McIDAS-X Programmer's Manual

Version 2015

McIDAS Programmer's Manual
Version 2015 Table of Contents

For a short history of McIDAS, read the Preface.

McIDAS Programmer's Manual

Preface

McIDAS (Man computer Interactive Data Access System) is a set of tools for obtaining, analyzing, displaying and integrating environmental data. Its design focuses on four attributes:

- the scientific use of time-sequential satellite data
- the merging of diverse databases
- access to real-time databases
- support for the operational and research user community

The Space Science and Engineering Center (SSEC) of the University of Wisconsin-Madison has actively developed McIDAS software since the early 1970s.

McIDAS was originally developed on a Raytheon-440 computer using punch cards, paper tape and magnetic tape. It provided animation, display and analysis of geostationary meteorological satellite data. Later, the code was moved to Harris minicomputers in a distributed network, with two data servers and several applications boxes. Radar, meteorological observations and forecasts were added to the system. McIDAS' ability to provide real-time access to satellite and conventional data made it appealing to the meteorological forecast community.

As the amount of McIDAS code grew, it needed more processing power to ingest data, run larger applications and simultaneously serve more users. Thus, it was centralized onto an IBM mainframe running the MVS operating system.

Prior to McIDAS-MVS, workstations connected to the mainframe were considered dumb with most processing done on the mainframe. The workstation handled only simple controls such as changing frames, activating looping and positioning the cursor. With McIDAS-MVS in place, the first smart workstation was developed on DOS-based personal computers. These machines front-ended display hardware, such as the tower workstation. There were very few local applications, since the necessary multitasking was simulated in the McIDAS software.

The first large-scale port of applications software occurred when the OS/2 operating system was embraced for PCs. This was the beginning of McIDAS-OS2. Efforts to create an environment similar to the mainframe resulted in a smooth port of many applications. However, some changes were needed to accommodate special hardware; for example, VGA displays had only 16 color/gray levels. In addition, drivers were written for each display head: VGA, tower, WIDE WORD and SDA. These drivers had to appear to the applications as the same kind of raster-oriented device, with some varying characteristics such as frame size and number of colors. Communications drivers satisfying all applications were also needed for the common modes: asynchronous, ProNET and TCP/IP.

McIDAS-OS2's success led to the migration of McIDAS to the Unix environment. McIDAS users wanted support for applications on these faster, larger hardware platforms. The first attempt at this migration was taking the OS/2 code and writing specialized routines for the keyboard, mouse, text display, and image/graphics display, plus the system-level interfaces required for communications and disk I/O. The result was McIDAS-X.

Except for the ASK command and the Graphical User Interface, the implementation of McIDAS-X was done using the X Window System. To support McIDAS-X on more platforms and have it adhere to industry standards, the base code was changed significantly to make it more portable and less platform- and vendor-dependent.

Today, McIDAS is a fully distributed, workstation-based system. It generates multicolored composites of conventional and satellite weather data in a variety of displays in two and three dimensions as well as time-lapse sequences of these analyses. Designed to handle large amounts of meteorological imagery and other atmospheric data in a convenient manner, the system provides a vast resource of image-processing and applications programs. McIDAS hardware and software is used worldwide.

Chapter 1

Introduction

This chapter provides an overview of McIDAS-X and explains the use and function of this manual. You'll learn about:

- the current McIDAS suite
- your responsibilities as a McIDAS-X programmer
- this manual's organization, and its symbol, text and screen conventions
- the McIDAS Help Desk
- where to find more information about McIDAS-X

This chapter is organized into these sections:

Overview of McIDAS

McIDAS (Man computer Interactive Data Access System) is a general-purpose collection of tools for acquiring, analyzing and displaying meteorological data. SSEC's McIDAS is a distributed system consisting of Unix workstations that run the ADDE (Abstract Data Distribution Environment) software to receive and process data.

McIDAS is an environment in which data is received, processed, and stored, and then distributed among multiple workstations. For example, users may be working at a McIDAS-X or McIDAS-V workstation, yet the data they are using may reside on other workstations. When users request data from a data storage machine, the request is processed and the data is sent to the user. In the McIDAS distributed system, data can also be received and processed on the same machines that store and serve it.

The McIDAS suite includes the following packages:

- user tools--McIDAS-X and McIDAS-V
- satellite data ingest--SDI
- conventional data collection--McIDAS-XCD

McIDAS-X

McIDAS-X can run as stand-alone packages using real-time data from one or more remote sites or from local sources. The ADDE (Abstract Data Distribution Environment) software is part of the core McIDAS-X package and provides a comprehensive set of application tools for managing, processing, intercomparing and merging data.

Users access and integrate various types of data by displaying images and graphics separately or combined. They can also animate current, past and forecast data displays to monitor evolving weather systems.

McIDAS-V

McIDAS-V is a free, open source, visualization and data analysis software package that is the fifth generation in SSEC's history of sophisticated McIDAS software packages. McIDAS-V displays weather satellite (including hyperspectral) and other geophysical data in 2- and 3-dimensions, and can be used to analyze and manipulate the data with its powerful mathematical functions.

McIDAS-V is built on VisAD and IDV and incorporates the functionality of McIDAS-X and HYDRA for viewing data, developing algorithms, and validating results. It includes tools for research and operational users of observational, model, and satellite data, including data from multi- and hyper-spectral sensors on both current and future satellites. It also allows automatic reprojection of both local and remote data for easy integration of different data sources and formats.

SDI

The SSEC Data Ingestor (SDI) is a hardware and software package that receives and processes satellite data on a PC workstation running a Unix operating system. SDI systems have been developed to receive and processes data from FY-2, GVAR, MTSAT, and POES satellites.

SDI saves satellite data in a format that the McIDAS-X ADDE servers can access and subsect. Any McIDAS-X function can be applied to the data, for example: enhancements, looping, overlaying other data, brightness stretching, and remapping to other projections.

McIDAS-XCD

The McIDAS-X Conventional Data (XCD) software package on Unix-based workstations receives and processes data from the National Weather Service Telecommunications Gateway. The data includes conventional observations, model gridded forecasts, weather summaries, international data, and local data sources.

The data arrives via satellite broadcast. This transmitted data is converted to McIDAS-X formatted files. McIDAS-X users can employ a wide array of McIDAS-X applications to interpret, change, search, and display this data.

McIDAS-X programming philosophy

The McIDAS-X software package includes source code that allows users to tailor applications to fit their needs and to develop new applications. Normally, the SSEC staff undertakes development efforts that impact the basic system, such as file structure changes, new data sources, and new communications methods. New software capabilities developed at SSEC as a result of SSEC's or another McIDAS-X site's efforts, are made available to the entire user community.

This section dicusses the following topics:

Your responsibilities

McIDAS-X continually evolves, due to the needs of scientists, researchers, forecasters, and programmers. Most new applications programs and functions are built on existing programs and functions. As a McIDAS-X programmer, you must maintain system integrity by developing programs and functions that not only satisfy an immediate need, but also promote long-term software development for future programmers and users.

It is also your responsibility to create applications and utility functions that can be used on all McIDAS-supported platforms. This requires attention to language standards and avoiding hardware and operating system specific interfaces. It is also important that you program in abstract ways, eliminating references to specific hardware characteristics whenever possible.

Chapter 2, *Learning the Basics* contains the standards that SSEC programmers use when writing code.

Planning for McIDAS-X upgrades

System enhancements made by SSEC or external users are distributed to all McIDAS-X users via upgrades. Below are some helpful hints for maintaining your locally developed code after a McIDAS-X upgrade.

- McIDAS-X upgrades contain all the code needed for an initial installation of McIDAS-X, as well as aperiodic updates for enhancements or fixes. Thus, you should keep all locally developed code in separate directories and libraries.
- Beware of using McIDAS-X library functions beginning with **m0** (Fortran) or **M0** (C). These functions are considered private to the McIDAS-X core code. The use, capability, and life of these functions are not guaranteed.
- Recompile and relink your local applications after each upgrade is installed. Otherwise, changes in McIDAS-X may result in your code not working. Each upgrade package contains a list of changes for that upgrade.
- As new library functions replace existing ones, old functions will move to a compatibility library. Functions in this library are no longer referenced by McIDAS-X applications and are not tested for upgrades. If you still use these functions, you must treat them as locally developed code.

Using this manual

The *McIDAS-X Programmer's Manual* is designed to be an instructional guide for new McIDAS-X programmers and a reference for experienced McIDAS-X programmers. It assumes that you know the Fortran and C programming languages, and have a basic knowledge of McIDAS-X and your operating system.

This manual provides the information that you need to:

- Write McIDAS-X applications on Unix-based systems
- Understand the McIDAS-X data file formats
- Program at the API (Application Program Interface) level, using the detailed API descriptions from the information provided in the API man pages.

This section discusses the following topics:

How this manual is organized

The *McIDAS-X Programmer's Manual* is divided into seven chapters plus appendices, a glossary and an index. Use the table below as a general guide to help you find the information you need.

If you're interested in	**Turn to**
Knowing what's in this manual and who it's written for, how to get help, and other McIDAS-X documentation that you can reference	Chapter 1, Introduction
The basics about developing applications programs in McIDAS-X, including the types of data available in McIDAS-X and the formats and conventions to use when writing online helps	Chapter 2, Learning the Basics
Setting up the McIDAS-X environment, and compiling, testing and debugging your McIDAS-X code	Chapter 3, Getting Started in McIDAS-X
A description of the McIDAS-X library functions that you'll use to write your applications programs	Chapter 4, McIDAS-X Utilities
Learning how to use McIDAS-X library functions to access data files	Chapter 5, Accessing Data
The file formats for the data files developed for applications running under McIDAS-X	Chapter 6, Format of the Data Files
Writing and maintaining locally developed servers for ADDE	Chapter 7, Writing ADDE Servers
SSEC's guidelines for writing online helps, and information about satellites and data parameters	Appendices: A, B, C, D
Definitions of some terms used in this manual	Glossary
The detailed descriptions for each API function that you need to build applications	Online man pages

Conventions used in this manual

Becoming familiar with the symbol, text and screen conventions in this manual will make the text easier for you to read and understand.

The conventions described below are:

Symbol conventions

1., 2., 3., ... Numbered items indicate a task with two or more sequential steps.

- This symbol indicates a list of items that are not sequential.

 This symbol indicates a reference to other parts of this manual or to companion documentation for additional information.

Text conventions

- The term *function* is used throughout this manual to describe C procedures and functions, and Fortran functions and subroutines. Function names are bolded; for example, **Mccmd** and **mcgget**.
- File names, paths, environment variables and script names are in **`courier bold`**.
- McIDAS-X commands are uppercase and unbolded.
- Actual keyboard entries that you will type appear in **bold**. For example:

  cd $HOME/mcidas/help

You will type keyboard entries exactly as they appear, leaving a space between each term or number in the command line. Always press Enter after typing a keyboard entry.

- When you see an Alt entry, it means you will press two keys simultaneously. For example, **Alt G** means you should hold down the Alt key and press the G key.

Screen conventions

System prompts and responses, and code examples look like this:

```
c --- set up the ADDE transaction ... 100 continue c --- read the data block status =
mcalin(handle, data_buffer) if( status.lt.0 ) then call edest('Data Read failed',0) return
c --- got a line of data else if( status.eq.0 ) then c --- read the line prefix status =
mcapfx(handle, prefix_buffer) if( status.lt.0 ) then call edest('Prefix Read failed',0)
goto 100 endif c --- process the data ... goto 100 endif ...
```

Code samples longer than one page are not boxed.

An ellipsis (...) in a code sample means one of the following:

- Lines of code were omitted to condense the sample
- You will write lines of code to replace the ellipsis

Getting help

The McIDAS Users' Group (MUG) provides a McIDAS community mechanism for maintaining and upgrading McIDAS software and documentation, and other activities appropriate to a users group. The MUG provides SSEC with the financial support necessary to furnish these services to the users of McIDAS packages. Through its Help Desk, the MUG offers support to members on user, operations, and programming topics. In addition, SSEC can provide contract programming to develop or assist you with developing certain applications.

Use the table below to determine your best method of getting help.

If you want	**Contact**
clarification of any information contained in this manual to report an error in the manual to make a constructive suggestion for improving the manual	the Help Desk at (608) 262-2455, or send email to the Help Desk.
information about software maintenance and upgrades	the Help Desk at (608) 262-2455 or the McIDAS Website
help writing specific software or debugging your software	your SSEC Program Manager

Companion documentation

The following documentation provides additional information about McIDAS-X.

Online man pages

The online man pages provided with the McIDAS-X software contain detailed descriptions of all the API (Application Program Interface) functions that you will need to build applications. The following section provides more information on using the McIDAS-X man pages.

McIDAS User's Guide

This manual provides users with the installation instructions, general information and commands needed to run McIDAS-X. Each command includes a description of its usage, definitions for all positional parameters and keywords, remarks and examples. The manual also describes the Graphical User Interface (GUI), F Key Menu System, McBASI Interpreter, Abstract Data Distribution Environment (ADDE) and site administration and management.

McIDAS Learning Guide

This manual is a tutorial that guides users through the basics of McIDAS-X using a dataset supplied by SSEC. Each section introduces a new topic, including an introduction and practice session with step-by-step instructions. It contains these eleven lessons: Getting Started, ADDE, Loop Control System, Satellite Imagery, Graphics and the Cursor, Enhancements, MD Files, Grids and Grid Files, String Tables, Real-time Data Access, and the GUI.

Using the online man pages

The online man pages provided with the McIDAS-X software contain detailed descriptions of all the API (Application Program Interface) functions that you will need to build McIDAS-X applications.

The McIDAS-X man pages differ from Unix man pages in two ways.

- The McIDAS-X man pages are written for for McIDAS-X software only and tell programmers how to use a particular piece of McIDAS-X software in their programs.
- The McIDAS-X man pages are references and are not as complete as Unix man pages. For a complete description of each utility, see Chapter 4, *McIDAS-X Utilities*.

In this section, you'll find the following information about using the man pages.

Setting the Environment

You must modify the **MANPATH** environment variable in your **.profile** (ksh) or **.cshrc** (csh) files before you can view the man pages.

1. Login to your account and open the **.profile** or **.cshrc** file.
2. Add **~mcidas/man** to the **MANPATH** environment variable.
3. Logout and login again for the change to take effect.

Viewing man pages

To display the man page for an API function, enter the command below from the Unix prompt.

Type: **man** *APIfunction*

For example, if you wanted to display the man page for the API function **sdest**, you would enter the command below.

Type: **man sdest**

The man page is displayed as shown below.

```
Name: sdest - Puts a string to standard I/O device or file. Interface: subroutine sdest
(character*(*) charstring, integer n) Input charstring - Character string to be output. n
- An integer that will be output after the string. Input and Output: none Output: none
Return Values: none Remarks If 'n' is zero it is not printed. Categories text Filename
spout_.c
```

If you don't know the name of the API function, or want to search for all the API functions related to a particular topic, you can use the **xrefit** command using the formats below.

- **xrefit** *matchword*
- **xrefit -c** *matchword*

The **xrefit** command searches the name and category fields for *matchword*. In the sample **sdest** man page, the Name field contains the description *puts a string to a standard I/O device or file* and the Categories field contains the word *text*.

