrared.
ant_track

This process displays the current position of an antenna that is tracking polar orbiting satellites.

jd

The journal process, jd, receives and stores status messages from other McIDAS-XSD processes. It writes all journal messages that it receives to the journal files, which can be listed with the jls command.

The ingestor process sends journal messages when the following events occur:

- an image begins for all ingestor types
- an image ends for all ingestor types
- navigation is filed; GMS and geostationary satellites only
- calibration is filed
- a beta is filed; geostationary satellites only
- a piece of common doc is bad for any ingestor type
- a bad scan line is detected for any ingestor type

The event handler, ehd, sends a journal message when the following events occur:

- ehd starts or shuts down
- ehd receives a message
- ehd updates its list of entries
- ehd receives a message in an incorrect syntax

The product generator process, pgd, sends a journal message each time the following events occur:

- the pgd process starts or stops
- the product ends earlier then scheduled
- the percentage of good data in the product is less than the minimum required to retain the product
- the beginning and end of each product
The monitor process, **mond**, sends a journal message each time it starts a new image.

The journal process writes those messages to journal files in the $OPROOT directory. Within that directory, journal files are broken into subdirectories depending on the process sending the message and the type of ingestor that is running. For example, the disk file $OPROOT/GOES/db/ingd contains journal messages from the ingd process of the GOES ingestor.
McIDAS-XSD setup

The McIDAS-XSD setup consists of the following procedures, each of which is described below.

- initializing the schedule files
- modifying the startup environment
- obtaining navigation
- setting up the satellite schedules and event schedules

Initializing schedule files

Before McIDAS-XSD can ingest data, the software has to be told what type of data it will ingest, and when to ingest it. This information is kept in the satellite scheduler files, which must be initialized with the files command before they can be edited or added to using the geoe and pole commands. The files command initializes the satellite scheduler data files by deleting their contents. Run this command once at installation to initialize these files. Use this command with caution! It deletes all entries and windows. It is usually run only at installation, when major ingestor/scheduler software changes occur, or if the satellite scheduler files are accidentally deleted or become corrupt.

Another command that must be run at installation is popdbs. This command creates a binary version of the satellite scheduler database file (SATDBS). You must run popdbs before requests for satellite data can be made with the geoe and pole commands. You must also run it after any changes are made to the scheduler database file. These changes are usually provided in software upgrades.

sites.dat

The scheduler command polx will not work with earth coordinates until your antenna location is defined in the sites.dat file in the $XSD/lib/pfdata directory. Below is an example of a sites.dat file:

```
SSEC 43.07 89.41
sentry -1000.0 -1000.0
```

The first line contains the name of the site and the location of the site’s antenna. The second line denotes the end of the file.
Modifying your startup environment

The attributes of the processes started by xsdm are determined by the McIDAS-XSD system defaults. To change these attributes, you must create or modify the text file xsd.cfg, which is located in the $XSD directory. The xsd.cfg contains a set of user defined arguments that xsdm uses when it starts the other McIDAS-XSD processes. An example of an xsd.cfg file is shown below:

```
# 
MOND_Poes=\"-s $DATA -k $OUT -w 640 -h 500 -I 30 -b 2 -f\"
MOND_Poes=\"-s $DATA -k $OUT -w 640 -h 350 -I 160\"
MOND_Goes=\"-s $DATA -k $OUT -w 1000 -h 350 -I 25\"
```

This file overrides the default for mond and starts mond with a window size as defined by the w and h flags. The symbol # denotes comment lines and is ignored when the file is read.

Obtaining navigation

In addition to the commands described above, you must obtain navigation for the satellites.

**POES**

POES navigation is sent via DDS under the header TBUS. The TIRDEC decoder in the miscellaneous data monitor dmmsc (provided in McIDAS-XCD), decodes POES navigation and files it into the file SYSNAV1 on a McIDAS-XCD machine. SYSNAV1 is then transferred via rcp to the McIDAS-XSD machine and put into the directory $XSD/lib/mcdata.

**DMSP**

DMSP navigation is sent as part of the DMSP data stream. The DMSP ingester decodes and files the navigation into the proper system navigation files.

**GMS**

GMS navigation is sent as part of the GMS data stream. The ingester puts navigation into the GMS DOC areas. The event handler then runs an entry containing the McIDAS-X utility GMSGRD, which reads the navigation from the GMS DOC areas and files landmarks into the appropriate system navigation file. The McIDAS-X utility, NVUP, which is run from the crontab, reads those landmarks and updates the navigation.
**GOES**

GOES navigation is sent as part of the GOES data stream. The GOES ingestor decodes and files the navigation into the proper system navigation files.

**PRED**

PRED navigation predictions are done for all types of data to ensure that the current day's navigation always exists in $XSD/lib/mcdata/SYSNAV1.

---

**Setting up satellite schedules**

The satellite scheduler commands determine ingest windows and products. The command syntax, keywords, flags, and results are very similar to those for the McIDAS-MVS commands SSKx and GEOx. The scheduler commands for polar and geostationary satellites are defined below.

<table>
<thead>
<tr>
<th>Commands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pole</td>
<td>creates satellite scheduler windows, entries and named sectors for polar orbiting satellites.</td>
</tr>
<tr>
<td>polu</td>
<td>modifies satellite scheduler windows, entries and named sectors for polar orbiting satellites</td>
</tr>
<tr>
<td>poll</td>
<td>lists satellite scheduler windows for polar orbiting satellites</td>
</tr>
<tr>
<td>geoe</td>
<td>creates satellite scheduler windows, entries and named sectors for geostationary satellites.</td>
</tr>
<tr>
<td>geou</td>
<td>modifies satellite scheduler windows, entries and named sectors for geostationary satellites</td>
</tr>
<tr>
<td>geol</td>
<td>lists satellite scheduler windows for geostationary satellites</td>
</tr>
</tbody>
</table>
Setting up event schedules

ehe

Use ehe to create, edit, activate, or deactivate an event handler entry. Each entry contains a Boolean expression and a Unix command. As the event handler receives messages from other McIDAS-XSD processes, it checks the message fields against entry expressions. If the expression is true, the event handler runs the specified command. Utilities used by ehe can be found in Miscellaneous utilities later in this section. The format is as follows:

$progs{field} operator matchstring

For example:  $ingd {event} eq "image end"

Use command ehls to list the event scheduler; use command ehrm to delete entries from the event scheduler.

show_conf and ant_table

show_conf and ant_table resolve antenna conflicts for polar orbiting satellites and send the antenna table to the xsdm, which in turn uses that table to control the antenna movements. ant_table and show_conf read all the polar orbiter windows in the satellite schedule and determine which subentries apply to the time interval given, and if any of those subentries conflict with each other. If no conflicts exist, the antenna table is sent to xsdm. If conflicts do exist, use show_conf to resolve the conflicts via a series of graphical user interfaces (GUIs). If you use ant_table with the -d flag, it will resolve any conflicts by choosing the longest of the conflicting orbits; ant_table will not send the antenna table to xsdm if conflicts exist.
Starting and stopping McIDAS-XSD

This section describes how to start and stop geostationary satellite ingests and polar orbiter ingests.

Geostationary satellite ingest

When all the necessary files are initialized and the satellite scheduler files are updated, you can begin to ingest data. Use the steps below to start McIDAS-XSD.

1. From an xterm window, logon to the account where McIDAS-XSD is installed.

2. From the Unix prompt, start the master control process.

   Type: xsdm

   The xsdm process starts cm, watchd and ui. The Unix prompt in the xterm window is replaced with the xsdm-User Interface prompt, Xsd:.

3. From the Xsd: prompt, reserve the Satellite-data Acquisition System (SAS).

   Type: acquire address

   where address is the IP address of the SAS to be used. Some sites allow only numeric IP addresses. Contact your Unix system administrator for more information.

4. From the Xsd: prompt, specify the type of satellite data to ingest.

   Type: start satellite

   where satellite is the type of satellite data to ingest, GMS or GOES, for example. Enter the satellite name in uppercase. This entry tells xsdm to allow data flow from the SAS when data is available. Currently, McIDAS-XSD can process only one type of geostationary satellite data at a time.
When xsdm detects data, it starts ingd, jd, pgd, statusd, and mond using the satellite type's default flags or user defined flags from the configuration file xsd.cfg; see the *Modifying your startup environment* section above.

To enter McIDAS-XSD satellite schedule or event handler entries, or Unix crontab entries, use another xterm window logged on to the account where McIDAS-XSD is installed.

**Polar orbiter ingest**

Starting a polar orbiter ingest is similar to the geostationary ingest. However, before you start McIDAS-XSD and before any scheduling is done, verify that current navigation parameters exist; otherwise, the antenna tracking will not function properly. See the *McIDAS-XSD Users Guide* (9/95) for a complete description.

**Stopping McIDAS-XSD**

To stop a geostationary or polar orbiter ingest system, you must stop the master control process. From the Xsd prompt,

Type: `quit xsdm`

This command stops all McIDAS-XSD processes and releases the SAS.
Miscellaneous utilities

This section describes some of the miscellaneous Unix and McIDAS utilities.

jls

jls lists journal records according to specified sort parameters. The search keys available for the ehd, ingd, and pgd processes can be found in the McIDAS-XSD Users Guide. jls is a Unix utility and must be issued from a Unix prompt.

mcommand

mcommand creates an environment so that McIDAS commands can run without an active McIDAS-X session. mcommand requires a connection to an X Window server on which you have privileges. Use the Unix export command to set your display environment variable, so the server’s output is sent to your display. For example, export DISPLAY=unix:0

mcommand is ideal for scheduling McIDAS commands to run via the event handler or UNIX crontab entries. When scheduling commands, you may want redirect text output to a file, which can serve as a log of errors and successful commands run by mcommand. The command format is:

mcommand COMMAND

where COMMAND is the complete McIDAS-X command, in uppercase.

To enter multiple McIDAS commands on one command line, separate the commands with a backslash (\) and a semicolon (;). Use this format:

mcommand FIRST CMD;SECOND CMD

There are many McIDAS-X utilities available for post processing of ingested areas. On McIDAS-MVS, these utilities were run from McIDAS with the event scheduler. McIDAS-XSD runs these commands via mcommand. These commands are set up with the event scheduler command ehe. The information below describes the McIDAS-XSD utilities likely to be run.
**DMSPCAL**

You must run the DMSPCAL command on an SSMI area before you can display the area. DMSPCAL does not change the data values, but it does record calibration information to the CAL portion of the area prefix. You can enter DMSPCAL manually, but the event handler is normally used to automate this process. Use the command format below for scheduling DMSPCAL. Enter the entire command on one line, as shown.

```
ehe -e 'Spgd{event} eq "product end" & & $pgd{sig} eq "ssmi"' -c 'mccommand DMSPCAL pgd{area}'
```

**GMSGRD and GMSNVUP**

GMSGRD and GMSNVUP update the navigation for GMS images. GMSGRD files grid locations (landmarks) from the GMS DOC areas into the specified navigation file. GMSNVUP is a macro that uses NVUP to upgrade the current navigation with the landmarks created by GMSGRD. GMSNVUP also uses PRED to predict navigation for the next day.

**IMGFLIP**

IMGFLIP flips polar orbiter images from left to right or top to bottom, so the northwest corner of the image is in the upper-left. You can use this command with all DMSP sensors and the POES AVHRR sensor. You cannot flip POES HIRS and MSU images.

**ORB PLOT**

ORB PLOT displays the orbital tracks of polar orbiting satellites. It gets the navigation information defining the orbital tracks from two different sources. If you use the AREA keyword, the navigation stored with the specified area generates the tracks. Otherwise, the navigation parameters are taken from a system navigation file, specified with the NAF keyword.

By default, ORBPLOT plots orbital tracks over the displayed image or map. If the frame contains no navigation, a Mercator map of the world is displayed before plotting the orbital tracks. Use the MAP keyword to specify a different map, or use MAP=DEF and the LAT and LON keywords to define a custom map.


**PRED**

PRED files predicted navigation based on a master navigation entry. It is usually run from the crontab once per day.

**TIRCAL**

TIRCAL calibrates POES AVHRR images. It is scheduled to run at the end of each image via the event scheduler. Use the command format below for scheduling TIRCAL:

```
ehe -e 'spgd(event) eq "product end" && spgd(bandmap) ne "<6>" && spgd(satmode) eq "POES"' -c 'mccommand TIRCAL $pgd(area)'
```

**TIPTI**

TIPTI creates HIRS and MSU products from the raw data in the POES TIP areas.
Troubleshooting

If you encounter any of the problems below, use the suggested solutions to fix them. Although restarting McIDAS-XSD may fix a problem, do a restart only after investigating all other avenues.

**pgd has stopped**

Problem: The watchdog process (watchdog) reports that **pgd** has stopped. After restarting **pgd**, it stops again.

Possible solution: The file system that **pgd** writes to could be full. Check the file systems from the Unix prompt with **df**. Delete any unneeded files, such as old areas or debug files.

**Navigation is lost**

Problem: Loss of navigation

Possible solutions: McIDAS-XSD may have been down for more than 24 hours and PRED was not run. You may need to run PRED. Or, TBUS may be old. Check the TBUS time on a McIDAS-XCD workstation. If TBUS is more than two days old, contact your TBUS supplier.

**Ingestion or processes are hung**

Problem: McIDAS-XSD is not ingesting or processes seem to be hung. After trying a restart, McIDAS-XSD still does not come up.

Possible solution: Duplicate processes may have been started. Multiple processes are usually a result of **xsdm** being stopped by means other than **quit xsdm** at the Xsd: prompt. Stop/delete the extra processes from the Unix prompt. Restart McIDAS-XSD.
McIDAS-XCD Operations

Presented by
Chad Johnson
Operations Programmer

Session 11
McIDAS Developer/Operator Training
October 23-25, 1995
# Table of Contents

Overview .................................................. 11-1
Terminology ............................................. 11-1
McIDAS-XCD ingesters and data monitors .......... 11-2

Bouncing ingesters and data monitors .............. 11-3

Updating station changes ............................ 11-4
  Station dictionaries ................................ 11-5
  Site-specific ID tables ............................. 11-5

Setting up real-time data locations ................. 11-6

Customizing the GRIB decoder ....................... 11-9

Status display ......................................... 11-12

The DATARECV program .............................. 11-14

Allocating disk space ................................ 11-15

Deleting text, MD and grid files ................... 11-16

Archiving data ......................................... 11-17

Recovering archived data ............................ 11-19

Troubleshooting .......................................... 11-20
Overview

This training session will provide McIDAS operators with the information they need to set up and maintain McIDAS-XCD. It describes the conventional data ingesters and monitors, and the status display. It also explains how to perform the following procedures:

- bounce ingestors and data monitors
- update station tables
- set up real-time data locations
- customize the GRIB decoder
- allocate disk space
- delete data files
- archive data
- troubleshoot

Terminology

The terms below are used throughout this section.

- **circuit file**: file containing text data blocks received from a text ingester; the file has the extension .XCD
- **data block**: text data containing a WMO header
- **data monitor**: a McIDAS program that periodically checks newly ingested data to determine if a specific decoder should be called
- **DDS**: Domestic Data Service
- **decoder**: software that parses data from one format into a common format for use by another process such as a plotter or lister
- **GRIB**: GRIded Binary message format accepted by the World Meteorological Organization for the distribution of gridded data
**HRS**  High Resolution Service

**IDS**  International Data Service

**index file**  a file, written by ingesters, that contains pointers to data blocks in circuit files; index files have the extension .IDX

**ingestor**  process that listens to data received by a communications port and reformats the information for further processing

**NGM**  National Meteorological Center Nested Grid Model

**NWS**  National Weather Service

**PPS**  Public Products Service

**STARTXCD**  mother task of the entire McIDAS-XCD ingestor/decoder package

**status display**  X Windows application that displays the current state of the McIDAS-XCD ingestor/decoder system

---

**McIDAS-XCD ingestors and data monitors**

The McIDAS-XCD ingestors read data from a communications port and reformat that data for later processing. It has two types of ingestors:

- **text**
- **binary**

The text ingestor reads ASCII data from the communications port and files that data in a circuit-specific file. For each block of text data written, additional information is written to an index file specific to the WMO category to which that block belongs. The binary ingestor reads data from a circuit and files that data into a circular spool file for later processing.

The -XCD data monitors traverse through the index files or binary spool files as data is filed by the ingesters. When new data arrives that a decoder is interested in, that block of data is passed to the decoder for processing. The decoder then parses the data and converts it into McIDAS file format.