Use the **xrefit** *matchword* option to list all the APIs that contain the specified *matchword* in the name or categories fields. For example, to list all the API functions that contain the word *convert*, you would enter the command **xrefit convert**. A subset of the output is shown below.

```
graphic McTSTimeToXAxis Converts a time object to a TV element graphic McTSValueToYAxis
Converts a value to a TV line number converter Mcargparse Parses text into arg-fetching
structure day/time Mccydtodmy Converts ccyyddd to day/month/year sys_config Mcdev2uc
Converts keyword DEV= character value user_interface Mcstrtodbl Converts token to type
double format text cfe Converts a real character*12
```

The first column lists the category, the second column lists the API function, and the third column provides a one-line description of the API.

To further define your search, use the **-c** flag, which lists the APIs whose categories match the specified entry. For example, to list all the API functions with the category *converter*, you would enter the command **xrefit -c converter**. A subset of the output is shown below.

```
converter mcargparse Parse text into arg-fetching structure converter mchmstostr Converts
a time to a character string converter mcinsort Performs an insertion sort on a list
converter mcpcal Parses a list of comment cards converter mcstrtodbl Converts token to
type double format converter mcucvtd Converts an array of double precision
```

Directory location of the man page files

The McIDAS-X man files have names such as sdest.3, Mcgettime.4, and mcgetimageframenumber.3. The extension refers to the subdirectory containing the man files. The xref.tab file is a cross-reference file containing a one-line description of all the API functions. The **xrefit** function searches this file, as described in the previous section.

The McIDAS-X man page files and tools are placed in the **`~/mcidas/man`** directory, except for the search function **xrefit** which is stored in **`~mcidas/bin`**. All man files have the .3 extension and reside in the man3 subdirectory, as shown in the table below.

Directory	Contents
`~mcidas/bin`	xrefit
`~mcidas/man`	xref.tab xreftawk
`~mcidas/man/man3`	sdest.3 Mcgettime.3 all other man page files

Chapter 2

Learning the Basics

This chapter provides an overview of the basic concepts you must know to develop applications programs in McIDAS-X. First, it looks at McIDAS-X from a user's perspective. Then it provides an overview of McIDAS-X at the system level. Finally, it describes how to build user applications that will interact appropriately with the McIDAS-X environment. You will learn:

- Types of data available in McIDAS-X and how users access them
- Types of commands users can enter on the McIDAS-X Text and Command Window
- Coordinate systems McIDAS-X uses for displaying images and graphics on the McIDAS-X Image Window
- Components of the McIDAS-X environment and how they're organized
- Tools available for writing and debugging your code
- Formats and conventions to use when writing online command helps, setting status codes, naming new functions, or creating interface documentation blocks
- Some helpful hints for programming in McIDAS-X

This chapter is organized into these sections:

McIDAS user applications

McIDAS-X is a tool for collecting, displaying and analyzing earth science data. As a McIDAS-X programmer, you should have a basic, user-level understanding of McIDAS-X. This section provides a general overview of McIDAS-X from a user's perspective, including:

- The types of McIDAS-X data available to users
- How users access McIDAS-X data
- The windows displayed when a McIDAS-X session is started and how to use them

For a more complete description of user-level applications in McIDAS-X, see the *McIDAS User's Guide*.

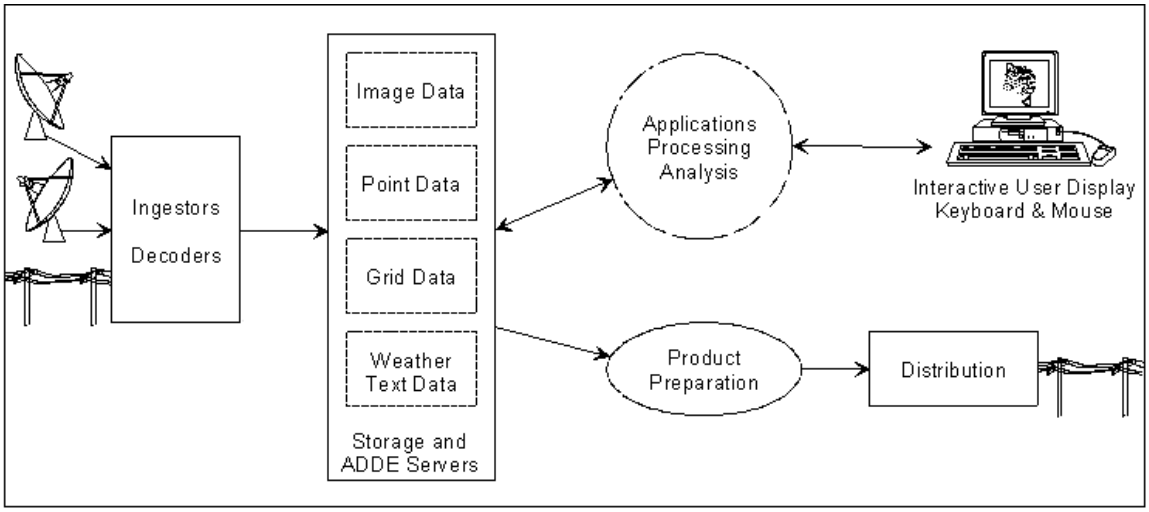
Data types

Ingestors and decoders receive raw data signals from a variety of sources and convert them into these McIDAS-X data types:

- Images
- Point observations
- Grids
- Weather text

Once the data is converted, users can store, analyze, display and manipulate it using McIDAS-X applications. Figure 2-1 shows how data is received, stored and distributed in McIDAS-X.

Figure 2-1. Ingestors and decoders partition, format, and file data on machines running ADDE (Abstract Data Distribution Environment) server software. Then client applications analyze, process, and convert the data.



A fifth data type, called static data, is also provided with the McIDAS-X software. Each type of McIDAS-X data is described briefly below. More information is presented in following chapters.

Images

McIDAS-X images refer to satellite and radar data. The primary characteristics of image data include:

- High volume
- High spatial resolution
- Low resolution of data values at individual points

Gray shading is the most common method of displaying image data.

Satellite images are received from geostationary and polar orbiting satellites. Geostationary satellites remain above a fixed location on the Earth's surface, about 36,000 km above the equator. They are limited in view, approximately 60° either side of the equator. Because geostationary satellites rotate with the Earth, they always view the same portion of the globe. There are usually five geostationary satellites approximately equally spaced, resulting in nearly complete global coverage.

Polar orbiting satellites orbit at much lower elevations, normally 800 to 900 km. Their field of view is about 2,400 km, centered on the orbit path. Unlike geostationary satellites, one polar orbiting satellite generally provides complete coverage of the Earth's surface twice per day.

Radars use active sensors that emit short-wave radiation and sample the signals reflected back to the radar antenna. The information represented in radar data is related to the strength of the reflected radar signal and is usually correlated with rainfall intensities. Modern radars also sense the radial component of droplet velocity.

Point observations

Point observations refer to data reports at specific, irregularly spaced locations. These reports usually contain data for multiple parameters. Most data gathered by direct measurements, such as weather balloons and synoptic reports, is stored as point data.

Point data is listed on the McIDAS-X Text and Command Window in tabular form. If displayed on the Image Window, it is superimposed as a graphic on a frame at the reporting locations.

Grids

Grids refer to data placed at regularly spaced intervals at some level in the atmosphere. Grids are displayed graphically using isopleths.

Like images, grids represent data in a two-dimensional, evenly spaced matrix. Unlike images, grids represent data having low volume and high resolution at each data point.

Since grids may represent forecast fields from numerical models, they must have two separate time attributes to identify the data temporally. One time attribute represents the time the model was initialized. The second represents the time the data is valid. For example, a 36-hour forecast from a model run at 00 UTC on 17 January is valid at 12 UTC on 18 January.

Weather text

Weather text refers to any information transmitted in alphanumeric form. It can be man- or computer-generated output containing forecasts, observations, weather advisories or other public information.

Static data

In McIDAS-X, static data refers to database information that changes little over time. Two examples of static data are:

- Map files
- Station tables

These static data make dissemination of the other McIDAS-X data easier.

The McIDAS-X MAP command uses map files to superimpose outlines of political or geographic boundaries on the McIDAS-X Image Window. These outlines may be associated with data already on the frame or defined in the MAP command itself.

Station tables provide a cross reference list of reporting stations, independent of the reports themselves. For example, if a synoptic observation tells you that the station reporting is number 72641, the report doesn't tell you that 72641 is Madison, Wisconsin. The station tables provide this information.

How users access McIDAS-X data

Users request, receive and display McIDAS-X data using the following:

- ADDE (Abstract Data Distribution Environment)
- File redirection
- MCPATH

ADDE makes McIDAS-X image, point, grid and weather text data available to users. During a McIDAS-X session, however, users may need to access other files, such as font files, political and geographic boundary files or station tables, which reside on their workstations but are not available with ADDE. These files can be accessed using file redirection. Additionally, users may want to access files specific to their application that aren't needed by other users. The Unix environment variable MCPATH allows them to do this.

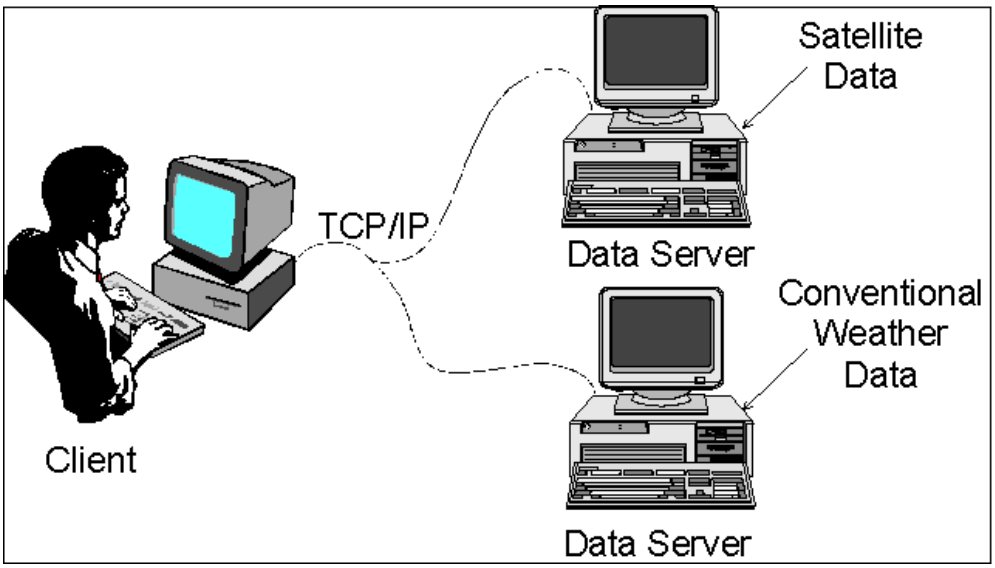
ADDE, file redirection and MCPATH are briefly described on the following pages.

For more information about accessing McIDAS-X data, see the *McIDAS User's Guide*.

ADDE

ADDE is a protocol developed for McIDAS that uses a networked system of clients and servers to make weather and other geophysical data available to users. A *client* is a workstation in a distributed system that initiates a data request, then receives and displays the requested data. A *server* is the machine in a distributed system that stores data and supplies it to the client upon request. ADDE allows many users to access ingested and decoded data from the same machine, regardless of their location. See Figure 2-2 below.

Figure 2-2. ADDE is a networked system of clients and servers that communicate using the TCP/IP communications protocol.



Each McIDAS-X workstation session acts as both a client and a local server. When a client requests data from the local server, this server searches for the data in the directories that the user's session has access to. A client may also request data from a *remote server*, which can be one of the following:

- An account on another McIDAS-X workstation configured as a remote server
- A different account on the same McIDAS-X workstation configured as a remote server

ADDE naming scheme

Any ADDE command entered by a user to access data must include the ADDE dataset name. The ADDE dataset name consists of two parts:

- Group name
- Descriptor name

The *group name*, which the user configures with the McIDAS-X DATALOC command, is stored in a *client routing table*. On the client, the group name identifies the server machine to get the data from. On the server, it helps identify the data that the client is requesting.

The server uses the *descriptor name* to identify the type of data the user wants to access and the range or names of files to search. The table containing this information is called the *server mapping table*. Users assign dataset names on the server with the DSSERVE command.

Data flow in ADDE

When a user enters an ADDE request, the McIDAS-X software performs the steps below to determine the source of the data.

1. The application extracts the group name from the request.
2. The group name is compared to the entries in the client routing table to determine if the requested group is defined in the user's account or on a remote server.
3. *If the requested group exists locally*, a server is started under that user's account. The server tries to resolve the dataset name. If the dataset name is resolved, the server mapping table provides the server with information about the type of data being served and where it can find the data. When the data is found, it is sent back to the application for processing.
4. *If the requested group exists on a remote workstation*, a connection is made between the client and the remote workstation and a server is started on that workstation. The remote server tries to resolve the dataset name stored in the server mapping table. The server mapping table provides information to the server about the type of data being served and where it can find the data. When the data is found, it is sent back to the application for processing.
5. When all the data is returned, the connection between the client and server is dropped.Once the client routing table and server mapping tables are configured, the application won't know or care if the data arrives remotely or locally.

File redirection

File redirection lets a user identify the location of individual files on the system. The default configuration of a McIDAS-X session is to search for local files in the following directories.

- **`$HOME/mcidas/data`**
- **`~mcidas/data`**

Local files may exist anywhere on the workstation, as long as an application knows where to look for them.

Entries in the file redirection table are made with the McIDAS-X REDIRECT command. For example, a user who has a McIDAS-X application that requires the file MYFILE in the directory **`/home/fred/data`**, would enter this REDIRECT command:

REDIRECT ADD MYFILE "/home/fred/data

All subsequent commands that access MYFILE would assume it is in **`/home/fred/data`**.

MCPATH

McIDAS-X also contains the environment variable, **`MCPATH`**, which defines directories for McIDAS-X commands to search when looking for data and help files. Commands requiring these types of files search each directory in the order listed in **`MCPATH`**.

If a user lets McIDAS-X set **`MCPATH`**, it will contain the directories shown below. Other directories can be added to **`MCPATH`** using the colon (:) character to separate the directory names.

```
$HOME/mcidas/data:$HOME/mcidas/help:~mcidas/data:~mcidas/help
```

If a file exists in more that one directory in **`MCPATH`**, McIDAS-X will use the first one that it finds. If you write data to a new file, **`MCPATH`** will place that file in the first writable directory in the **`MCPATH`** list.

The McIDAS-X windows

When a user starts a McIDAS-X session, these two windows appear on the screen:

- McIDAS-X Text and Command Window
- McIDAS-X Image Window

The title bar in each window lists the version of McIDAS-X. In McIDAS-X, it also shows the user's logon and the host name. Optionally, a Graphical User Interface (GUI) can be started, which will become part of the Image Window.

In this section, you will learn how the windows are used and the type of output displayed on them.