The McIDAS-XCD ingestors and data monitors are started by the XCD master process, STARTXCD. Never start the ingesters or data monitors from the McIDAS command line.
Bouncing ingestors and data monitors

You will rarely need to bounce a McIDAS-XCD ingester or data monitor. Ingestors run continuously. The only time you may need to bounce them is for communications port changes, such as baud rate or port connection changes.

Data monitors are designed to periodically restart with no intervention so that any changes made since the last restart will become effective. For example, if you make a station change to MASTERID.DAT, such as adding a RAOB station with the IDU command, it becomes effective the next time the decoder restarts.

You must bounce a data monitor manually if either of these events occur:

- a decoder within a data monitor is activated or inactivated; for example, if you activate the TIROS Navigation decoder and want the change to take effect immediately
- a configuration file is modified and must take effect immediately; for example, if you activate the station ID monitoring system or move real-time MD files

STARTXCD continuously monitors the state of -XCD. To bounce an -XCD ingester or data monitor, simply stop the version of the process currently running just as you would stop any McIDAS command. STARTXCD samples the system every 30 seconds and will restart any process it finds no longer running.
Updating station changes

With the modernization currently taking place in the weather service, stations are constantly being moved, commissioned, decommissioned, and augmented.

When a station’s status is changed, the NWS typically sends a message under WMO header NOUS41 containing relevant information about the station change. Because this method is not 100% reliable, you should also check the NWS gopher server, which was recently established to help disseminate modernization information. The address is: gopher.cominfo.nws.noaa.gov (140.90.5.206).

McIDAS-XCD can monitor some incoming stations for you, since most McIDAS-XCD decoders contain software to do this. If you keep statistics on stations that you are expecting to receive and those that are currently not in your database, you can inactivate stations that are no longer reporting.

To activate station monitoring, you must set the IDMONEFLAG value in the configuration file for that decoder. The configuration files reside in ~oper/mcidas/data. The IDMONEFLAG value is a number between 0 and 3, as described below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDMONEFLAG=0</td>
<td>no station monitoring; this is the default when McIDAS-XCD is installed</td>
</tr>
<tr>
<td>IDMONEFLAG=1</td>
<td>monitors new stations</td>
</tr>
<tr>
<td>IDMONEFLAG=2</td>
<td>monitors old stations</td>
</tr>
<tr>
<td>IDMONEFLAG=3</td>
<td>monitors both old and new stations</td>
</tr>
</tbody>
</table>

Use the McIDAS-XCD command, IDMONE, to display the information collected with station monitoring.
Station dictionaries

You can find station ID information in these two files: IDMSL and MASTERID.DAT

IDMSL is the Master Station Catalog, which is maintained by Tinker Air Force Base, and is distributed with McIDAS-X. IDMSL is used by McIDAS commands that convert station IDs to lat/lon groups, such as MSL, PC L, and DF.

MASTERID.DAT is used by the McIDAS-XCD decoders. It contains both geographical information about stations and the types of observations each station reports. MASTERID.DAT resides in ~mcidas/data and is replaced with each McIDAS-XCD upgrade. The site administrator uses the McIDAS-XCD utility, IDU ADD/EDIT, to keep MASTERID.DAT up-to-date. IDU is specifically designed to manipulate the individual entries in MASTERID.DAT.

MASTERID.DAT and IDMSL are updated monthly and are available to you via the MUG BBS (Bulletin Board System). If you don’t want to replace your version of the file MASTERID.DAT, a list of changes made each month and cumulative since the last McIDAS-XCD upgrade is provided. Instructions for upgrading your MASTERID.DAT and IDMSL are also available on the BBS.

If you know of stations whose status is incorrect in MASTERID.DAT, contact the McIDAS Help Desk and we will integrate those changes in the next release if they are beneficial to the entire McIDAS community. Until the stations are updated in the core system, you can add them yourself. Keep a copy of the IDU commands that you run locally in a batch file. Then if you need to reload the core station list, you can run a batch command to reimplement the station changes.

Site-specific ID tables

If you have numerous, specific station changes at your site, you should maintain a batch file of IDU commands with these changes. When you upgrade to a new version of MASTERID.DAT, you can simply run this batch file to add your site-specific stations.

You can also maintain this station table separately from the core MASTERID.DAT in LOCALID.DAT. To do this, write to an alternate file with the FILE= keyword of the McIDAS-XCD IDU command. Then modify the MASTERFILE= keyword in the configuration file for your local decoder to use this station ID table.
Setting up real-time data locations

When you install McIDAS-XCD, real-time data is filed in these locations:

<table>
<thead>
<tr>
<th>Data type</th>
<th>MD/Grid files</th>
<th>Files</th>
<th>Config. file</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAO/METAR</td>
<td>1 - 10</td>
<td>SAOMETAR.RAP</td>
<td>ISFCDEC.CFG</td>
</tr>
<tr>
<td>RAOB</td>
<td>11 - 30</td>
<td>RAOB.RAP</td>
<td>IRABDEC.CFG</td>
</tr>
<tr>
<td>Ship/ Buoy</td>
<td>31 - 40</td>
<td></td>
<td>ISHPDEC.CFG</td>
</tr>
<tr>
<td>FOUS14</td>
<td>41 - 50</td>
<td></td>
<td>FO14DEC.CFG</td>
</tr>
<tr>
<td>SYNOPTIC</td>
<td>51 - 60</td>
<td>SYNOPTIC.RAP</td>
<td>SYNDEC.CFG</td>
</tr>
<tr>
<td>PIREP/AIREP</td>
<td>61 - 70</td>
<td></td>
<td>PIRDEC.CFG</td>
</tr>
<tr>
<td>FTs/ TAFs</td>
<td></td>
<td>TERMFCST.RAP</td>
<td>TERDEC.CFG</td>
</tr>
<tr>
<td>Watch Box</td>
<td></td>
<td>WXWATCH.DAT</td>
<td>WBXDEC.CFG</td>
</tr>
<tr>
<td>TIROS Nav</td>
<td></td>
<td>SYSNAV1</td>
<td>TIRDEC.CFG</td>
</tr>
<tr>
<td>Real-time Grids</td>
<td>5001 - 5300</td>
<td></td>
<td>RTMODELS.CFG</td>
</tr>
</tbody>
</table>

MD filing

Do not alter the MD file locations at this time. Most of the applications that display real-time conventional data call a subroutine that has not yet been modified to allow more flexibility.

Grid filing

The miscellaneous grid file group (5001-5010) stores output from models that send minimal data; for example, the Wind Wave Forecast Model (WWFM). The grid files are divided by model: ETA, NGM, MRF, MAPS. The file RTMODELS.CFG, located in ~mcidas/data, determines the schemes used for filing gridded data. The three basic filing schemes are described below.

- All grids from a model are filed in one grid file regardless of model run and valid forecast times. Although this scheme is the simplest to understand, the search time for specific grids may be excessive if many products are generated by the model.

- Everything from the model is stored in one grid file per model run time per day. For example, all the NGM data from the 12 Z model run is stored in grid files 6001-6010, based on Julian day.

- Grids are filed based on model, model run time and model forecast validation time. For example, all the MRF data from the 12 Z model run with forecast times of 00hr through 24hr are stored in grid files 5071-5080, based on Julian day. This scheme is the most difficult for a user trying to find a specific grid file, but search times for individual grids are shorter.
Software that uses the appropriate API for locating real-time grid files will make whichever method you choose irrelevant to the user.

A complete description of RTMODELS.CFG can be found in Appendix D of the McIDAS-XCD Installation and Users Guide.

**Example**

Below is an example of an RTMODELS.CFG file and a description of how the various models are filed.

```plaintext
# Positional parameter descriptions
# position  description
# 1  grid filing format
# 0  - everything from the model is stored in one grid file per model run time
# 1  - grids are filed based on model run time and valid forecast time
# 2  - all the grids from a model run are filed in the same grid file regardless of run time or forecast time
# 3  - same as 1 except no grids are assumed beyond the max forecast time (parameter 5)
# 2  first grid file in the range for this model
# 3  interval between model runs (hhmmss)
# 4  which forecast period interval to use to separate forecast grids
# 5  maximum forecast time, after which all grids are stored in the same grid file

SCRATCH=5001
MAPS= 0 1001 30000
MRP= 1 1101 120000 240000 480000
NGM= 2 1201

In the above example, any model not specified in RTMODELS.CFG is filed in the range specified by the SCRATCH= keyword. For example, ETA grids are filed in a grid file between 5001 and 5010 based on the Julian day of the model run time.

MAPS grids are filed only by model run time beginning with grid file 1001. See the table below.

<table>
<thead>
<tr>
<th>Grid file</th>
<th>Run time</th>
<th>Forecast range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001-1010</td>
<td>00 Z</td>
<td>all forecast times</td>
</tr>
<tr>
<td>1011-1020</td>
<td>03 Z</td>
<td>all forecast times</td>
</tr>
<tr>
<td>1021-1030</td>
<td>06 Z</td>
<td>all forecast times</td>
</tr>
<tr>
<td>1031-1040</td>
<td>09 Z</td>
<td>all forecast times</td>
</tr>
<tr>
<td>1041-1050</td>
<td>12 Z</td>
<td>all forecast times</td>
</tr>
<tr>
<td>1051-1060</td>
<td>15 Z</td>
<td>all forecast times</td>
</tr>
<tr>
<td>1061-1070</td>
<td>18 Z</td>
<td>all forecast times</td>
</tr>
<tr>
<td>1071-1080</td>
<td>21 Z</td>
<td>all forecast times</td>
</tr>
</tbody>
</table>
```
MRF grids are filed based on model run time and model verification time. In this example, the forecast period interval separating the forecast grids is 24 hours, and the maximum forecast time is 48 hours. This means that forecast grids in 24-hour intervals are filed in one grid file. All grids with a validation time greater than 48 hours are filed in another grid file. The interval between model run times determines how many groups of these model verification time grid files are created. In this example, the interval between model run times is 12. This means that two of these groups will be created. See the table below.

<table>
<thead>
<tr>
<th>Grid file</th>
<th>Run time</th>
<th>Forecast range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101-1010</td>
<td>00 Z</td>
<td>00hr &lt;= Forecast Time &lt;= 24hr</td>
</tr>
<tr>
<td>1111-1120</td>
<td>00 Z</td>
<td>24hr &lt; Forecast Time &lt;= 48hr</td>
</tr>
<tr>
<td>1121-1130</td>
<td>00 Z</td>
<td>&gt;48hr Forecast Time</td>
</tr>
<tr>
<td>1131-1140</td>
<td>12 Z</td>
<td>00hr &lt;= Forecast Time &lt;= 24hr</td>
</tr>
<tr>
<td>1141-1150</td>
<td>12 Z</td>
<td>24hr &lt; Forecast Time &lt;= 48hr</td>
</tr>
<tr>
<td>1151-1160</td>
<td>12 Z</td>
<td>&gt;48hr Forecast Time</td>
</tr>
</tbody>
</table>

In the RTMODELS.CFG example above, all NGM grids are filed in one range of grid files beginning with grid file 1201. Although this scheme is the easiest to understand, it results in large search times and is very difficult to locate grids of interest.
Customizing the GRIB decoder

McIDAS-XCD version 1.1 contains the first release of the GRIB decoder. You can configure this decoder to discard grids that are not of interest to you. This is advantageous because certain models are sent in multiple projections and there is no need to store this duplicate information. It can also limit the number of grids filed if disk space is a concern.

You can configure the GRIB decoder to discard individual grids based on the following:

- model name
- model run time
- forecast time
- projection
- level
- parameter

This information is stored in the file NOGRIB.CFG in the directory ~oper/mcidas/data.

Below is the format of the entries in the configuration file.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>where:</td>
<td>model</td>
<td>model run time minimum</td>
<td>model run time maximum</td>
<td>model forecast validation hour minimum</td>
<td>model forecast validation hour maximum</td>
<td>grid projection minimum</td>
<td>grid projection maximum</td>
<td>grid level minimum</td>
<td>grid level maximum</td>
<td>grid parameter minimum</td>
</tr>
</tbody>
</table>

The entries placed in NOGRIB.CFG are based on values sent in the Product Definition Section (PDS) of the GRIB message. The information in the PDS includes the model that generated the grid, the forecast period this grid covers, and the parameter type and units of the grid. The values used as parameters in NOGRIB.CFG are the same values sent in the PDS.
The first value to an entry in NOGRIB.CFG is the model number, which is the only required value. The values sent are as follows:

<table>
<thead>
<tr>
<th>PDS number</th>
<th>Model</th>
<th>McIDAS name</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>Nested Grid Model</td>
<td>NGM</td>
</tr>
<tr>
<td>64</td>
<td>Regional Optimal Interpolation</td>
<td>ROI</td>
</tr>
<tr>
<td>77</td>
<td>Spectral Model, Aviation Run</td>
<td>MRF</td>
</tr>
<tr>
<td>78</td>
<td>Medium Range Forecast Model</td>
<td>MRF</td>
</tr>
<tr>
<td>83</td>
<td>80 km ETA</td>
<td>ETA</td>
</tr>
<tr>
<td>84</td>
<td>40 km ETA</td>
<td>ETA</td>
</tr>
<tr>
<td>85</td>
<td>30 km ETA</td>
<td>ETA</td>
</tr>
<tr>
<td>86</td>
<td>Mesoscale Atmos. Prediction Sys</td>
<td>MAPS</td>
</tr>
</tbody>
</table>

For example, the entry to discard all output from the MAPS model will look like this:

```
86 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1
```

The second and third values are the model run times to discard. If you are not interested in the MAPS model runs from 03 Z through 09 Z, your entry will look like this:

```
86 | 3  | 9  | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1
```

The fourth and fifth values are the forecast valid times to discard. If you aren’t interested in the MRF forecast times beyond 48 hours, your entries should look like this:

```
77 | -1 | -1 | 49 | 99 | -1 | -1 | -1 | -1 | -1 | -1
78 | -1 | -1 | 49 | 99 | -1 | -1 | -1 | -1 | -1 | -1
```

The sixth and seventh values relate to the projections in which the models are sent. Some models are sent in more than one projection. It is usually redundant to store more than one projection of the same data. The default configuration for the GRIB decoder when McIDAS-XCD is installed is to discard duplicate projections of the NGM, ROI and MRF.

Below is a table of common projections, their associated models, and the PDS values to enter to discard them.

<table>
<thead>
<tr>
<th>Projection</th>
<th>Description</th>
<th>Models</th>
<th>PDS values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mercator</td>
<td>2.5x5.0 global</td>
<td>MRF</td>
<td>21 - 24</td>
</tr>
<tr>
<td>Mercator</td>
<td>5.0x5.0 global</td>
<td>MRF</td>
<td>25 - 26</td>
</tr>
<tr>
<td>Mercator</td>
<td>1.25x1.25 global thinned</td>
<td>MRF</td>
<td>37 - 44</td>
</tr>
<tr>
<td>Mercator</td>
<td>1.25x2.5 North America</td>
<td>NGM, ROI</td>
<td>50</td>
</tr>
<tr>
<td>Polar St.</td>
<td>North America</td>
<td>ETA,ROI,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NGM, MRF</td>
<td>211</td>
</tr>
</tbody>
</table>
For example, to discard all the 1.25x1.25 Mercator of the MRF, the entry will look like this:

```
77 | -1| -1| -1| -1| 21| 24| -1| -1| -1| -1
```

Use parameters eight and nine to discard particular levels. For example, to discard all data at 900 mb, and levels 350 through 150 mb for the MAPS model, your entries in NOGRIB.CFG will look like this:

```
86 | -1| -1| -1| -1| -1| -1| 900| 900| -1| -1
86 | -1| -1| -1| -1| -1| -1| 150| 350| -1| -1
```

The final two values on the NOGRIB.CFG configuration line are the parameters to discard. Below is a table of the most common values transmitted. You can find a complete list in the file gbtbpd001.2v2 in ~oper/mcidas/data.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>McIDAS name</th>
<th>PDS value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure</td>
<td>P</td>
<td>1</td>
</tr>
<tr>
<td>Pressure reduced to MSL</td>
<td>P</td>
<td>2</td>
</tr>
<tr>
<td>Geopotential Height</td>
<td>Z</td>
<td>7</td>
</tr>
<tr>
<td>Temperature</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>u-component wind</td>
<td>U</td>
<td>33</td>
</tr>
<tr>
<td>v-component wind</td>
<td>V</td>
<td>34</td>
</tr>
<tr>
<td>Relative Humidity</td>
<td>RH</td>
<td>52</td>
</tr>
</tbody>
</table>

To discard all relative humidity grids from the ETA model, your entry will look like this:

```
83 | -1| -1| -1| -1| -1| -1| -1| -1| -1| -1| 52| 52
```

A complete description of NOGRIB.CFG and the discarding of grids is described in Appendix D of the McIDAS-XCD Installation and Users Guide.
Status display

The status display (statdisp) monitors each decoder and ingester. The program statdisp reads from a file and displays the output as shown below.