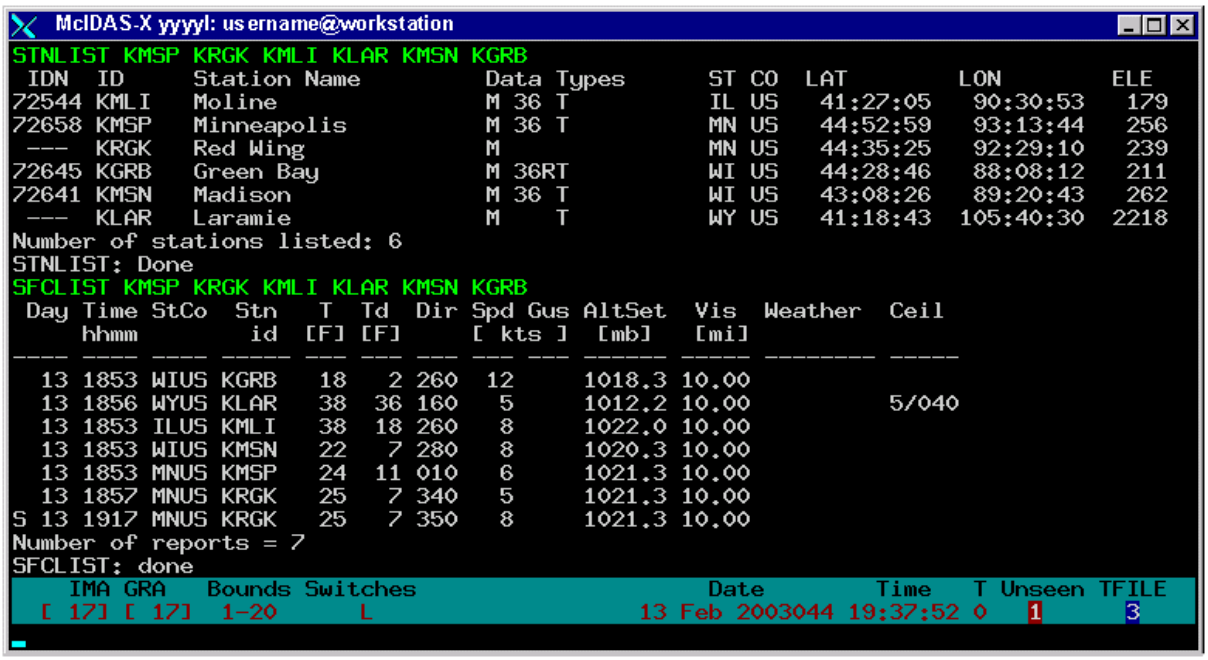
McIDAS-X Text and Command Window

The McIDAS-X Text and Command Window is used for:

- Entering McIDAS-X commands
- Displaying command output
- Showing workstation status information

When a session is started, McIDAS-X can display output on 10 different text frames in this window. You can switch between these text frames using the numeric keypad on the keyboard. Figure 2-3 shows a sample McIDAS-X Text and Command Window.

Figure 2-3. The McIDAS-X Text and Command Window displays text messages.



Users enter two kinds of McIDAS-X commands on this window:

- Single-letter commands, which can be either system defined or user defined
- Multiple-letter commands containing positional parameters, keywords and quoted text

Single-letter commands

Many system defined single-letter commands toggle the McIDAS-X session between different modes. Others run commands to provide information. These system defined commands are run by simultaneously pressing the Alt key and the letter key, or by typing the letter and pressing Enter from the Text and Command Window.

The table below lists the system defined single-letter commands currently provided in McIDAS-X.

Command	Description
A	advances one frame
B	backs up one frame position
C	lists a frame's directory
D	lists the digital values at the cursor center
E	lists the earth coordinates at the cursor center
F	displays the workstation's state
J	toggles graphics frames to and from the loop control system
K	toggles image frames on and off
L	toggles frame looping on and off
O	switches the image frame position to the opposite frame
R	resizes the Image Window to the size of the current frame
S	lists Metar, Synoptic and RAOB stations within the cursor bounds
V	toggles the velocity cursor on and off when tracking cloud drift winds
W	toggles the graphics frame on and off

X	tracks cloud drift wind targets with one keystroke
Y	toggles image frames to and from the loop control system
Z	toggles the McIDAS-X zoom function on and off

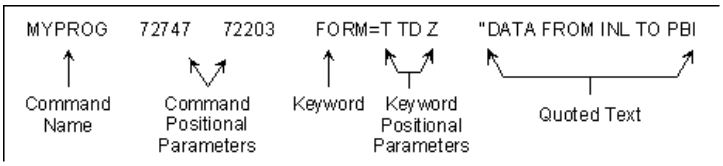
User defined single-letter commands are run by simultaneously pressing the Ctrl key and the letter key. This causes the contents of the McIDAS-X string corresponding to this letter to be run.

Multiple-letter commands

Multiple-letter commands, which are the most common interface for manipulating a McIDAS-X session, have four components:

- Command name
- Command positional parameters
- Keywords with positional parameters
- Quoted text

A sample McIDAS-X command line is shown below.



Each command line component and value is separated by one or more blank spaces. An individual parameter requiring a space must be surrounded with single quotes (` ') so it is treated as one parameter.

Each part of the McIDAS-X command line is described below.

Command positional parameters

Command positional parameters provide input to a command and must be entered in the exact order specified. Use them in commands that have options which the user must always enter. One advantage of positional parameters is minimizing the number of keystrokes a user types. Be careful not to negate this advantage by using so many positional parameters that the user can't remember them all.

Keywords

Like command positional parameters, keywords are used for entering command input. Unlike command positional parameters, they are optional for most McIDAS-X commands and can be entered in any order as long as they follow command positional parameters and precede quoted text.

Keyword parameters are often used to clarify commands with many complicated options. Although keywords can occur in any order, their positional parameters must be entered in the order indicated. Since users don't always specify all keyword positional parameters in a command, be sure to assign them reasonable defaults.

In addition to the keywords that an application has built into it, McIDAS-X has seven global keywords, which are common to all McIDAS-X commands:

- DEV= specifies the destination device of the text output generated by a command.
- FONT= specifies the font for drawing text on the McIDAS-X Image Window.
- MCC= specifies the data compression method the ADDE remote server should use when sending the requested data to your workstation.
- PAN= specifies frame panel number for displaying output or reading frame information .
- TCOL= specifies the color for command text output in the text window.
- TWIN= specifies the destination text window to route the command output.
- VIRT= specifies a virtual graphics number to write graphical output.

You can retrieve the parameters for these global keywords in your applications, as long as you don't change their functionality.

For more information about the McIDAS global keywords, see the *McIDAS User's Guide*.

Quoted text

Quoted text input is most often used when a single string entered by a user may require extra spaces. Each application can contain only one quote string and it must be the last part of the command the user enters. Using quote fields in McIDAS commands is relatively easy from the McIDAS

Text and Command Window; running McIDAS commands with quote strings from Unix shell prompts is more difficult. For this reason, the preferred method for entering strings with spaces in them from the command line is to use a positional or keyword parameter and surround the string with single quotes.

Text output

Text output generated by commands occurs in three different forms.

- *Standard text* messages are sent to **stdout** (standard output) and are used to display normal text output, such as listings or user instructions.
- Error text messages are sent to **stderr** (standard error) and describe the failure of an application to perform a specific task. These messages are automatically prefixed with the program name followed by a colon. Error messages typically occur when a command is entered incorrectly or data requested is not available.
- *Debug* messages are sent to **stderr** (standard error) and contain details about the internal state of an application at runtime. Although debug messages aren't normally of interest to a user, they can identify why an application fails to perform a task. Debug messages are prefixed with the program name followed by an asterisk.

The default mode for text output is to display standard and error text messages on the Text and Command Window, and to suppress debug messages. Use the DEV global keyword to specify the destination device of the text output generated by a McIDAS command.

Online helps

Online helps list the syntax of each McIDAS-X command including the parameters, keywords and remarks. To access the online help, type HELP in the McIDAS-X Text and Command Window followed by the command that you want information for. To get the online help for the DSINFO command, for example, you would Type: **HELP DSINFO**.

Alternatively, you can invoke the HELP command while entering a command from the keyboard by typing the command name, then pressing **Alt ?**. An abbreviated help is displayed on the current text frame.

McIDAS-X Image Window

The McIDAS-X Image Window displays frames containing McIDAS-X generated images and graphics. Figure 2-4 shows a sample McIDAS-X Image Window. The McIDAS-X Image Window can optionally be surrounded by the McIDAS-X GUI, as shown in Figure 2-5 below.

Figure 2-4. The McIDAS-X Image Window displays images and graphics.

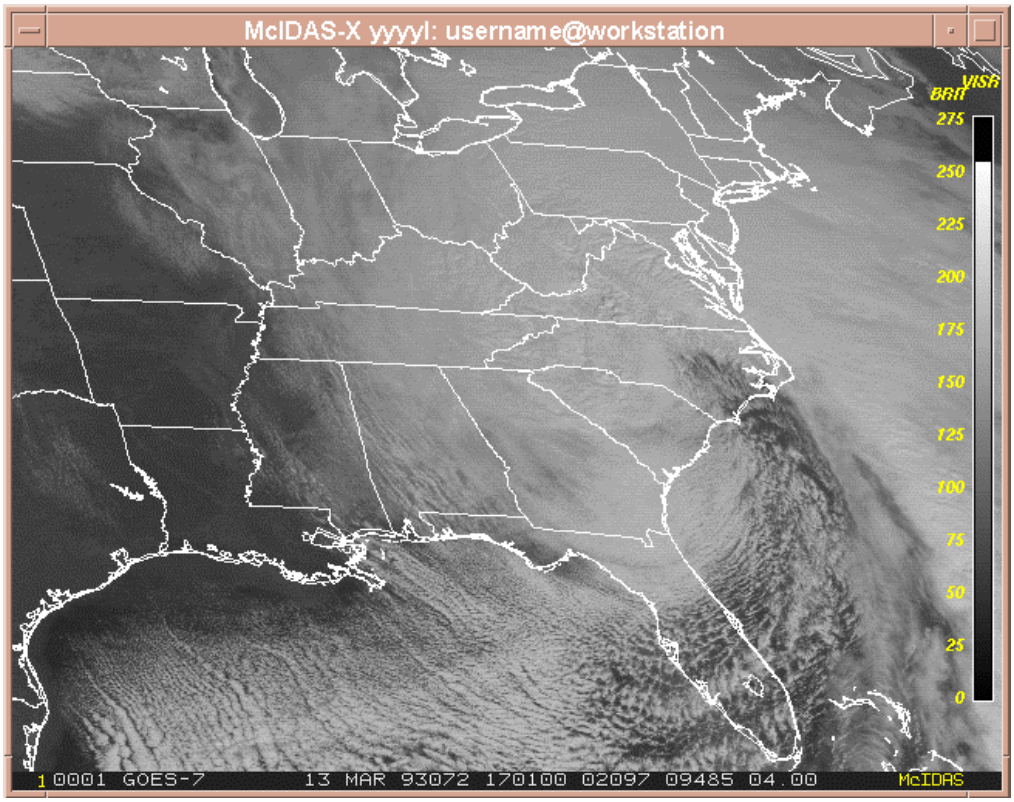
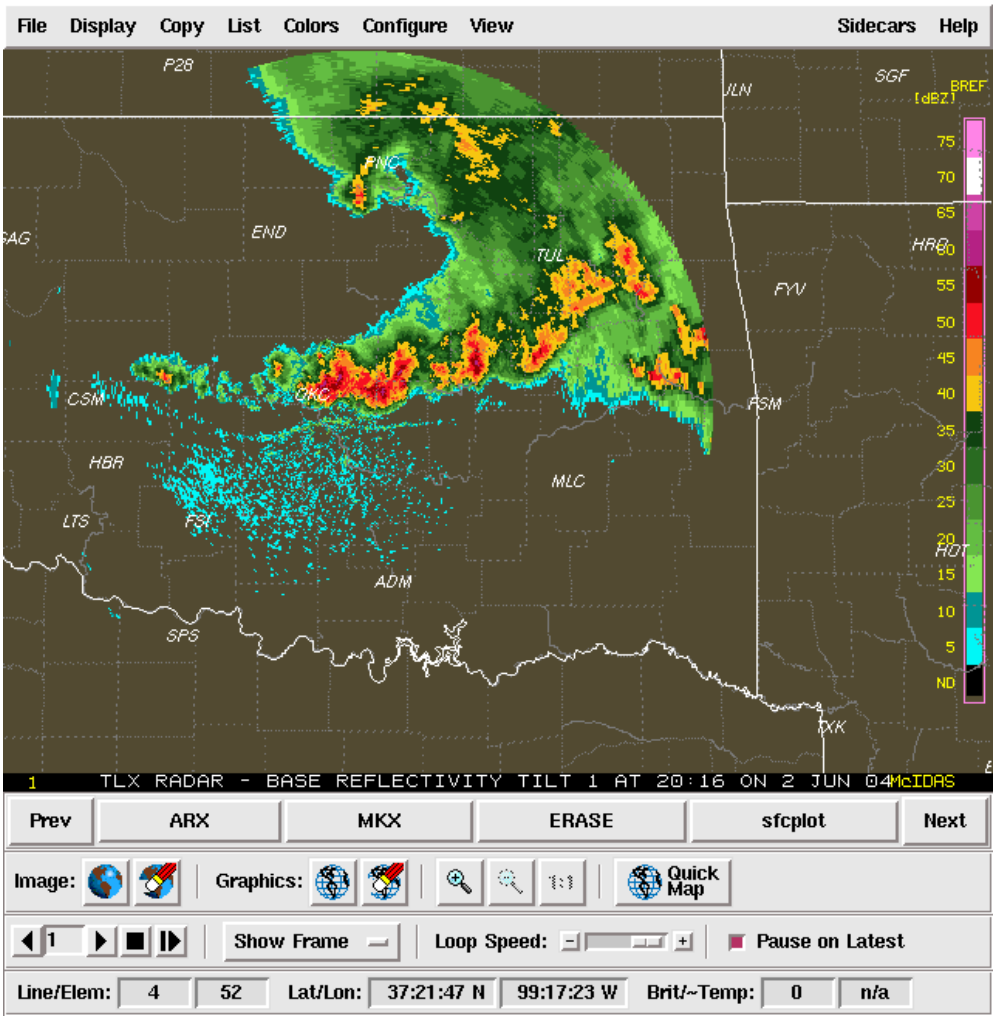


Figure 2-5. The McIDAS-X Image Window can optionally be surrounded by the McIDAS-X GUI.



The manner in which images and graphics are displayed on the Image Window is determined by several factors:

- Number of gray shades and graphical colors defined by the user
- Red, green and blue color values assigned by the user
- Number of frames in the window and their size
- Frame looping sequence
- Coordinate system used to define the location of data points in an image

These factors are described below.

Image and graphics levels

The McIDAS-X Image Window displays satellite and radar data in shades of gray. Alphanumeric data and scientific diagrams are displayed with colored graphical lines and symbols independent of the grayshaded data. Users define the number of gray shades and graphical colors in the **`.mcidasrc`** file. The default configuration is 128 gray shades and 16 graphics levels with a maximum of 254 total levels for the image and graphics levels.

Color enhancements

When McIDAS-X displays images and graphics on the Image Window, it stores each pixel value in a frame object. The user can map these pixel values to specific red, green and blue color values. The McIDAS-X EU and EB commands modify the color combinations for image data; the GU command modifies the colors of graphics levels.

Frame size

Via the **`.mcidasrc`** file, users can also define the number and size of frames in the McIDAS-X Image Window. The default is to start the session with four frames that are 480 lines (vertical) by 640 elements (horizontal).