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Time</th>
<th>Begastr</th>
<th>Lastpr</th>
<th>Grid</th>
<th>Grid</th>
<th>Row</th>
<th>Col</th>
<th>Text</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>146DEC</td>
<td>174448</td>
<td>288592</td>
<td>288592</td>
<td>8</td>
<td>53</td>
<td>1154</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 RGBDEC</td>
<td>174445</td>
<td>59680</td>
<td>59680</td>
<td>30</td>
<td>5</td>
<td>4474</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 SYNDEC</td>
<td>174449</td>
<td>189584</td>
<td>189584</td>
<td>58</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 SHPDEC</td>
<td>174449</td>
<td>189584</td>
<td>189584</td>
<td>38</td>
<td>18</td>
<td>253</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 F14DEC</td>
<td>174436</td>
<td>14672</td>
<td>14672</td>
<td>48</td>
<td>28</td>
<td>238</td>
<td></td>
<td>Hatch 1014</td>
<td>F1014</td>
</tr>
<tr>
<td>6 IMBDEC</td>
<td>174436</td>
<td>4968</td>
<td>4968</td>
<td>468</td>
<td>19</td>
<td>133</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 PRDEC</td>
<td>174436</td>
<td>17856</td>
<td>17856</td>
<td>68</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 TIRDEC</td>
<td>174436</td>
<td>46336</td>
<td>46336</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 TIRDEC</td>
<td>174436</td>
<td>1632</td>
<td>1632</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 GRIB</td>
<td>173311</td>
<td>5158</td>
<td>141</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>HHPE96 KJBC 051220</td>
<td></td>
</tr>
</tbody>
</table>

The default status display file is DECOSTAT.DAT, which resides in ~oper/mcidas/data. If you want your local decoders to write to their own status display, set the environment variable XCD_disp_file. Then statdisp will use that file to determine what to update. For example, if your local decoders write to their own status display file structure called ~local/mcidas/data/LOCSTAT.DAT, you can force statdisp to read that file by typing the following from the Unix command line:

```
export XCD_disp_file=~local/mcidas/data/LOCSTAT.DAT
statdisp &
```

The contents of the file can only be changed by the ingestors and decoders. The ingestors change the contents of the file any time data is received from a communications port. For the text ingestors, this occurs after a complete data block is received. For the binary ingestors, this occurs after it reads from the port 20 times. The decoders update DECOSTAT.DAT any time the .IDX file that they are indexing through is updated.

Every five seconds, statdisp checks the contents of DECOSTAT.DAT. If the status for any decoder or ingester has not changed in five minutes, the color of the output turns red. If a line for a decoder turns red, it means that no data for the WMO product identifier that the decoder is interested in was filed in the last five minutes. Except for the GRIB decoder, it is a rare occurrence for a decoder line on the status display to turn red.
An ingester line turning red means data was not received from the communications port within the last five minutes. DDS, IDS and PPS almost never change to the warning color. If they do, all of them will probably change, indicating a fundamental problem with either the source or your hardware. The ingester for HRS often changes colors between 10 Z and 12 Z, and again between 22 Z and 00 Z. These are usually quiet periods when no gridded data is sent.

To view the status display on a workstation other than the one processing conventional data, you can set the DISPLAY environment variable to the appropriate X-server, and then start statdisp by entering the following lines from the Unix command line.

```bash
export DISPLAY=outfield.ssec.wisc.edu:0
statdisp &
```

Alternatively, you can use the -display command line option:

```bash
statdisp -display outfield.ssec.wisc.edu:0 &
```
The DATARECV program

You can use the McIDAS-XCD program DATARECV to graphically display which stations are expected to be decoded for a data type, and which stations were decoded the previous hour. A sample display is shown below.

Although you can’t discern the color of the dots below, the white dots on your display indicate the stations with their decoder turned on, and that data was received for those stations. Red dots indicate stations with their decoder turned on, but data was not filed by that decoder for those stations.
Allocating disk space

Below is a list of all dynamic files used in McIDAS-XCD and the approximate disk space they consume per day, in MB.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Format</th>
<th>Disk space per day</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD+</td>
<td>TEXT</td>
<td>75</td>
</tr>
<tr>
<td>ID</td>
<td>TEXT</td>
<td>25</td>
</tr>
<tr>
<td>HRS</td>
<td>SPOOL</td>
<td>16</td>
</tr>
<tr>
<td>SFC</td>
<td>MD</td>
<td>20</td>
</tr>
<tr>
<td>IRAB/IRSG</td>
<td>MD</td>
<td>6</td>
</tr>
<tr>
<td>ISHP</td>
<td>MD</td>
<td>3</td>
</tr>
<tr>
<td>FO14</td>
<td>MD</td>
<td>2</td>
</tr>
<tr>
<td>SYN</td>
<td>MD</td>
<td>7</td>
</tr>
<tr>
<td>PIRP</td>
<td>MD</td>
<td>6</td>
</tr>
<tr>
<td>ETA</td>
<td>GRID</td>
<td>52</td>
</tr>
<tr>
<td>NGM*</td>
<td>GRID</td>
<td>50</td>
</tr>
<tr>
<td>MRF*</td>
<td>GRID</td>
<td>350</td>
</tr>
<tr>
<td>MAPS</td>
<td>GRID</td>
<td>140</td>
</tr>
<tr>
<td>Other grids</td>
<td>GRID</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>751 MB</strong></td>
</tr>
</tbody>
</table>

* if you save all the projections
Deleting text, MD and grid files

Up to 751 MB per day is a considerable amount of data and could fill your file system to capacity within a matter of days. These two McIDAS programs can help you remove old data files:

- DELWXT removes text and index files
- QRTMDG removes MD or grid files

They are designed to run from the McIDAS scheduler to delete files older than a specified number of days. For example, to keep four days worth of data online and remove all grid, MD, and text data older than that, enter the following three lines in the McIDAS schedule.

```
#Y 00:05:00 999999 00100:00:00 *DELWXT 4
#Y 00:05:00 999999 00100:00:00 *QRTMDG MD mdl mdn 4
#Y 00:05:00 999999 00100:00:00 *QRTMDG GRID grid1 gridn 4
```
Archiving data

The most complete way to archive data is to save the text (*.XCD) and index (*.IDX) files, and the MD and grid files.

**Raw text**

Use the steps below to archive raw text.

1. Collect the necessary files into an archive file. The resulting file will be approximately 100 Mb for the Family of Services text circuits.

```
tar -cvf TEXT93017.tar IDXALIAS.DAT ??93017.IDX ??930170.XCD
```

2. Compress the file. The resulting .Z file will be about 19 Mb.

```
compress TEXT93017.tar
```

3. Write the compressed tar file to tape, replacing tape-device with the unix file name of your tape drive.

```
tar -cvf tape-device TEXT93017.tar.Z
```

**Real-time MD files**

Use the steps below to archive real-time MD files.

1. Collect the necessary files into an archive file. The resulting file will be approximately 45 Mb for the standard McIDAS-XCD real-time files.

```
tar -cvf MD93017.tar MDXX00?7
```

2. Compress the file. The resulting .Z file will be about 2.5 MB.

```
compress MD93107.tar
```

3. Write the compressed tar file to tape, replacing tape-device with the unix file name of your tape drive.

```
tar -cvf tape-device MD93017.tar.Z
```
Real-time grid files

Use the steps below to archive real-time grid files.

1. Collect the necessary files into an archive file. The resulting file will be approximately 400 Mb for the standard McIDAS-XCD real-time files.

   tar -cvf GRID93017.tar GRID5[0-3]???