Loop control

McIDAS-X can display an automatically repeating sequence of frames much like a movie loop. The McIDAS-X commands LS and LB define the sequence of frames. The DR command modifies the speed of the loop. The **Alt A** and **Alt B** commands step through the individual frames in the loop. **Alt L** starts and stops the loop.

Coordinate systems

McIDAS-X uses four different, yet interconnected coordinate systems to define the location of data points within an image:

- Image coordinates
- File coordinates
- Earth coordinates
- Frame coordinates

These coordinate systems are used to reference image data on disk and depict it on the McIDAS-X Image Window. McIDAS-X graphics also use a *world* coordinate system, which is described in the section titled *Advanced McIDAS-X graphics techniques* in *Chapter 4, McIDAS-X Utilities*.

Image coordinates

McIDAS-X receives satellite and radar images. The image coordinates for this data are defined by the sensor source providing the image and form the basis for the other McIDAS-X coordinate systems.

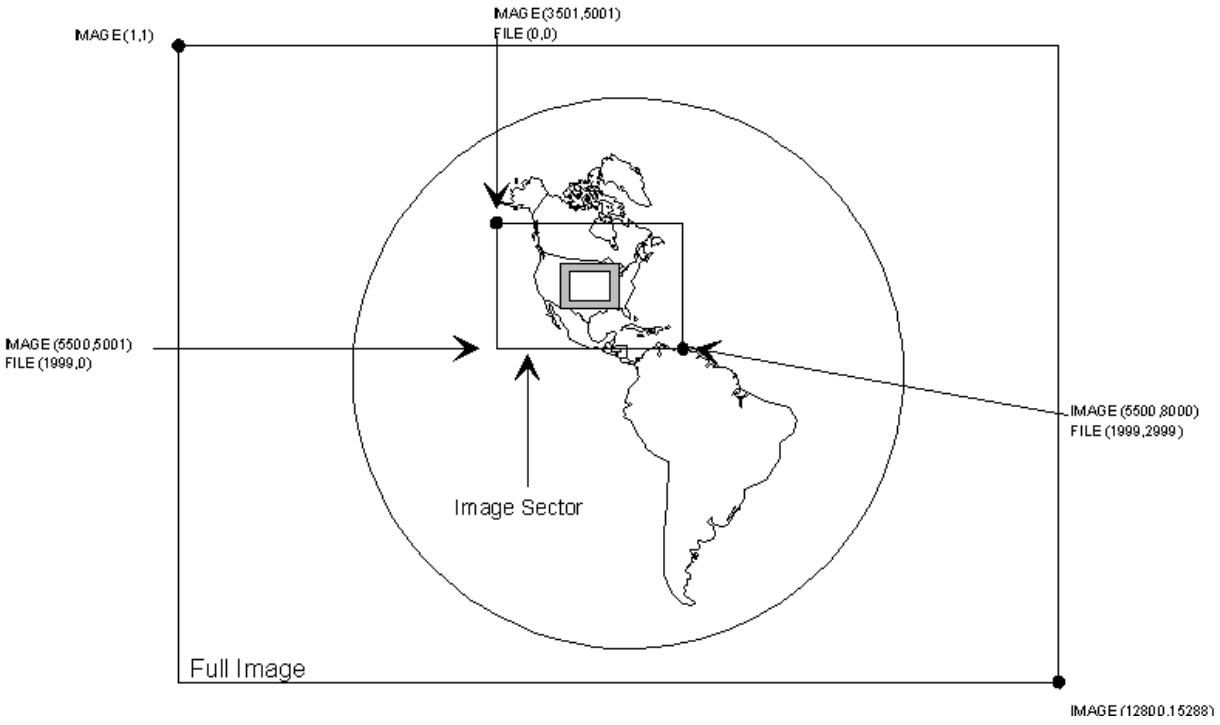
A *full image* is the entire image transmitted by the sensor. It is a sequence of lines and elements usually arranged from top to bottom, forming a grid for displaying data points on a McIDAS-X frame. Lines run horizontally across the frame; elements run vertically up and down the frame. The top line and leftmost element have the image coordinates (1,1). Each pixel has a unique pair of line and element values that are its image coordinates.

Some sensors have the ability to send only a sector covering a region of interest instead of transmitting a full image. This image sector is a rectangular subset of an image with the same coordinate system. An image sector may also be at a lower *resolution*, meaning data sampled in lines and/or elements, but the image coordinate system always refers to it relative to a full image. See Figure 2-6 below, which shows a full image and image sector. The upper-left image coordinates of the full image are (1,1); the upper-left coordinates of the image sector are (3501,5001).

File coordinates

In McIDAS-X, images and image sectors are stored in files called *areas*. File coordinates are based on the physical size of the image sector. They are zero-based and represent the location of a data point in an area file referenced sequentially by lines and elements. In Figure 2-6, the first data point in the image sector has file coordinates (0,0). The bottom-right data point has file coordinates (*number of lines* minus 1, *number of elements* minus 1).

Figure 2-6. Image and file coordinates are shown for a full image and image sector.



Earth coordinates

If the data is navigated, the image coordinates can be converted to earth coordinates (latitude and longitude) and vice versa. Earth coordinates are specified in degrees, minutes, and seconds in the form DDD:MM:SS, or in decimal degrees such as 46.36. In McIDAS-X, all latitudes south of the equator and longitudes east of Greenwich are negative. Latitudes run from -90:00:00 to +90:00:00 and longitudes run from -180:00:00 to +180:00:00.

Frame coordinates

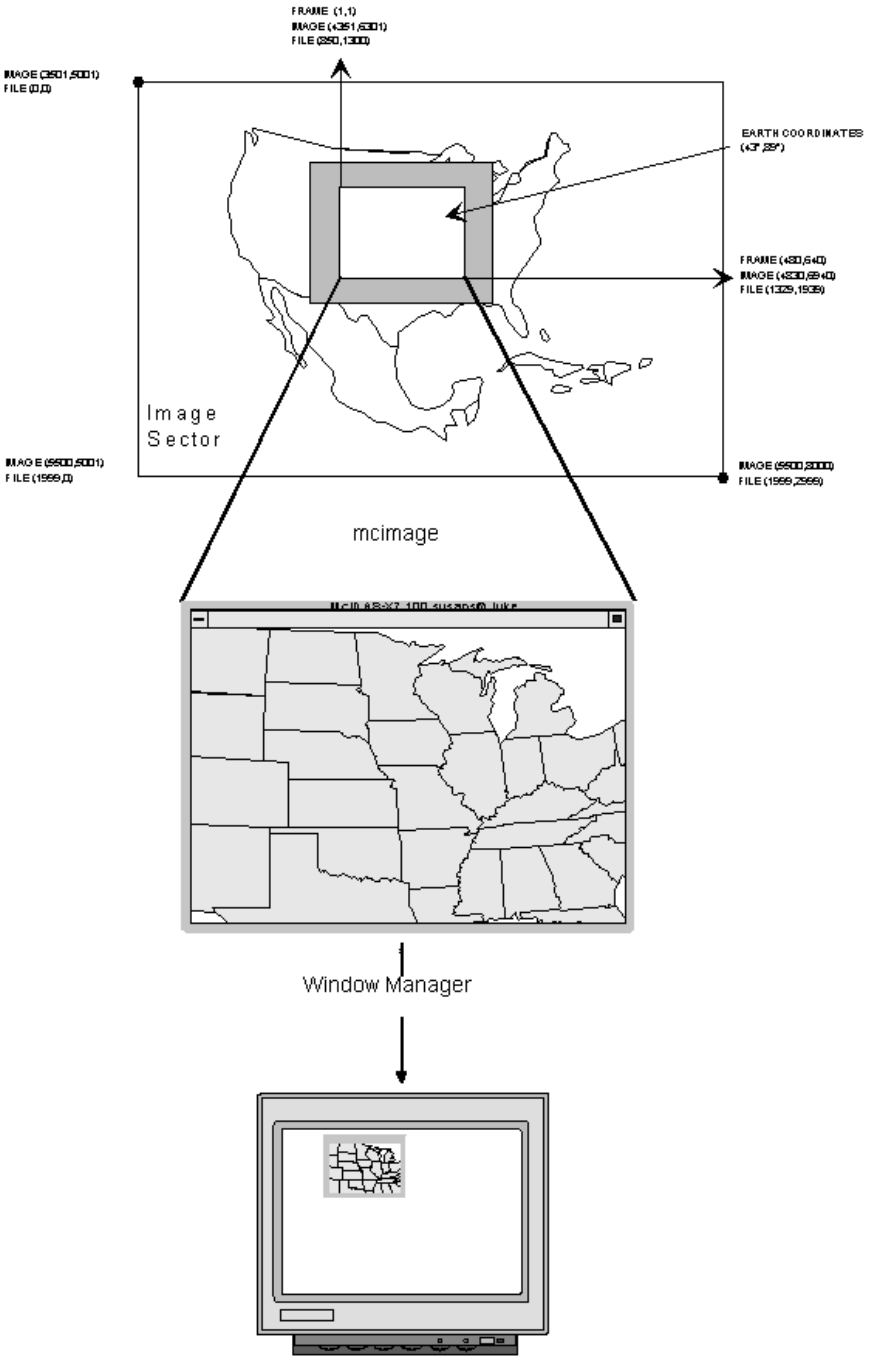
The pixels on McIDAS-X frames are arranged by lines and elements. A *frame* contains a representation of an image sector displayed on the McIDAS-X Image Window. The image sector shown in Figure 2-7 is an enlarged view of the image sector shown in Figure 2-6. The **`mcimage`** program, which is discussed later, takes the image data in the frame and maps it to the McIDAS-X Image Window. The Window Manager then displays the McIDAS-X Image Window on the workstation monitor.

The pixel in the upper-left corner of the frame has the frame coordinates (1,1) which means (line 1, element 1). The total number of lines and elements on the frame is determined by the frame size. The lower-right corner of the default-sized frame is (480,640).

For information about converting coordinate systems from line/element to latitude/longitude and vice versa, see the *Image data* and *McIDAS-X navigation* sections in *Chapter 5, Accessing Data*.

Frames may be subdivided into rectangular regions called panels. Each panel on an image frame has its own relative coordinate system, which defines the upper-left corner as (1,1). For information about panel generation, and interaction, refer to the *McIDAS User's Guide*.

Figure 2-7. Earth and frame coordinates are shown for the image sector.


FRAME (1,1)
IMAGE (+351,6301)
FILE (350,1300)
IMAGE (3501,5001)
FILE (0,0)
EARTH COORDINATES
(43°,25°)
FRAME (480,640)
IMAGE (4330,6940)
FILE (1329,1939)
Image
Sector
IMAGE (9500,5001)
FILE (1999,0)
IMAGE (9500,8000)
FILE (1999,2999)
mcimage
Window Manager

McIDAS system overview

Now that you've seen McIDAS-X from a user's perspective, you need to understand how McIDAS-X operates at the system level. When McIDAS-X is started, it creates the environment in which it will run. In this section, you will learn about the system-level components of the McIDAS-X environment and how they fit together.

The McIDAS-X environment consists of these three components:

- Shared memory
- Resident programs
- Applications programs

Each is described below.

Shared memory

When a McIDAS-X session is started, a block of shared memory is created as an inter-process communication mechanism between McIDAS-X applications, resident programs and the display environment. This shared memory block is composed of:

- User Common
- Image frame objects
- Redirection tables

The shared memory key associated with the memory is stored in the environment variable **`MCENV_POSUC`**.

User Common

User Common (UC) contains information about the McIDAS-X session, such as the current image frame in view, the maximum number of image frames and the cursor position. UC has two parts:

- Positive UC
- Negative UC

Positive user common

Positive UC contains the current state of the McIDAS-X session. This information is available as long as the session that created positive UC is active. Values in positive UC may change. For example, if frames are looping or you're moving the cursor on the screen, the values in positive UC related to the current frame number or cursor position will change.

Negative user common

Negative UC is not part of the shared memory block created when the McIDAS-X session is started. It is created by **`mmap`** when a McIDAS-X command is started, and is initialized with the values in positive UC at the time the command is entered. The file descriptor key associated with negative UC is then placed in the environment variable **`MCENV_NEGUC`**.

Negative UC values are not changed by other activities in the McIDAS-X session, such as frame looping or cursor motion. When a command finishes running, its copy of negative UC is deleted if it was not inherited from a parent process.

Visibility and inheritance

As a McIDAS-X programmer, you need to understand the concepts of *visibility* and *inheritance* when accessing positive and negative UC.

- Visibility describes the portions of the McIDAS-X session that can access specific UC information.
- Inheritance refers to the origin of the UC that an application receives at initialization.

The User Common policy for visibility and inheritance when running a McIDAS-X command is described below.

Does positive UC exist?

- If positive UC does not exist, the McIDAS-X software creates the shared memory block and assigns the memory key to **`MCENV_POSUC`** (positive UC). Otherwise, the software attaches positive UC to the McIDAS-X command.
- Once positive UC exists, it is visible to all the commands run in that session.

Should this application inherit negative UC?

- The parent process determines if an application inherits negative UC.
- *If a McIDAS-X command runs synchronously*, it inherits negative UC from its parent process. This happens, for example, when a command is run from a McIDAS-X command file. The application inherits the parent by attaching itself to the file descriptor key stored in the parent's **MCENV_NEGUC**. Values in positive UC are not copied into negative UC when a command inherits its parent's copy of negative UC. The visibility of this copy lasts until the parent process that created it ends.
- *If a McIDAS-X command runs asynchronously* relative to its parent process, the command creates its own negative UC and associated **MCENV_NEGUC**, and initializes it from values in positive UC. This can occur, for example, when a command is started from the McIDAS-X Text and Command Window. The version of negative UC created is visible only while the command that created it is active.

For a complete list of the contents of User Common, see *Chapter 6, Format of the Data Files*.

Image frame objects

Image frame objects contain information describing the contents and appearance of a frame displayed on the McIDAS-X Image Window. McIDAS-X provides a variety of library functions and commands to manipulate the contents of frame objects. An individual frame object contains not only pixel values to display, but also tables for image color enhancements, graphics color enhancements and stretching tables.

Redirection tables

Redirection tables are stored on disk in the file **$HOME/mcidas/data/LWPATH.NAM**. When McIDAS-X is started, the contents of the current redirection table are loaded into shared memory. This is where the McIDAS-X disk file system retrieves its redirection information.

Resident programs

The second component of the McIDAS-X environment is resident programs. When a user types the **mcidas** command, the user-specified configuration information, which is in the text file **$HOME/.mcidasrc**, is determined.

Once the configuration information is extracted, the resident programs below are started.