2. Compress the file. The resulting .Z file will be about 200 Mb.

   compress GRID93107.tar

3. Write the compressed tar file to tape, replacing tape-device with the unix file name of your tape drive.

   tar -cvf tape-device GRID93017.tar.Z
Recovering archived data

Use the steps below to recover archived data files.

1. Change the directory to an appropriate directory in which to recover the data.

2. Extract the archived grid files.
   
   `zcat GRID93017.tar.Z | tar -xvf -`

   This command uncompresses the file and extracts the tar file.

3. Extract the archived text files
   
   `zcat TEXT93017.tar.Z | tar -xvf -`

4. Extract the archived MD files
   
   `zcat MD93017.tar.Z | tar -xvf -`
Troubleshooting

Below are some typical problems you may encounter and their solutions.

No real-time data

**Symptoms:** The user reports no real-time data, or the status display is red for the ingestors.

**Problem:**
- The file system is full.
- The antenna has an obstruction or there is a problem with the antenna hardware.
- The source provider is experiencing a problem.

**Solution:**
- Check the status of the file system on the workstation with the Unix command `df`. If the file system is full, clean the file system. You can use the McIDAS-XCD programs QRTMDG and DELWXT to delete older text, MD files and grid files. Do not delete any files for the current day.
- Check for an obstruction in the antenna and verify that all receiving hardware is working properly.
- Contact your source provider to see if they are having a problem with the broadcast.

No grid data

**Symptom:** The GRIB decoder is not filing grids.

**Problem:** The decoder can’t find RTMODELS.CFG, which contains information about the grid files to search.

**Solution:** The file RTMODELS.CFG should reside in `~mcidas/data` when McIDAS-XCD is installed correctly. Either the decoder can’t reach the file or it is missing. If it’s missing, either recreate the file or copy a new version from `~mcidas/xcd1.1/data/RTMODELS.CFG`, if your site is using the default configuration for real-time grid file locations.
**Garbled or missing data**

**Symptom:** Text data is missing or text output is garbled. If decoding grids, grids are missing.

**Problem:** More than one ingestor is trying to read the same circuit, resulting in garbled or missing data. Or, there could be an obstruction in the receiving antenna.

**Solution:** Check the process status of the system to see how many ingestext and ingebin processes are running. There should be only one ingestext process running for each text circuit, and one ingebin process running for each binary circuit. If this isn’t the case, stop all McIDAS-XCD processes that are running.

Do a ps to get the PID of all the -XCD processes; for example: `ps -u | grep oper`. Use the kill `-9 PID` command to stop the -XCD processes in the following order.

```
startxcd.mx
all ingestext.mx
all ingebin.mx
all dmx*.mx
```

Then restart -XCD with the McIDAS command STARTXCD.

If the above process doesn’t work, check for an obstruction in the receiving antenna.

**TIROS data lacks navigation**

**Symptom:** Users report that TIROS data does not contain navigation.

**Problem:** The McIDAS-XCD decoder TIRDEC may not be running or is not able to write the SYSNAV1 file.

**Solution:** Verify that the TIRDEC decoder is turned on and the decoder configuration file is correct. If you are ingesting TIROS with McIDAS-XSD, verify that your -XSD ingestor can retrieve the SYSNAV1 file from the -XCD ingestor.
McIDAS Operations on a Distributed System

Presented by
Chad Johnson
Operations Programmer

Session 12
McIDAS Developer/Operator Training
October 23-25, 1995
# Table of Contents

Overview ................................................. 12-1  
Terminology ............................................. 12-1  
The McIDAS distributed system ................. 12-2  

The ADDE server ........................................ 12-3  
ADDE server transaction logging ............... 12-3  
ADDE server security ................................. 12-4  
ADDE server installation ......................... 12-5  
ADDE server configuration ....................... 12-6  
ADDE server startup ................................. 12-11  

McIDAS system support ............................... 12-12  
File system limitations ........................... 12-12  
NFS (Network File System) ....................... 12-13  
Off-line data storage and tape devices ....... 12-14  
Bringing a backup ingester and server online .. 12-15  
Frequently asked questions ....................... 12-16
Overview

This training session will provide McIDAS site administrators and operators with information about McIDAS' role in a distributed data system. It includes the following topics:

- installing and configuring ADDE (Abstract Data Distributed Environment) servers
- providing server security
- choosing dataset names
- following the life cycle of the server as it fulfills a request

This session will also present the following topics, which should be of interest to Unix system administrators:

- file system limitations of the supported platforms
- NFS (Network File System) and where it might be useful
- off-line data storage and tape devices in a distributed system
- how to bring a backup ingestor/server online

Terminology

The terms below are defined as they apply to a McIDAS distributed system. Some of these definitions may not be applicable to other distributed systems.

**client**
workstation receiving and displaying data; initiates the requests

**daemon**
background process that periodically wakes up, checking to see if it should do something

**dataset**
collection of one or more files with a common format

**dataset name**
name used by ADDE to reference a dataset; consists of the group name and a descriptor
**descriptor**  
name used to reference a dataset, separated by a slash (/).

**file system**  
method for cataloging files in a computer system

**group**  
name used to reference a collection of descriptors; used by the client to determine which server to query for data requests

**inetd**  
Unix system daemon that listens to various network ports

**mcserv**  
ADDE server program started by inetd when a network connection is made to a port

**server**  
the machine that stores and supplies data in response to clients’ requests

**transaction logging**  
record keeping done by ADDE servers for each transaction

---

**The McIDAS distributed system**

A distributed system is a computing system in which the storing and serving of data are distributed across multiple workstations. For example, users may be working at a workstation, yet the data they are using may exist on other workstations. Users request the data from the data storage machines. The request is processed and the data is sent to the user.

The McIDAS distributed system is similar, but differs from the generic distributed system in two ways:

- the ingest of data may also be done on the data storage and serving machines
- client and server software have specific data abstractions they support, such as the McIDAS Area file in ADDE

ADDE has been part of the McIDAS-X software package since the June 1995 upgrade.
The ADDE server

This section provides information about the ADDE server, including the following topics:

- transaction logging
- security
- installation and configuration
- startup

ADDE server transaction logging

ADDE servers can provide logging for each request they serve. You can initiate transaction logging for the following reasons:

- accounting
- server usage statistics
- dataset usage statistics

When server transaction logging is turned on, each transaction is logged as a record to the file SERVER.LOG. This file resides in ~mcadde/mcidas/data or the first writable directory in the environment variable MCPATH. Some of the parameters available in each log record are: IP of the server and client, date and time of the transaction, user initials, project number, return code, dataset name, bytes sent and received, CPU time, and error messages.

The SERVER.LOG file is a continuously growing file. To prevent this file from becoming very large, schedule a move of this file to a different file. You can schedule the move via the McIDAS scheduler or the Unix cron scheduler. SERVER.LOG is created again when the next transaction record is written.
ADDE server security

You should consider these two forms of security before setting up your ADDE server:

- file security, which attempts to prevent file deletion or modification by unwanted sources
- access security, which attempts to prevent unauthorized users from accessing your data

File security

The Unix operating system allows multiple users to share the same workstation’s processing power and storage space. In this type of working environment, it is logical to have a system that prevents users from being able to modify and overwrite system and other user’s files. In the Unix operating system, each file is owned by a user and a group, with a set of associated permission flags. The owner can set read, write, and execution privileges for each class: user, group and all other users on the system. It is this that gives the ADDE servers a method of file protection.

To prevent users from writing to an ADDE dataset on your server, you should set up another user account named mcadde. The ADDE server will then be configured to run from this account. The name of the account you choose doesn’t matter, as long as it is different from the account of your data ingestion/storage.

The oper account is typically used for ingest and data storage. The permission of the data files in this account should be set to allow reading for all users, but writing by only oper. This will prevent the server from overwriting your real-time data.

Access security

Access security is available with ADDE by turning on transaction logging, as described on the previous page.

Access security uses the three pieces of information below, which are sent to the server with each data request:

- user initials
- project number
- IP address of the client machine
The list of valid user initials, project numbers, and IP addresses is contained in three text files: SERVER.USR, SERVER.PRJ and SERVER.IP, respectively. These files are created by the operators and should exist in the -mcaida/mcidas/data directory of each server workstation. The format of the file is one line per parameter; comments begin with a *. Below is an example of a SERVER.PRJ file.

* DDE VALID PROJECT NUMBER LIST
  1000
  1001
  1002
  1003
  1004
  1006

Access validation occurs if any of the above files exist. It is not mandatory that all files be present. For example, if only the SERVER.USR file exists, only requests made by users listed in that file are allowed. If all files exist, validation will be checked for all three parameters, and only requests made by valid users with valid project numbers from a valid workstation are allowed. If none of the above files are present, no validation will take place and all requests are allowed.

ADDE server installation

The ADDE server/client software package was included in the standard distribution and installation of the June 1995 McIDAS-X upgrade. To collect transaction logs or use the method of access security mentioned above with the ADDE server, you must perform additional steps when installing McIDAS-X 2.1. You must compile McIDAS-X 2.1 with the DDE_ACCOUNTING compile flag turned on. The steps to accomplish this are provided below.

1. Logon to the workstation as user mcidas.

2. Do not run the installation script mcidas2.1version#.sh.

3. Extract the tar file manually.

   Type: zcat mcidas2.1xx.tar.Z | tar -xvf -

   This will extract the tar file and create the mcidas2.1version# directory. This directory contains the src and data directories.

4. Change the directory to the McIDAS source directory.

   Type: cd SHOME/mcidas2.1xx/src
5. At the shell prompt, export the following two variables.

   Type:  `export McINST_ROOT=$HOME`
   `export McIDAS_ROOT=$HOME`

6. Build McIDAS. This will build the ADDE servers to perform transaction logging.

   Type:  `make INCARGS="-DDDE_ACCOUNTING"`

7. Have all your McIDAS users exit their McIDAS sessions.

8. Install the binaries that will perform transaction logging.

   Type:  `make install`

9. Have all McIDAS users start their McIDAS sessions.

The ADDE servers will now write to the LW file, SERVER.LOG, in the
~mcadde/mcidas/data directory as transactions are logged. This is a binary file and is not viewable by any text editor. However, McIDAS Operations has software to view it. Contact McIDAS Operations if you are interested in any of these utilities.

### ADDE server configuration

ADDE servers will typically run on the same workstation performing the ingest. For example, you may have multiple workstations ingesting data from multiple satellite sources. Each of these workstations must be configured to run an ADDE server.

### Naming the account

When setting up a workstation as an ADDE remote server, you must create a remote server account on the workstation. Consider the following guidelines when deciding on a name for the remote server account.

- Don’t use the `mcidas` account. You can’t run McIDAS-X from the `mcidas` account or other SSEC-supplied software packages that run under it.

- Don’t use any accounts that run McIDAS-XCD and McIDAS-XSD packages which provide conventional or satellite data that you may want to make available to users. For example, don’t use the `oper` account.
• Don’t use accounts that may hold data you want to provide to users via the ADDE server.

• Dedicate this account to the administration of the ADDE remote server. For example, use it only for acquiring and naming data for ADDE clients.

We recommend naming this new account mcadde. This account must be configured as a McIDAS-X user account; for example, setting the appropriate PATH and directories to run McIDAS-X.

**Configuring the workstation**

You must configure the Unix workstation to accept port connections from ADDE clients and to run the server to fulfill these requests. Included with the June '95 McIDAS-X upgrade is the script `mcinetversion#.sh`, which will configure the system daemon `inetd` to do this.

The script `mcinetversion#.sh` configures two system files:

- `/etc/services`
- `/etc/inetd.conf`

The file `/etc/services` specifies the names of services available through the Internet and the protocol of these services, and assigns a port number that each service will connect. Below is an example of the `/etc/services` file.

```plaintext
# UNIX specific services
#
# these are NOT officially assigned
#
nfsd  2049/udp  nfs  # NFS server daemon
mcserv  500/tcp  # McIDAS ADDE port
xcd_rlycl  502/tcp  # XCD core data stream
```

The ADDE service is named mcserv and is defined as a TCP service that connects to port 500.

The `/etc/inetd.conf` files contains a list of programs that `inetd` will start when it receives an Internet request on that port. For example:

```plaintext
mcserv  stream tcp  nowait
/home/mcidas/bin/mcservsh
mcservsh -H /home/mcoper
```

You must configure the ADDE server account so the servers can locate the data files. The ADDE servers use the MCPATH environment variable and the REDIRECT table to locate the McIDAS data files. For example, if you run a McIDAS-XSD ingestor on this same system, you will add the path to the data files in the environment variable MCPATH or add them to the REDIRECT table of the `mcadde` user.
Assigning dataset names

To assign dataset names, perform the steps below.

1. Verify that McIDAS of the server account is able to locate the data files you want to serve. You can do this by adding the data directory to the MCPATH environment variable, or adding it to the REDIRECT table.

2. Choose the dataset name for this dataset.

3. Start a McIDAS session under the ADDE server account and add this dataset using the McIDAS command DSSERVE.

4. Notify your users that you have a new dataset available. Give them the name of the dataset and the host name or IP address of the server workstation.

For example:

Suppose areas 1000 through 1024 are GOES-8 CONUS visible areas and the files reside in /home/oper/mcidas/data on the workstation foo.ssec.wisc.edu. Verify that the McIDAS server account can locate areas 1000 through 1024 in the /home/oper/mcidas/data directory by starting McIDAS in the server account.

**Type:** DMAP AREA

If the paths listed for AREAs 1000 through 1024 are different from /home/oper/mcidas/data, you need to add the directory to the MCPATH environment variable.

In this example, we will assign the dataset name to EAST/CONUSV. Start a McIDAS session in the server account and assign this dataset to a group of areas by entering the command below.

**DSSERVE ADD EAST/CONUSV AREA 1000 1024 "GOES-8 visible CONUS**

The ADDE server is ready to serve AREAs 1000 through 1024 as dataset EAST/CONUSV. Notify your users that GOES-8 visible CONUS data is available from the server foo.ssec.wisc.edu as dataset EAST/CONUSV. They should add this dataset to their client routing tables with the following command.

**Type:** DATALOC ADD EAST "foo.ssec.wisc.edu"
Selecting a dataset name

All ADDE commands use dataset names (in a group(descriptor format) that map to datasets on a server. It is easier for users to locate the data if you follow a logical convention when assigning group and descriptor names. There are three tiers in the hierarchical naming scheme for datasets.

- type
- group
- descriptor

_Type_ is the top tier and can be either image, grid, or point.

_Group_ is the next tier in the naming scheme. A group name can be used only once under each type. Groups are defined by the operator when assigning dataset names to a dataset; for example, a range of McIDAS areas.

_Descriptors_ are the bottom tier in the naming scheme. They further classify or describe the dataset. Descriptors are defined by the operator when assigning dataset names to a dataset.

For example:

<table>
<thead>
<tr>
<th>Group(descriptor)</th>
<th>Data format</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEST/ALL</td>
<td>AREA 101 150</td>
<td>ALL DATA FROM WEST SATELLITE</td>
</tr>
<tr>
<td>WEST/CONUSV</td>
<td>AREA 101 104</td>
<td>CONTINENTAL US; 1KM; VIS; GOES-7</td>
</tr>
<tr>
<td>WEST/DS</td>
<td>AREA 131 136</td>
<td>DWELL SOUNDING; 8KM; GOES-7</td>
</tr>
<tr>
<td>WEST/DSV</td>
<td>AREA 141 142</td>
<td>DWELL SOUNDING; 1KM; GOES-7</td>
</tr>
<tr>
<td>WEST/FDIR</td>
<td>AREA 109 112</td>
<td>FULL DISK; BAND 8; 4KM; GOES-7</td>
</tr>
<tr>
<td>WEST/FDMSI</td>
<td>AREA 121 128</td>
<td>FULL DISK; MSI; 8KM; GOES-7</td>
</tr>
<tr>
<td>WEST/FDV</td>
<td>AREA 105 108</td>
<td>FULL DISK; VISIBLE; 4KM; GOES-7</td>
</tr>
</tbody>
</table>

The table above shows datasets that are being served by a server. All the datasets with the group name WEST reference AREA files ingested from the GOES-7 satellite. There are multiple WEST groups, each with a different descriptor name. Each unique group(descriptor pair represents a defined dataset on the server. Users locate the data using that name.
Defining group names

Below are some hints to help you select group names when defining your dataset names.

Group names are used by the client to locate data. You cannot use a group name that is already used to describe data served from another workstation. You can use the same group name for different types served from the same server.