- **mcenv**
- **mctext**
- **mcimage**

mcenv

The **mcenv** program performs the following procedures to start a McIDAS-X session:

- Creates the shared memory segment that will contain positive UC, the frame objects and the redirection table
- Assigns the shared memory key ID to the environment variable **MCENV_POSUC**
- Creates a temporary directory in **$HOME/.mctmp** with the same name as the shared memory key
- Starts a command, if one is specified; note that when the **mcidas** script starts **mcenv**, it specifies that **mctext** should be started

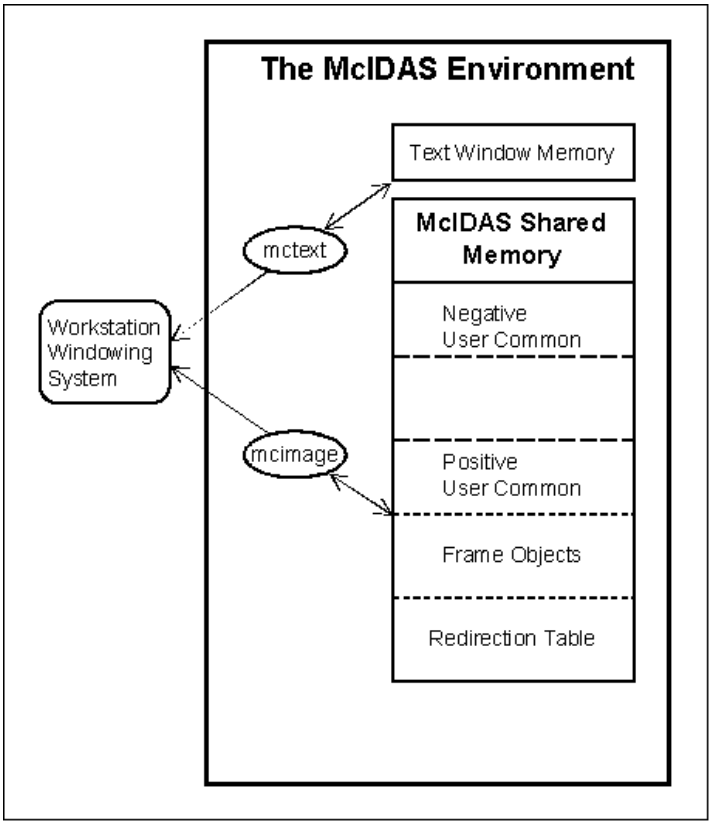
The temporary directory created by **mcenv** stores session-dependent files such as frame directories, string tables and frame enhancement tables.

mctext

The **mctext** program performs the tasks below, as shown in Figure 2-8:

- Maintains the Text and Command Window for McIDAS-X
- Allocates enough memory to contain the command line and alphanumeric text components needed for the session
- Monitors keyboard entry and the placement of the alphanumeric output generated by the commands

Figure 2-8. The mctext program controls command line input from the keyboard and displays text output on the Text and Command Window, while mcimage controls the McIDAS-X Image Window for displaying images and graphics.



The command line arguments for **mctext** can alter the number of text lines in the window buffer and recall the number of previously run commands. **mctext** can also start several other commands once it initializes its window environment.

When the **mcidas** script starts a session, one of the command arguments passed to **mctext** is **mcimage**.

mcimage

As shown in Figure 2-8, the **mcimage** program does the following:

- Translates the contents of the frame objects into display characteristics for the workstation windowing system
- Combines the brightness values, color enhancement information and stretching tables into pixel values that are displayed on the McIDAS-X Image Window
- Controls the display of the McIDAS-X cursor

Applications programs

The third component of the McIDAS-X environment is applications programs. A McIDAS-X applications program is any program that runs from the McIDAS-X command line. McIDAS-X provides Application Programming Interface (API) functions that interact with all the McIDAS-X components described in this chapter. These API functions interact with the Image Window, Text and Command Window, and mouse. The programs access data both locally and through ADDE, and manipulate the contents of the session, as shown in Figure 2-9 below.

Figure 2-9. McIDAS-X applications rely on the API functions to read data files for them.

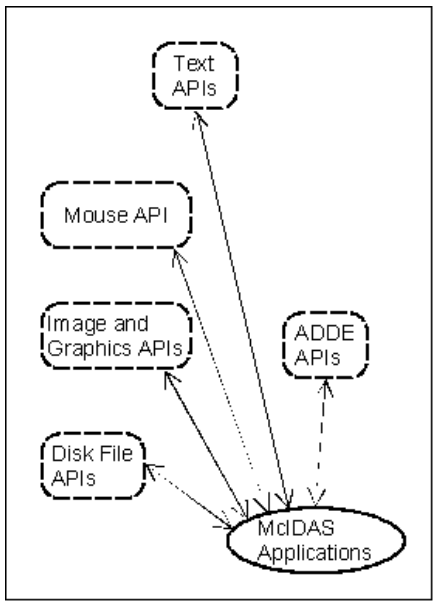
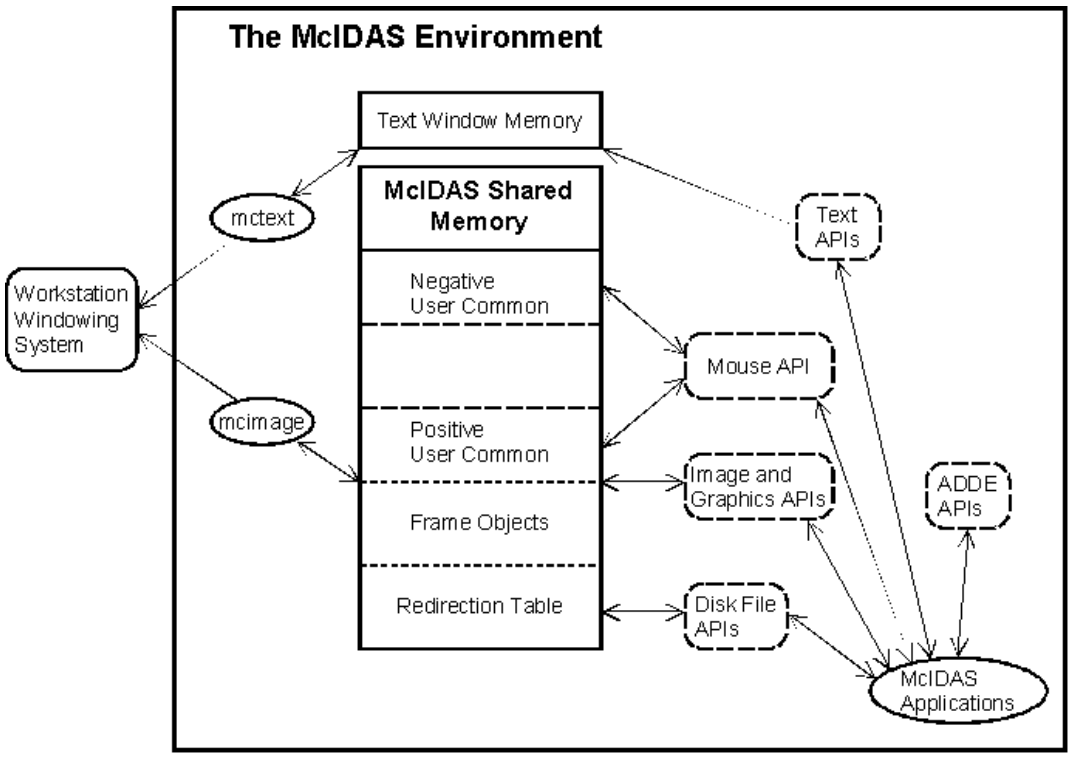


Figure 2-10 combines the three components of the McIDAS-X environment: the shared memory, resident programs and applications. The arrows represent APIs and imply the flow of data or information.

Figure 2-10. The McIDAS-X environment consists of shared memory, the resident programs **`mctext`** *and* **`mcimage`***, and user applications.*



McIDAS developer overview

The previous two sections introduced you to McIDAS-X from a user's perspective and provided an overview of the McIDAS-X environment. This final section describes how to build applications that will interact with the McIDAS-X environment. In this section, you will learn how to:

- Get started writing McIDAS-X applications in Fortran and C
- Use McIDAS-X include files
- Set status codes in McIDAS-X
- Write online command helps
- Name your McIDAS-X functions
- Write an interface documentation block
- Create routines that interact in both C and Fortran
- Debug your code
- Develop your code to be platform-independent

This chapter contains references to specific source files. The table below describes the suffixes McIDAS-X uses to name source files.

Suffix	Language	Description
`.c`	C	functions and McIDAS-X commands
`.cp`	C	ADDE servers and non-McIDAS-X applications
`.dlm`	Fortran	dynamic link modules
`.for`	Fortran	functions and subroutines
`.fp`	Fortran	ADDE servers and non-McIDAS-X applications
`.h`	C	include files
`.inc`	Fortran	include files
`.mac`	Fortran	McIDAS-X macros
`.pgm`	Fortran	McIDAS-X commands

Writing McIDAS-X applications

You can write McIDAS-X applications in either C or Fortran. This section describes the concepts you need to know to write basic McIDAS-X applications in either language, along with the formats and policies for writing online command helps and setting status codes.

Fortran programs

When you write a Fortran program for McIDAS-X, you can't write it as a Fortran MAIN program. Instead, you must place it in the McIDAS-X environment as a subroutine named MAIN0, which is linked with a command jacket called **main**.

The command jacket **main** initializes the run-time environment for the command, including:

- Parsing the command line
- Initializing User Common

The first executable line of any McIDAS-X application written in Fortran must be the statement **`subroutine main0`**. The code fragment below is a sample command to print the phrase *Hello World* to the McIDAS-X Text and Command Window.

```
subroutine main0 call sdest ('Hello World',0) call mccodeset (0) return end
```

C programs

When writing a C program for McIDAS-X, the main is provided directly in the C code. The first thing you will do when writing a McIDAS-X application in C is to initialize the McIDAS-X environment with a call to the function **Mcinit**. This function performs the same command

environment initialization performed by the Fortran command jacket **main**. To write McIDAS-X applications in C, your source code should also contain a reference to the McIDAS-X include file **`mcidas.h`**.

The code fragment below demonstrates how the same McIDAS-X command to print *Hello World* is written as a C application.

```
#include <stdio.h> #include "mcidas.h" int main (int argc, char **argv) { /* initialize
the McIDAS environment */ if (Mcinit (argc, argv) < 0) { fprintf (stderr, "%s\n",
Mciniterr ()); return (1); } Mcprintf ("Hello World\n"); Mccodeset (0); return
(Mccodeget()); }
```

Include files

McIDAS-X uses include files to hold definitions of constants specific to its software. **`mcidas.h`** contains system-wide constants and function prototypes. Equivalanet constants for Fortran applications are found in the **`*.inc`** files.

mcidas.h

The include file **`mcidas.h`** contains all the function prototypes for C-callable API routines in the McIDAS-X library. It also contains two constant values that are commonly used in McIDAS-X applications.

Constant value	Description
MCMISSING	1-byte McIDAS-X standard missing value code
MCMISSING4	4-byte McIDAS-X standard missing value code

`mcidas.h` also includes a group of defined types that you should use to ensure platform independence of data types. Use the defined types below when writing interface routines that will be used between C and Fortran.

Defined type	Description
Fdouble	equivalent to a double precision declaration type in Fortran
Fint	equivalent to an integer declaration type in Fortran
Fint2	equivalent to an integer*2 declaration type in Fortran
Fint4	equivalent to an integer*4 declaration type in Fortran
Freal	equivalent to a real declaration type in Fortran
Freal4	equivalent to a real*4 declaration type in Fortran
Freal8	equivalent to a real*8 declaration type in Fortran
FsLen	used for passing string lengths into Fortran routines
Mcint2	2-byte signed integer
Mcuint2	2-byte unsigned integer
Mcint4	4-byte signed integer
Mcuint4	4-byte unsigned integer

To ensure portability of Fortran jackets written in C, the preferred types to use are Fdouble, Fint and Freal, as the INTEGER*2, INTEGER*4, REAL*4 and REAL*8 declarations are not part of the Fortran-77 standard.

Error handling

In both *Hello World* examples above, calls were made to the functions **mccodeset** and **Mccodeset**. These functions set a status code that can be passed from the application to the calling environment. Currently, the acceptable return codes for McIDAS-X commands are as follows:

Return code	Description

0	command completed successfully
1	command contained an unrecoverable error
2	command contained an error that may be recoverable
3 - 99	reserved for SSEC
100 - 127	site-defined return values

This information may be very useful to the calling environment, especially when using McIDAS-X commands in scripts designed for batch processing. For example, if you have a script that copies satellite images from several different sources and then creates a mosaic of the images, you may not want the script to generate the mosaic until all of the images are available. The processing script will not be able to verify that all of the data is available until all of the commands necessary to acquire the satellite data have returned a status of 0 indicating success.

If a command returns a status of 1, it typically means the user entered the command incorrectly and it will never run properly. A status of 2 is typically returned when data requested by a user is not yet available.

Setting return statuses for McIDAS-X applications became mandatory, beginning with McIDAS-X version 7.0 released in May 1996. Most McIDAS-X applications written before that upgrade do not yet contain the **Mccodeset** /**mccodeset** features.

Online command helps

The McIDAS-X command provides users with a quick way to get a listing of a command's structure and available options. The original text for these helps is stored in the source file for each command. You will use one of two template formats when creating online helps. The format for Fortran commands is shown below.

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

The similar help template for commands written in C is shown below.

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

The first line of each help contains the command name followed by two dashes (--). These dashes are used as flags to the system for the **Alt ?** help option described earlier in this chapter.

To learn how to build help files, see *Chapter 3, Getting Started in McIDAS-X*.

The McIDAS-X library

In the *Hello World* examples above, several function names are in **bold** type. These are examples of functions included in the McIDAS-X library. The library contains all the object code for the functions and subroutines that make up the McIDAS-X Application Program Interface (API).

The name of the McIDAS-X library is **`libmcidas.a`**.

API naming conventions

To provide easy recognition of McIDAS-X functions, the names of all new McIDAS-X functions begin with one of the prefixes below.

Prefix	**Description**

Mc	C-callable, API-level
mc	Fortran-callable, API-level
M0	C-callable, non-API
m0	Fortran-callable, non-API

The McIDAS-X philosophy regarding function prefixes is that once a function is released with an **Mc** or **mc** prefix, it is a stable member of the McIDAS-X library and will have neither its functionality nor calling sequence changed. The **M0** or **m0** prefix is assigned to:

- Functions that will eventually become **Mc/mc** prefixed, or
- Functions that are never intended to be called by applications

SSEC uses the **M0/m0** prefix while testing a routine's functionality and calling sequence. When the function is deemed stable, the prefix is changed to **Mc/mc**.

SSEC adopted this function prefix naming convention in the spring of 1995. Most of the routines currently found in the McIDAS-X library predate this naming convention. As SSEC modifies existing software, it will make every effort to change the names of older functions where practical.

If new functions are written to replace older, non-prefixed functions, SSEC's policy for removing the old functions from McIDAS-X is as follows:

- All references to the old function will be removed from all applications in the McIDAS-X packages.
- The old function will be moved to a compatibility library.

Interface documentation block

SSEC has also developed a standard interface documentation block template that you should use for all new McIDAS-X library functions. This format is used to generate the online API function documentation distributed with the McIDAS-X software.

The table below lists and defines each field in the documentation block. More information follows along with examples from the **mchmstostr** function. If a function doesn't require a description for every field, you should still include all fields in the documentation block and fill the unused fields with the word *none*.

Field	Description
Name	name of the function
Interface	details of the function's calling sequence and return type
Input	variables in the calling sequence that are input only
Input/Output	variables in the calling sequence used for both input and output
Output	variables in the calling sequence that are output only
Return values	possible function return values and what they mean
Remarks	information about a function that may be useful to a programmer
Categories	keywords identifying which subsystem this function operates on

Name

This field is usually a one-line description of what the function does. The example below is taken from the function **mchmstostr**.

```
Name: mchmstostr - Converts a time to a character string
```

Interface

This field describes the function type value returned, such as float, integer, or subroutine. It also contains any include files associated with the function, and the type and size of each parameter in the calling sequence.

```
Interface: integer function mchmstostr (integer hms, integer format, character*(*) string)
```

Input

This field describes each of the input parameters to a function. It should include any discussion of expected formats and units.

```
Input: hms - Time in the form hhmmss. format - Output format desired for the string.
```

Input and Output

This field describes parameters that pass input to a function and have their values modified within the function. It is important to describe the state of these parameters both on input and output. Since this example doesn't contain parameters in this field, the entry looks like this:

```
Input and Output: none
```

Output

This field describes each of the output parameters to a function. The description should include any discussion of expected formats and units.

```
Output: string - Destination character string.
```

Return values

This field describes the possible values that can be returned by this function. The McIDAS-X convention for return values is described below.

- If a function has only one successful return value, it should be 0.
- If a function has multiple successful return values, it should return positive numbers.
- If a function fails, it should return negative numbers.

A function should have as many unique failure return values as ways the function can fail, as shown in the example below.

```
Return values: 0 - Success. -1 - Invalid value for hms. -2 - Invalid value for format. -4
- Destination string is not big enough.
```

Remarks

This field describes additional information that is necessary or useful to a programmer. For example, it would be helpful for a programmer to know:

- If this function allocates memory, requiring the memory to be freed when it's no longer used
- If this function must call another function to initialize the environment before it can be used
- If this function has size limits or *infinities*

Other useful information may include references to:

- A book where an algorithm used in a function was found
- Other functions in the library having similar features

An example of this field for **mchmstostr** is shown below.

```
Remarks: If the input value for hms is 23444 then: form string 1 02:34:44Z 2 02:34:44 3
02:34:44UTC 4 02:34:44 Z 5 02:34:44 UTC 6 2:34:44
```

Categories

This field is used to generate the cross reference list for the online API function documentation. The categories available in McIDAS-X are:

calibration	converter	day/time	display
event	file	graphic	grid
image	ingest/decode	met/science	navigation
point	text	sys_config	system
user_interface	utility		

Be careful not to assign more categories than needed to a function. For example, the McIDAS-X library function to display a contour graphic of a grid is **mcgrdcon**. The logical categories for this routine are *graphic* and *grid*. The function **mcgget** retrieves a grid, so the logical category for this routine is *grid*. Even if that grid is stored in a file, you wouldn't need to assign the category *file* since the fact that the grid is stored in a file is irrelevant. Use the *utility* category sparingly; it is intended only for functions that do not easily fit into any other category.

In **mchmstostr**, the categories field looks like this:

```
Categories: converter day/time
```

The next two pages contain the complete interface documentation block templates for both C and Fortran functions. Although the categories field contains the complete list of available entries, you will choose only those appropriate to your function. Text files containing the templates are available on the MUG Web Site.

Fortran template

```
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 *$ *$ Output: *$ none *$ param3 - description of it *$ *$ Return values:
*$ 0 - success *$ *$ Remarks: *$ Important use info, algorithm, etc. *$ *$ Categories: *$
grid *$ image *$ point *$ text *$ system *$ event *$ file *$ sys_config *$ display *$
graphic *$ utility *$ converter *$ day/time *$ calibration *$ navigation *$ ingest/decode
*$ met/science *$ user_interface INTEGER FUNCTION MCNAME (...) IMPLICIT NONE C ---
symbolic constants & shared data (...) C --- external functions (...) C --- local
variables (...) C --- initialized variables (...)
```

C template

```
/* THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED. */ /**** McIDAS Revision
History *** */ /**** McIDAS Revision History *** */ #include "mcidas.h" /* *$ Name: *$
Mcname - short description of purpose/use/etc *$ *$ Interface: *$ #include "mcidas.h" *$
*$ int *$ Mcname(int param1, char *param2, int *param3) *$ *$ Input: *$ none *$ param1 -
description of it *$ *$ Input and Output: *$ none *$ param2 - description of it *$ *$
Output: *$ none *$ param3 - description of it *$ *$ Return values: *$ 0 - success *$ *$
Remarks: *$ Important use info, algorithm, etc. *$ *$ Categories: *$ grid *$ image *$
point *$ text *$ system *$ event *$ file *$ sys_config *$ display *$ graphic *$ utility *$
converter *$ day/time *$ calibration *$ navigation *$ ingest/decode *$ met/science *$
user_interface */ int Mcname(...)
```

C and Fortran function interfaces

McIDAS-X applications and functions can be written in either Fortran or C. Because McIDAS has its roots in Fortran, not all functions in the McIDAS-X library have both C and Fortran calling interfaces.

While McIDAS-X attempts to accommodate both languages, C language programs will sometimes need to interface with Fortran-coded functions from the McIDAS-X library. When calling Fortran functions from C applications, you must keep in mind these four important differences between the languages.

- In Fortran, parameters are passed into functions by address; in C, they are passed by value.
- In Fortran, character strings are preallocated, blank-padded memory segments. The length of each string is passed into functions as additional hidden arguments. In C, strings are null-terminated memory segments that may be preallocated or dynamically allocated. Because the string is null-terminated, it is not necessary to explicitly pass the string length into the function.
- In Fortran, multi-dimensional arrays are stored in memory in column-major format. In C, they are stored in memory in row-major format.
- In Fortran, functions added to the library are given an underscore character (_) as a suffix. No suffix is appended to C functions when placed in the library.

Below are examples of how these language differences affect a C application calling a Fortran function. Also provided is an example of writing Fortran-callable routines in C and information about writing Dynamic Link Library modules.

Passing parameters into functions

The McIDAS-X Fortran function **iyxll** converts integer values from frame (tv) line and elements to single precision floating point values for latitude and longitude. The sample Fortran code fragment below calls this function.

```
integer tvlin, tvele integer onscreen real lat, lon tvlin = 100 tvele = 200 onscreen =
iyxll(tvlin, tvele, lat, lon)
```

To call the same function from a C application, your code would be similar to this:

```
#include "mcidas.h" Fint iyxll_ (Fint *, Fint *, Freal *, Freal *); /* function prototype
*/ Fint tvlin, tvele; Freal lat, lon; Fint onscreen; tvlin = 100; tvele = 200; onscreen=
iyxll_ (& tvlin, & tvele, & lat, & lon);
```

The special type declarations **`Fint`** and **`Freal`** are Fortran integer and real declarations defined in the include file **`mcidas.h`**. They are designed to provide platform-independent interfaces between Fortran and C applications.

Since parameters are passed into functions in C by value, you must reference the variables in the calling sequence with the ampersand (**&**) symbol. Calling **iyxll** by value, as shown below, would most likely result in a segmentation violation error.

```
onscreen = iyxll_ (tvlin, tvele, lat, lon);
```

Passing character strings into functions

The McIDAS-X library function **mdsvc**, which is coded in Fortran, returns the appropriate real-time MD file given a character string schema type and an integer value representing a Julian day. Below is a sample Fortran code fragment to call this function.

```
integer mdfile, day character*4 schema schema = 'ISFC' day = 96017 mdfile = mdsvc(schema,
day)
```

To call the same function from a C application, your code would be similar to this:

```
#include "mcidas.h" Fint mdsvc_ (char *, Fint *, FsLen); /* function prototype */ Fint
mdfile, day; FsLen schema_length; char *schema; day = 96017 schema = strdup ("ISFC");
schema_length = (FsLen) strlen (schema); mdfile = mdsvc_ (schema, &day, schema_length);
```

Note that the variable schema is passed in with no ampersand character. The function expects the variable to be passed in by address and since *schema* is already a pointer, no ampersand is required.

Also note that the call to **mdsvc_** in C has an additional parameter, *schema_length*. This argument is passed by value to the Fortran function so the length of the string containing the schema type is known to **mdsvc** internally.

Storing arrays

The conflict of Fortran column-major arrays and C row-major arrays is difficult to resolve. Arrays are contiguous memory segments, no matter how many dimensions they have. The difference between the languages is in how the individual elements of the arrays are arranged and accessed, as described below.

If you declare a Fortran array like this:

```
integer array(2,3)
```

Then to reference the memory segments in increasing order, you will reference them like this:

```
1,1 2,1 1,2 2,2 1,3 2,3
```

If you declare a C array with the same dimensions:

```
int array[2][3]
```

Then to reference the memory segments in increasing order, you will reference them as shown below. They are shown as one-based values for comparison to the Fortran array even though the C language is actually zero-based.

```
1,1 1,2 1,3 2,1 2,2 2,3
```

When accessing data stored in multi-dimensional arrays, it is difficult to recognize if an array is stored in Fortran or C. Since most McIDAS-X applications were historically written in Fortran, multi-dimensional data is usually stored in column-major format. To address values for column-major data in C, dimension your arrays as one-dimensional and use the conversion equations below to index to the appropriate location.

The indexing equation for a column-major (Fortran) array is as follows, assuming 1-based indexing:

index = ((*column*-1)**Number Of Rows*) + *row*

The indexing equation for a row-major (C) array is as follows, assuming 0-based indexing:

index = (*row***Number Of Columns*) + *column*

Writing Fortran-callable routines in C

Now that you understand how the C and Fortran languages treat different elements of parameter passing into functions, you need to know how to write Fortran-callable functions in the C language. The example below builds the calling sequence for a unit conversion routine that takes an integer value and unit types and returns a floating point number. The function also returns an integer type for a status code.

In C, the prototype and function would look like this:

```
int convert (int, char *, char *, float *); /* function prototype */ int convert (int
input, char *input_units, char *output_units, float *output) { : : }
```

The entire source code for the Fortran-callable version of this function would appear as follows. The important concepts are **bolded**.

```
Fint convert_ (Fint * input, char * input_units, char * output_units, Freal * output,
FsLen input_len, FsLen output_len) { char *c_input_units; /* C string representation * of
input units */ char *c_output_units; /* C string representation * of output units */ int
rc; /* convert() function return value */ /* convert the Fortran character strings into C
strings */ c_input_units = fsalloc (input_units, input_len); c_output_units = fsalloc
(output_units, output_len); /* call the C callable version of convert */ rc = convert
((int) * input, c_input_units, c_output_units, (float *) output); /* free up the memory
for the C strings */ free (c_input_units); free (c_output_units); /* return from the
function */ return ((Fint) rc); }
```

Note that the Fortran-callable function declaration explicitly includes the underscore character (_). The McIDAS-X library function **fsalloc** converts Fortran strings to C strings. The function **strtofs** converts C strings to Fortran if you need to pass character string output back to the Fortran program that called this function jacket. Be sure to call the C function with the appropriate data types.

Writing shell scripts with McIDAS-X commands

A shell script is a program containing a set of executable commands. Shell scripts are useful for running a series of individual McIDAS-X commands outside of McIDAS-X. Generally, you must invoke the McIDAS-X resident program, **mcenv**, before running a series of McIDAS-X commands.

For information about writing shell scripts in McIDAS-X, see *Appendix H, Running Commands Outside a McIDAS-X Session*, in the *McIDAS User's Guide*.

Debugging McIDAS-X applications

It's a fact of life that programs do not always behave as expected. Therefore, you should implement some type of debugging strategy to isolate and repair errors. This section describes the tools available to help you debug applications. McIDAS-X provides two facilities to aid your search for the elusive bug. Additionally, Unix systems provide a variety of symbolic debugging tools, the most common of which is *dbx* or *gdb*.

Print statements

In McIDAS-X, the simplest debugging aide is **ddest**/**Mcdprintf**. These functions allow applications to print hidden statements when the third character in the DEV= global keyword is C. Strategically placing debug print statements in your code is an excellent way to identify invalid values or corrupted data in disk files. Use debug messages prudently, however, and avoid putting them inside large loops that may generate thousands of lines of useless messages.

For more information on **ddest** and **Mcdprintf**, see the section titled *Text messages* in *Chapter 4, McIDAS-X Utilities*.

M0ASSERT

Another debug facility provided in McIDAS-X is the C language development tool, M0ASSERT, which is used to trap severe errors in code, such as a memory overlap problem or trying to pass a NULL character pointer into a routine.

M0ASSERT is a macro that takes a boolean expression as input. If the expression resolves to false, a message is printed to **stderr** similar to this:

```
Assertion failed: source_file line nnn
```

where *source_file* is the name of the file and *nnn* is the line number in *source_file* where the failure occurred. After the message is printed, a call to **`exit(1)`** is made to end the program.

M0ASSERT can take two different forms based on compile-time options that you specify. The compile-time default for M0ASSERT is the NULL statement, meaning M0ASSERT does nothing. If you add the option -DM0ASSERT_ON to the compile statement, the preprocessor adds the assertion test.

M0ASSERT is not a substitute for standard error handling. Don't assume that capturing assertion failures will be activated as part of the production version of the software. Also, don't use M0ASSERT to test for conditions that, while rare, could indeed happen. The sample code fragment below demonstrates both good and bad uses of M0ASSERT.

```
int foo (const char *string) /* if the NULL pointer is passed into foo, you have a big
```

```
problem. This is a good example of M0ASSERT */ M0ASSERT(string != (const char *) NULL); /*
below is a bad use of M0ASSERT because memory allocation * failure is quite possible */
ptr = (char *) malloc (10); M0ASSERT (ptr != (char *) NULL);
```

For more information about assertions, read the excellent description provided in Chapter 2 of *Writing Solid Code* by Steve Maguire, Microsoft Press.

dbx

Most Unix-based systems include the symbolic debugger, **dbx**. To interactively debug McIDAS-X applications with **dbx**, you must perform a series of steps, which are necessary for these reasons:

- **dbx** gets the source file name to include in the debugging session from the executable created when the application is compiled. Because McIDAS-X file extensions are not the standard file extensions for the language, **dbx** will not recognize the original source file name.
- In general, McIDAS-X expects the data an application will use to be in the directory **~/mcidas/data**.
- Because McIDAS-X commands are started with a fork/exec, **dbx** is unable to directly link to the main of the application.

Perform the steps below to set up your environment for interactive debugging. For this description, assume that the development source code is in **~/mcidas/dev**, an application is in the source file **foo.pgm**, and it calls functions found in **suba.c** and **subb.for**.

1. Copy a version of your source files into the **~/mcidas/data** directory with the standard file extensions for the language.
   Type: **cp ~/mcidas/dev/foo.pgm ~/mcidas/data/foo.f**
   cp ~/mcidas/dev/suba.c ~/mcidas/data/suba.c
   cp ~/mcidas/dev/subb.for ~/mcidas/data/subb.f

2. Compile and link your functions and command with the **-g** option, which produces additional symbol table information necessary for the debugger. The resulting binary will be put in **~/mcidas/bin/foo.k**.

3. Change to the **~/mcidas/data** directory.

   Type: **cd ~/mcidas/data**

4. Verify that your **MCPATH** environment variable is set up appropriately. At the minimum, **MCPATH** should contain the directories **$HOME/mcidas/data** and **~/mcidas/data**.