Group names should describe the data source. For example, if you are serving GOES-8 data, you can use EAST as the group name in all your dataset names referencing data for the east satellite on that server.

Choose group names that do not change often. For example, don’t use satellite names such as GOES8 as group names.

Defining descriptor names

Below are some hints to help you select descriptor names when defining your dataset names.

When assigning descriptor names for gridded data types, the name should describe the time of the grid, or the model run time and verification time if the grid is model output.

Descriptors should describe the geographical coverage of the data. For example, a visible image area that covers a region in the Northern Hemisphere could have a descriptor name such as NHEMV.

Make the names as short as possible. Some users may make aliases for the dataset names, but all users may not. In this case, the users must remember and type in the names you have chosen.
ADDE server startup

A user on a workstation starts an ADDE client process by entering the McIDAS command IMGCOPY. This client process looks in the DATALOC table to determine the IP address of the server machine that is serving the data requested. The client attempts to make a connection to port 500 of the server machine. If the connection is successful, the inetd daemon on the server machine wakes up and determines which service is defined for port 500. inetd then starts the program mcservsh, which examines the request and activates the appropriate data server for finishing the request.

McIDAS Operations

There are several data servers, and each data server knows only about the data type it serves. For instance, the data server adirserv only knows how to subsect and serve McIDAS AREAs; the ggetserv server only understands how to serve McIDAS GRIDs. Below is a list of data servers and the services they provide.

<table>
<thead>
<tr>
<th>Data server</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>adirserv</td>
<td>sends McIDAS area directory listings</td>
</tr>
<tr>
<td>agetserv</td>
<td>sends McIDAS areas from a server</td>
</tr>
<tr>
<td>aputserv</td>
<td>receives McIDAS areas from a client</td>
</tr>
<tr>
<td>gdirserv</td>
<td>sends McIDAS grid directory listings</td>
</tr>
<tr>
<td>ggetserv</td>
<td>sends McIDAS grids from a server</td>
</tr>
<tr>
<td>gputserv</td>
<td>receives McIDAS areas from a client</td>
</tr>
</tbody>
</table>
McIDAS system support

This section discusses topics of interest to site administrators who must set up a McIDAS distributed system. The topics discussed include:

- file system limitations
- NFS (Network File System)
- off-line data storage
- how to bring a backup ingestor/server online
- FAQs (Frequently Asked Questions)

File system limitations

When setting up your McIDAS ingestors and configuring the ADDE server, keep in mind any file system limitations that exist for that particular platform. Below are the file system size limitations for each of the platforms that SSEC supports:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM AIX 3.2.5</td>
<td>2 GB per file system</td>
</tr>
<tr>
<td>HP HPUX 9.0.5</td>
<td>4 GB per disk device</td>
</tr>
<tr>
<td>SGI IRIX 5.3</td>
<td>8 GB per file system</td>
</tr>
<tr>
<td>SUN Solaris 2.4</td>
<td>File system size limit unknown; SSEC has had no problem with up to 9 GB per file system.</td>
</tr>
</tbody>
</table>

HPUX has a limitation of 4 GB per device. If you purchase a 9 GB disk drive for an HP system, you will only be able to utilize 4 GB (less than half) of the total storage capacity of that drive. The remaining 5 GB cannot be partitioned and mounted on additional file systems.
NFS (Network File System)

NFS is a protocol and program set that allows file transfers to occur over a network. It allows one workstation to share file systems with other workstations on the network, rather than keeping separate copies of these files on each workstation. When NFS is used, this shared file system will appear in the local file system as though it really does exist there, although the files actually reside in the file system on a remote workstation.

Some examples of where NFS may be used in the McIDAS distributed system include the following:

- If you’re sharing the -XCD data directory, users can mount the data directory on their system.

- If you’re performing access restrictions, you can share server initial, project, or IP address files; for example, SERVER.USR, SERVER.PRJ, SERVER.IP.

- If you’re performing transaction logging, you can mount the transaction log files from multiple workstations on one workstation.

To use NFS, you must share or export the file system containing the sharable files. When you share a file system, you allow this file system to be remotely mounted on a remote host. You have the option to allow only certain hosts to mount this file system, or you can give read-only permission for connecting hosts. A shared file system can be thought of as being served by an NFS server. To gain use of this shared file system on the external host, users must mount the shared file system on their local workstations. As with most system configurations, you must be root to share or mount remote file systems.

NFS relies on the RPC (Remote Procedure Call) subsystem. If you have trouble sharing or mounting shared file systems, check to make sure RPC is running on your systems.

Type: ps -ef | grep rpc

You should see a line containing the string "rpc.bind" in the output. If you don’t see it in the process listing, RPC isn’t running. There typically is an administrative script to start the RPC subsystem under the /etc directory. The name and location of this script varies among platforms; contact your system administrator for details.
Off-line data storage and tape devices

You can connect many different tape devices to Unix workstations for off-line data storage; for example: DAT-4mm, Exabyte-8mm, 9-Track, 3480, 3490E, 3590. The media formats that your site supports will depend on numerous factors, which may include:

- previous experience
- reliability
- the media on which the majority of your data is stored
- future trends in off-line storage

In a distributed system it is inevitable that the files you would like to write to tape are on a different workstation than the tape devices. Listed below are three possible solutions to this problem.

- Ftp or rcp (remote copy) the files to the system with the tape device.
- NFS mount the source file system to a file system on the workstation containing the tape drive; the data is then accessible via a file system on the machine with the tape drive.
- Execute a remote tar write across the network; for example, to write the files to a remote device,

  Type: `tar -cvf - | rsh host dd of=DEVICE`

To read the files from a remote device type,

  Type: `rsh -n dd if=DEVICE | tar -xvf -`

where `host` is the name of the remote system with the tape device, and `DEVICE` is the name of the device on the remote system you want to write.

Streaming tape devices, such as 8mm and 4mm devices, are very sensitive to having a steady stream of data. When writing across a network to these types of devices, network latency may cause tape write or read failures. To prevent this, you should have 8 Mbits per second bandwidth available on your network when writing to a streaming tape device. Due to the burst nature of network traffic, you can’t assume that the required bandwidth necessary for the device will be present at the moment data is written to tape. Options one and two above are not recommended.
It is not recommended that you use any hardware compression that your drive manufacturer supports. Different hardware vendors may not support the same compression algorithm, resulting in an unreadable tape when transferring it to another tape drive.

**Bringing a backup ingestor and server online**

Your backup system must be configured exactly as your primary system. Items that must be configured on the backup system include the following:

- If this is a hot spare for immediate switchover, verify that you have a second signal feed for the source you intend to back up.

- The McIDAS-XSD or -XCD software must be installed and configured to ingest the same data source as the primary ingester.

- If this is a backup for McIDAS-XCD, configure the decoders just as they are for the primary machine.

- If this is a McIDAS-XSD ingester, the satellite scheduler windows and McIDAS scheduler entries must be entered exactly as they are on the primary system.

- The ADDE server should be configured to use the same dataset names as the real-time system to reference data files. This makes the switchover to the hot spare easiest for the user.

Notify users that you are switching to a different workstation to serve data. Inform them of the following:

- the name of the backup workstation

- any changes to the dataset names

If the dataset names were not changed, the users need only run the McIDAS command **DATALOC** to change their routing table to point to the backup workstation for that group. Users should be able to access the data just as they had before with no change required to any scheduled commands.

If the dataset names were modified for some reason, the users must run the McIDAS command **DATALOC** to add the new group to their routing table. They must also modify their scheduled data retrievals to use the new dataset names.
Frequently asked questions

What is the effect of changing a machine's IP address?

Changing the IP address should have no effect on users, provided they reference that machine by the host name and not the IP address. When an IP address changes on a system, the local name server must be updated. This change will bubble up to the other name servers on the internet system, although this may take a few hours to occur. If users have difficulty accessing this machine, have the system administrator flush the cache of their name server.

If the system is an ADDE server, have all ADDE users who have IP numbers in their dataloc tables, change the IP address of this server with the McIDAS command DATALOC. Also, have all ADDE users run this McIDAS command: DATALOC HOST

ADDE client software does not look up the address for the server when making each request. This command refreshes the internal name/IP lookup table for servers.

How do I know who's accessing the server at any given time?

Use the Unix utility, netstat, to display the network status of the workstation. Appended to the end of this document is a shell script that you can use to display a continually updated status of ADDE server connections.