5. Start an environment for McIDAS-X applications to run under.

   Type: **mcenv**

6. Start the debugger.

   Type: **dbx ../bin/foo.k**

7. From the dbx prompt, activate the source code.

   Type: **list main0**

8. Input the McIDAS command parameters as you would running the command from a unix prompt.

Programming do's and don'ts

This section describes some of the common issues that McIDAS-X developers encounter. It will offer suggestions for avoiding platform dependency problems and for planning new software development.

Byte flipping

When writing data access applications that may reside on other machines, you must be aware of the differences in byte ordering between big-endian and little-endian machines. These differences are described in the section titled *Conversion Utilities* in Chapter 4 of this manual. You will use the functions **swbyt2** and **swbyt4** to switch the byte ordering.

Floating point numbers

Different platforms may use different methods of representing floating point numbers in memory. Do not store floating point numbers to disk in the native format of the machine if the data may be used on other platforms. Instead, store floating point numbers as scaled integer values.

Julian day representation for the 21st century

Historically, the McIDAS-X standard for representing days was to use the Julian day with two digits representing the year of the century in the *yyddd* format. For example, 1 January 1997 is 97001. From the user's perspective, the convention of the 2-digit year continues into the 21st century. McIDAS-X continues to determine the century by making the most reasonable choice: 97 indicates 1997 (not 2097), whereas 03 indicates 2003 (not 1903). The user can also input the full 4-digit year in the *ccyyddd* format.

For disk files we use the convention for year of the number of years past 1900. For example, 97 indicates 1997, 103 is for 203. The McIDAS-X library contains many functions for handling this new Julian day format. These functions are described in the section titled *Conversion utilities* in Chapter 4 of this manual. Use the new representation not only in memory-based applications but also in file structures.

Think globally

When writing software that deals with geographic locations, verify that your code works properly and efficiently on all quadrants of the globe. For example, even if you think your subsystem will only be used for Northern Hemisphere data, make it robust enough to work in the Southern Hemisphere as well.

Fortran-specific do's and don'ts

The suggestions below will help you when programming in Fortran.

- Don't use a NULL string as a substitute for a blank character. For example, don't:

```
call sdest('',0)
```

  Instead, do:

```
call sdest(' ',0)
```

- Don't concatenate strings of indeterminate lengths in parameter lists.

```
integer function foo (string) character*(*) string character*80 tempstring c--- don't do
this call sdest('The value of string is '//string,0) c--- do this instead tempstring =
string call sdest('The value of string is '//tempstring,0)
```

- Put **`data`** statements after **`integer`**, **`parameter`**, **`common`** declarations.

- Use **`status=`** with an **`open`** statement. Use **`NEW`**, **`OLD`** or **`SCRATCH`**. The default for **`status=`** is usually **`UNKNOWN`**, which is defined as an implementation-dependent option. The sample code below shows you what to do.

```
character*(MAXPATHLENGTH) cfile integer unitno parameter (unitno = 2) logical file_there
.... inquire (file=cfile, exist=file_there) if (file_there)then open (unitno,file=cfile,
status='OLD') else open (unitno,file=cfile, status='NEW') endif
```

- Fortran compilers don't necessarily pad character literals out to column 72. For example, don't:

```
call sdest ('this is a very long string I think this will &run way off the end',0)
```

  Instead, do:

```
call sdest ('this is a very long string I think '// &'this will run way off the end',0)
```

- Avoid memory overlapping when assigning character variables. For example, don't:

```
string(2:20) = string(1:19)
```

- Verify that you are passing the correct-length floating point representations into functions. For example, if **`foo`** is defined as:

```
integer function foo (input1, input2) implicit none real input1 double precision input2
```

  Make certain your code passes arguments to **`foo`** like this:

```
ok = foo (7.0, 9.d0)
```

- Avoid manipulating parameters passed in as function arguments directly. Many systems cause segmentation violation errors if the value passed in is a constant. For example:

```
function foo (string) character*(*) string call mcupcase (string) ! convert to uppercase
```

If the call to **`foo`** looks like the one below, segmentation faults will usually occur because the **mcupcase** call cannot modify the constant **`Australia`**.

```
status = foo ('Australia')
```

- Don't write Fortran routines that return character strings as the function return code if you're planning to write C jackets for them.

  For example, don't:

  ```
  character*12 function foo (input)
  ```

  Instead, do:

  ```
  integer function foo (input, output_string)
  ```

- Don't embed function calls in concatenation of character strings. For example:

  ```
  string = 'The temperature is '//cfr(temp)
  ```

- If you're using formatted write statements, be sure the data type being printed matches the output declaration type. For example, the code below:

  ```
  REAL temp temp = 32.0 write (string, FMT='(a20,i4)')'The temperature is ',temp call
  sdest(string,0)
  ```

  Will print *The temperature is 0* on some platforms. Instead, use:

  ```
  write (string, FMT='(a20,f6.2)')'The temperature is ',temp
  ```

- Fortran-77 requires that dimension variables' data type be known before encountering an array declaration that uses them. For example, don't:

  ```
  function foo (array, nrow, ncol) implicit none real array(nrow, ncol) integer nrow, ncol
  ```

  Instead, do:

  ```
  function foo (array, nrow, ncol) implicit none integer nrow, ncol real array(nrow, ncol)
  ```

- The only recommended continuation character for column 6 is `&'.

- Instead of declaring character constants like this:

  ```
  character*12 string parameter (string = 'Hello World!')
  ```

  Declare them like this:

  ```
  character*(*) string parameter (string = 'Hello World!')
  ```

C-specific do's and don'ts

The suggestions below will help you when programming in C.

- Include **string.h**, not **memory.h** when using the **mem** * functions.

- Variables with **typedef** integral types can be a problem when printing, since **printf** requires you tell it the size of the value. The solution is to cast all ambiguous types to **`long`**, as shown below.

  ```
  uid_t u; /* usually unsigned */ size_t j; /* usually unsigned */ ssize_t k; /* signed
  size_t */ printf ("%ld %ld %ld\n", (long) u, (long) j, (long) k);
  ```

- Don't use **realloc** (**NULL**, *size*) to be the same as **malloc** (*size*). Some compilers don't allow this convention.

- Don't use **malloc** (**0**) or **realloc** (*p*, **0**). It may return either a pointer to a segment size of 0 or a NULL pointer. The result is implementation-dependent.

- Don't use **strlen** (*p*) where p may be NULL, as this causes segmentation violation errors on some platforms.

- Don't assume the return value from **sprintf** is the length of the resulting string.

Chapter 3

Getting Started in McIDAS-X

This chapter provides the information you need to develop applications programs under McIDAS-X. You'll learn how to:

- Set up your user account, including directories and libraries
- Compile and link your McIDAS-X code
- Make program helps and help files

This chapter is organized into these sections:

Setting up a user account

In this section, you will learn how to establish a user account, which you and your system administrator will set up. You must have a user account to develop code in McIDAS-X. This user account is your development arena. From it, you can create and debug your own code and datasets without impacting other McIDAS-X users or developers.

Your user account will consist of these components:

- A personal Unix account with login name and password, which your system administrator will create for you
- User account directories, which you will create to hold locally developed source and executable code
- A local library, which you will make to keep your local functions separate from McIDAS-X core functions

Creating your user account

Establishing a user account is a joint effort between you and your system administrator. The responsibilities of each are described below.

The system administrator's responsibilities

Your system administrator can create an individual Unix account for you, configuring the system to acknowledge you as a user and assigning basic privileges and directories.

In addition, the system administrator should install the McIDAS-X package on your workstation and place it in a separate account named **`mcidas`**. The **`mcidas`** account contains directories and files that you will use when developing code.

> See Chapter 1 of the *McIDAS User's Guide* for instructions on creating and configuring the **`mcidas`** and user accounts, and for installing McIDAS-X.

Your responsibilities

Verify that the **`.profile`** (Korn shell) or **`.cshrc`** (C shell) file located in your user account's home directory contains the proper McIDAS-X- and vendor-specific directories. The presence and order of these directories in the environment variable PATH define the searching order when you run any command or script.

Insert these McIDAS-X-specific directories in your PATH in this order:

- **`$HOME/mcidas/bin`**, which contains your local applications
- Any directory (**`~mclocal/mcidas/bin`** for example) containing other site-developed McIDAS-X commands on this workstation
- **`~mcidas/bin`**, which contains the SSEC core applications

Also insert any required vendor-specific directories. Verify that the files and directories are placed in your PATH either by you, your system administrator, or the vendor. You can do this in your **`.profile`** file by appending your PATH environment variable. These directories contain the C compiler, the Fortran compiler, and other tools needed to develop your code.

> See the *Preparing the mcidas Account* and *Preparing a User Account* sections in Chapter 1 of the *McIDAS User's Guide* for more information on modifying their PATH environment variables and a complete list of the required vendor-specific directories.

Directories

When developing McIDAS-X code, you will use the following two types of directories:

- **`mcidas`** account directories
- User account directories

The **`mcidas`** account directories are provided with the McIDAS-X software, but you must create your own user account directories. These directories are described below.

mcidas account directories

When the system administrator installs the McIDAS-X software in the **mcidas** user account, a series of predefined directories is automatically created. You don't need copies of these directories in your user account; however, knowing their contents can help you when developing code. They contain examples of source code and data files used with core applications.

There are two types of **mcidas** account directories:

- Package directories
- Installation directories

Package directories

Each version of McIDAS-X and other McIDAS packages (for example, McIDAS-XCD) builds its own set of directories. The names of the directories depend on the package name and version number. See the *Package Directories* section in Chapter 1 of the *McIDAS User's Guide* for the package directories in the current version of McIDAS-X.

Installation directories

The installation directories contain the files that are automatically generated when McIDAS-X is installed. The table below lists the directories and their contents.

Directory	Contents
~mcidas/bin	program executables
~mcidas/data	data files
~mcidas/help	help files
~mcidas/inc	include files
~mcidas/lib	libraries
~mcidas/man	man files for subroutines and functions
~mcidas/tcl	Tcl and Tk executables and libraries

User account directories

A user account directory is a directory that you create to hold your locally developed code. It can include your local library with its source code, applications with their source code, and data files.

The table below lists the suggested user account directories you should create before you begin writing any local code. The **$HOME/mcidas**, **$HOME/mcidas/data**, and **$HOME/mcidas/help** directories are created when mcidas is first started from the **$HOME** account.

Directory name	Contents
$HOME/mcidas	root directory for local McIDAS-X code
$HOME/mcidas/bin	local executables
$HOME/mcidas/data	local data files
$HOME/mcidas/src	source code for local functions and applications
$HOME/mcidas/help	helps for local applications
$HOME/mcidas/lib	local development library

Libraries

When you develop a new function for McIDAS-X, you must either add it to the McIDAS-X library or put it in a local library that you've created as part of your user account. The purpose of these libraries and some of the rules governing them are described below.

The McIDAS-X library

The McIDAS-X library, **`libmcidas.a`**, contains all the object code for the functions, subroutines, and Dynamic Link Modules that make up the McIDAS-X Application Program Interface (API). If you develop applications programs that link to **`libmcidas.a`**, be aware that the functions in this library are subject to change. SSEC reserves the right to modify or remove library functions.

You will receive a new library of functions and list of function changes with each McIDAS-X upgrade. The new library is in the file **`~mcidas/lib/libmcidas.a`**.

Local libraries

When developing code, you will undoubtedly produce your own set of functions to support your applications, and need a local library to keep those functions with their applications' source code. Placing your functions in a local library allows you to isolate your local functions from the McIDAS-X core functions, making them less susceptible to naming collisions and eliminating the need to regenerate your functions after a McIDAS-X upgrade. A local library is also useful for referencing functions that SSEC has moved to the compatibility library.

Use the suggested conventions below when naming functions that you'll put in your local library. An example of creating and accessing a local library is provided in the next section, *Compiling and linking your code*.

- Don't start your function names with the prefixes **mc**, **Mc**, **m0** and **M0**, which are reserved for functions produced by the SSEC programming staff. The **mc** and **Mc** functions are considered public APIs and will not change except under extraordinary circumstances. The **m0** and **M0** functions are considered private APIs and are more subject to change.
- Uniquely prefix your function names to avoid name collisions with functions in other libraries.

Compiling and linking your code

In this section, you will learn how to compile your McIDAS-X source files, link object files, and store the object files in a library. You'll use the **mccomp** script to compile and link your code, and the **mcar** script to create and update libraries.

Compiling source files

The script **mccomp** provides a platform-independent compilation and linking environment for McIDAS-X source files. By recognizing source file extensions, **mccomp** can understand some of the compile options needed by some source files. For example, if the source file extension is **.for**, **.fp**, or **.pgm**, **mccomp** calls the Fortran compiler.

The **mccomp** script has four options that you will frequently use when compiling source files.

Option	Description
-c	compiles only
-Dval	sets preprocessor values
-g	produces symbol table for debugging
-Idir	searches a list of directories for include files

For example, to compile the source file **program.pgm**, use the **mccomp** script shown below. This script creates the object file **program.o**. The next section, *Linking object files* describes how to create the McIDAS-X command **program.k**.

```
mccomp -c -I. -I/home/mcidas/inc program.pgm
```

Linking object files

Once the object file has been created, you must link the file with the necessary libraries to create the executable code. The **mccomp** script recognizes the platform's linking options, so you do not need to change link options when moving between platforms.

The table below lists three **mccomp** script options that you will frequently use for linking source files.

Option	Description
-L*dir*	links the command with the a directory containing the libraries
--l*library*	links the command with a specified library
-o*binary*	specifies a name for the command created

To link the object file **program.o** created in the previous example to create **program.k**, run the script below.

```
mccomp /home/mcidas/lib/main.o program.o -L/home/mcidas/lib -lmcidas -o program.k
```

Note that the script contains the object file **main.o**. McIDAS-X commands written in Fortran do not contain their own MAIN program. You must link the appropriate MAIN program from **main.o**.

See the section *Writing McIDAS-X applications* in *Chapter 2, Learning the Basics* for more information about the MAIN program.

Storing object files in a library

You may want to keep all your local functions in a local library. This makes software management much easier. Use the script **mcar** to put object files in a local library. The example below shows you how to compile the file **foo.for** and store the resulting object file in the library **libdev.a**.

1. Use **mccomp** to compile your function and create the object file, **foo.o**.

```
mccomp -c -I. -I/home/mcidas/inc foo.f
```

2. Move the object file to the library **libdev.a**.

```
mcar libdev.a foo.o
```

3. If you want to create the file **bar.k** to link with the function **foo**, use the **mccomp** script below. Assume the **bar** command is written in C so that it contains its own **main.o**.

```
mccomp bar.o -L. -L/home/mcidas/lib -ldev -lmcidas -o bar.k
```

If your projects contain several source files, you may want to use a project management utility such as **make** to compile your code. The Unix **make** utility uses a description file or **makefile** to construct a sequence of Unix commands that the Unix shell runs. This utility helps you generate your **mccomp** and **mcar** commands. If you are not familiar with the **make** utility, SSEC recommends the book *Managing Projects with Make* (O'Reilly, 1991).

Building Dynamic Link Libraries

Dynamic Link Library modules (DLLs) allow functions to be linked into an application at runtime as needed, rather than when the application is initially compiled and linked. McIDAS-X does not support dynamic link modules. Instead, it emulates dynamic linking at compile time by preprocessing the module to contain unique type-based names for the API functions and imbedded common blocks. The preprocessor program, **convdlm** (in **convdlm.fp**) automatically modifies the entry point names in a unique way to avoid duplication. This mechanism is normally used in McIDAS-X for the navigation and calibration subsystems. For this process to work,

- The API function declarations must be uppercase, for example, **INTEGER FUNCTION NVXINI(...)**
- The common block names must be uppercase, for example, **COMMON /TANC/ ...**
- These phrases should not occur anywhere else in the module, including within comments
- The source file names must be prefixed with *nvx* for navigation modules and *kbx* for calibration modules

Generating DLL subsystem functions

When adding a new dynamic link library to McIDAS-X, you must generate a new subsystem initialization function. For navigation, this function is named **nvprep.for** and for calibration it is named **kbprep.for**. This is most easily done using the **nav_init** script and **cal_init** scripts.

1. Generate new **nvprep.for** and **kbprep.for** functions so these functions can use the new navigation and calibration modules as well as the navigation and calibration in McIDAS-X. Note that **~mcidas/mcidas***version***/src** is the directory where the core McIDAS-X source can be found for the current version of McIDAS-X. For example, in version 2003, this file would be **~mcidas/mcidas2003/src**.

2. To generate **nvprep.for**, run the command below at the Unix prompt in the directory containing your new navigation modules.

```
~mcidas/mcidasversion/src/nav_init ~mcidas/mcidasversion/src/nvx*.dlm -o nvprep.for
```

   To generate **kbprep.for**, run the command below at the Unix prompt in the directory containing your new calibration modules.

```
~mcidas/mcidasversion/src/cal_init ~mcidas/mcidasversion/scr/kbx*.dlm kbx*.dlm -o
kbprep.for
```

3. To allow applications to access your new navigation types, recompile **nvprep.for** and all applications that use navigation. For new calibration types, recompile **kbprep.for** and all applications that use calibration. For ADDE applications, the calibration is usually done on the server, requiring relinking of the ADDE servers.

Additional suggestions for writing DLL modules

Consider the guidelines below when writing a calibration or navigation module. Most restrictions are related to preprocessing the routine with **convdlm**.

- Write the module in Fortran, since **convdlm** only runs against Fortran. Using a C module will require a manual modification of the entry point names.

- Write the code in uppercase. When **convdlm** was written, all the modules were in uppercase. The only recognized comment line begins with a C.
- Don't use the words SUBROUTINE, FUNCTION, or COMMON in comment lines or messages such as DDEST. Also, in these lines, don't enter the name of any subroutine, function, or common block in uppercase.
- Don't use the Fortran ENTRY statement; **convdlm** doesn't recognize it or handle it correctly.
- Don't embed a function call within another function call if both functions are in the module. **convdlm** can't 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:

```
CALL ASUB (BFUNC(10))
```

- Don't 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.
- Don't output text with SDEST or Fortran WRITE, for example. This causes problems with ADDE servers that send data through standard output. You can embed DDEST calls for debugging, but they will only be output with non-ADDE commands.

To look at **.f** files, run the process **convdlm** manually, since these files are automatically deleted during compiling. When compiling **.DLM**s on Unix, **convdlm** reads the **.DLM** file and outputs three **.f** files: **kbxtest.dlm** becomes **kbxtest1.f**, **kbxtest2.f**, and **kbxtest3.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 **.DLM**s.

To run **convdlm**, use: **convdlm** *filename*
For example: **convdlm kbxtext.dlm**

Making program helps

The McIDAS-X command HELP allows users to quickly see the online structure of a command's argument list. When you write a McIDAS-X applications program, you should include a block of comments describing the functionality of the application, its positional parameters and keywords, and other notable remarks. This block of comments is called a program help or simply the *help*. These text comments, which reside in help files, form the input to the HELP command. The help files included in the McIDAS-X release are stored in **~mcidas/help**.

Use the utility **mcmkhelp** to produce a help file from source code. This script reads a source file from standard input and writes a help template to standard output. For example, the script below creates a help file from the source file **bar.pgm** then stores the resulting help template in your own mcidas help directory.

```
mcmkhelp < bar.pgm > $HOME/mcidas/help/bar.hlp
```

For more information about the HELP command, see the *McIDAS User's Guide*. For more information about the HELP command format, see Chapter 2 in this manual.

Chapter 4

McIDAS-X Utilities

This chapter describes many of the McIDAS-X library functions that you will use when writing your applications programs. These utility functions perform a variety of common programming tasks, such as:

- Acquiring command line parameters and pointing device status
- Displaying text, images and graphics on McIDAS-X windows
- Determining the number of display frames
- Converting physical units such as kilometers to nautical miles
- Computing meteorological parameters such as potential temperature

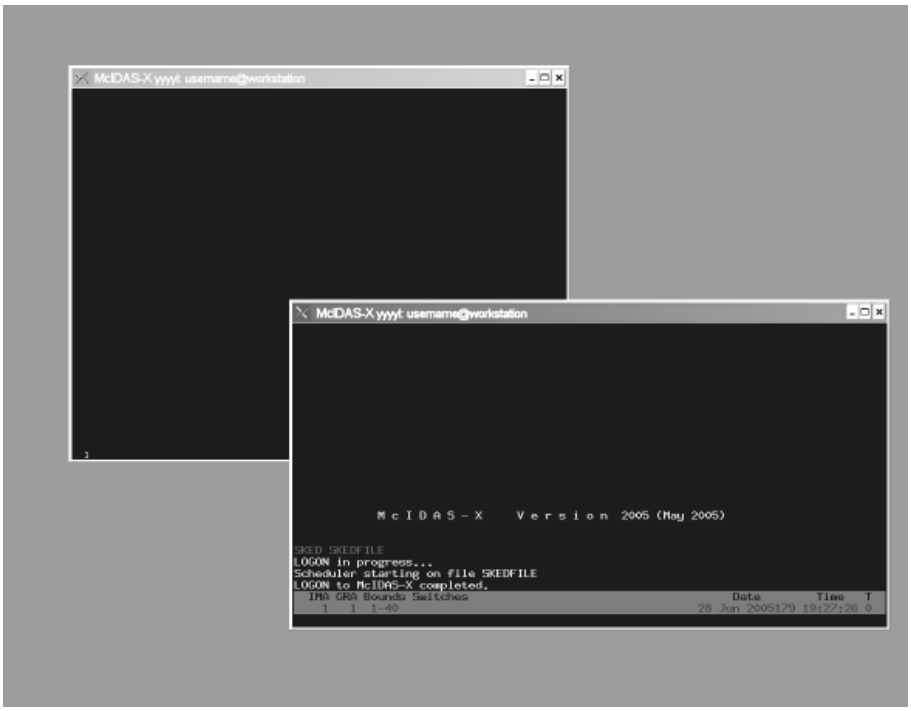
This chapter is organized into these sections:

Overview

The McIDAS-X library functions presented in this chapter are the building blocks for your applications programs. These functions abstract the hardware, such as the display, keyboard and pointing device, and provide routines for manipulating scalar quantities, such as dates and latitudes.

When a user starts a McIDAS-X session, a display similar to the one shown in Figure 4-1 appears on the screen. The window in the foreground is the McIDAS-X Text and Command Window. The window in the background is the McIDAS-X Image Window.

Figure 4-1. *At start-up, the McIDAS-X display contains the McIDAS-X Text and Command Window and the McIDAS-X Image Window.*



The visual output of your programs is displayed on these windows. Input is received from the keyboard via the command line and from the pointing device, or mouse, via the Image Window and cursor. Applications also make use of the terminal characteristics, the state of the McIDAS-X session and the application environment.

The functions described in this chapter are grouped by the tasks they perform:

- *User interface utilities*
  The user interface utilities include the functions for extracting numeric and text values from the command line and reporting the position and button state of the pointing device.
- *Display utilities*
  The display utilities include the functions for displaying text, images and graphics on the McIDAS-X windows, and obtaining information about the McIDAS-X display.
- *System utilities*
  The system utilities include functions for determining the terminal characteristics and the state of the McIDAS-X session, such as the number of color levels and frames, and for making applications interact predictably with the operating system, the McIDAS-X session and other applications such as error handling and file locks.
- *Conversion utilities*
  The conversion utilities include the functions for performing
  byte-level and character string manipulations on data buffers, and for converting day and time, latitude and longitude, and physical units such as speed and temperature.
- *Scientific utilities*
  The scientific utilities include the functions for computing meteorological parameters, such as mixing ratio and stability indices.

  For additional information about the functions described in this chapter, see the online man pages provided with the McIDAS-X software.

  For information about calling Fortran-coded functions from C, see the section titled *McIDAS developer overview* in *Chapter 2, Learning the Basics*.

User interface utilities

Users ultimately interact with McIDAS-X in one of two ways:

- The McIDAS-X command line
- The pointing device, or mouse

This section describes each part of the McIDAS-X command line, provides descriptions and examples of the command-line functions you can use in your programs, and defines the return status codes for these functions. This section also describes the standard pointing device for McIDAS-X and provides descriptions and examples of the functions available for reporting the state of the mouse buttons and changing the cursor's location, size, color and type.

For additional information about the functions described in this section, see the online man pages provided with the McIDAS-X software.

Command line

McIDAS-X applications are command-line driven. McIDAS-X has two basic command formats:

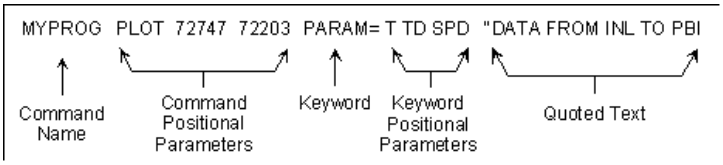
- Single-letter commands, which are run by simultaneously pressing the Alt key and the letter key, or by typing the letter and pressing Enter from the McIDAS-X Text and Command Window; for example, **Alt A**
- Multiple-letter commands containing positional parameters, keywords and quoted text

The multiple-letter command format is discussed in this section. The number of characters permitted in a command line is workstation dependent, although there is no practical limit.

A McIDAS-X command line may contain one or more of the following:

- command positional parameters
- keywords with positional parameters
- quoted text

A sample McIDAS-X command line is shown below.



Each command line value is separated by one or more blank spaces. Surround an individual parameter requiring blank spaces with single quotes (' ') so it is treated as one parameter.

Each part of the McIDAS-X command line is discussed below, followed by a description of the functions for extracting individual values from the command line and interrogating the structure of the command line.

Command positional parameters

Command positional parameters provide input to a command and must be entered in the exact order specified. Use them in commands that have options which the user must always enter. One advantage of positional parameters is minimizing the number of keystrokes a user types. Be careful not to negate this advantage by using so many positional parameters that the user can't remember them all.

Keywords

Like command positional parameters, keywords are used for entering command input. Unlike command positional parameters, they are optional for most McIDAS-X commands and can be entered in any order as long as they follow command positional parameters and precede quoted text.

Keyword parameters are often used to clarify commands with many complicated options. Although keywords can occur in any order, their positional parameters must be entered in the order indicated. Since users don't always specify all keyword positional parameters in a command, be sure to assign them reasonable defaults.

In addition to the keywords that an application has built into it, McIDAS-X has seven global keywords, which are common to all McIDAS-X commands:

- DEV= specifies the destination device of the text output generated by a command.

- FONT= specifies the font for drawing text on the McIDAS-X Image Window.
- MCC= specifies the data compression method.
- PAN= specifies a frame's panel number.
- TCOL= specifies the color for command text output in the text window.
- TWIN= specifies the destination text window to route the command output.
- VIRT= specifies a virtual graphics number to write graphical output.

You can retrieve the parameters for these global keywords in your applications, as long as you don't change their functionality.

For more information about the McIDAS-X global keywords, see the *McIDAS User's Guide*.

Quoted text

Quoted text input is most often used when strings entered by a user require blank spaces. Each application can contain only one quote string and it must be the last part of the command the user enters. Using quote fields in McIDAS-X commands is relatively easy from the McIDAS-X Text and Command Window; running McIDAS-X commands with quote strings from Unix shell prompts is more difficult. For this reason, the preferred method for entering strings with spaces in them from the command line is to use a positional or keyword parameter and surround the string with single quotes.

Functions for extracting values from a command line

The McIDAS-X library functions for extracting individual values from the command line are listed alphabetically in the table below. These functions have similar calling sequences.

C function	Fortran function	Description
Mccmddbl	**mccmddbl**	extracts a value as a double precision number
Mccmddhr	**mccmddhr**	extracts a time value as a double precision number in units of hours
Mccmddll	**mccmddll**	extracts a latitude or longitude value as a double precision number in units of degrees
Mccmdihr	**mccmdihr**	extracts a time value as an integer value in the hhmmss format
Mccmdill	**mccmdill**	extracts a latitude or longitude value as an integer value in the dddmmss format
Mccmdint	**mccmdint**	extracts a value as an integer
Mccmdiyd	**mccmdiyd**	extracts a day value as an integer value in the Julian day format ccyyddd
Mccmdquo	**mccmdquo**	extracts a character string value from the quote field
Mccmdstr	**mccmdstr**	extracts a character string value

You can use any of these functions, except **Mccmdquo**/**mccmdquo**, to extract information from any command or keyword positional parameter.

All of the functions in the table above, except **Mccmdquo**/**mccmdquo**, expect the following arguments:

- Keyword name, which can be a blank string (" ") if you are retrieving a command positional parameter
- Position number, 1-based
- Default value
- Variable to store the result

These functions, except **Mccmdstr**/**mccmdstr**, also expect the following additional information:

- Character string description of the requested value, which is printed if the user enters an incorrect syntax or a value out of range for this parameter
- Minimum and maximum acceptable value for this parameter; range checking can be inactivated by setting the minimum acceptable value greater than the maximum value

All the functions above perform similar operations when retrieving data from the command line, including:

- Verifying the user-entered value is in an appropriate syntax; you can type the command **ARGHELP** from the McIDAS-X Text and Command Window to receive a list of the acceptable syntaxes
- Verifying the user-entered value is within an acceptable range
- Returning a proper default value if the user didn't enter a value
- Verifying that there is enough room in the destination variable to contain the user-entered value
- Printing a message to standard error if the user enters an inappropriate value; error messages are discussed later in this chapter in the section titled *Text messages*
- Returning a status code to the calling routine

McIDAS-X uses a special syntax for identifying keyword names in these functions within an application. This syntax identifies a mandatory and an optional section, separated by a period. For example, if you use FRAME as a keyword in a command, it may be used in the above functions as FRA.ME to denote only the first three characters are mandatory. The table below shows some acceptable and unacceptable forms the user can enter.

Acceptable forms	Unacceptable forms
FRA=	FR=
FRAM=	FRAEM=
FRAMENUM=	

Keyword names can be letters or numbers, as long as the name is unique; that is, you can't use COL.OR and COL.UMN in the same command.

Below is a description of each function for extracting values from a command line, along with examples.

Sample TEST command

The sample output in the examples below is based on the following user-entered command string for the program TEST. If you see the line **goto 999** in the code fragments, it means: exit the command and return an error status.

```
TEST LIST 4 END=X 4:30 STA=03-APR-1996 12:30 PAR=T TD HEAD=WI 'Dane County' "Home of the
Black Wolf
```

Extracting a character string value

Use the **mccmdstr** function to extract a value from the command line as a character string. The code fragment below extracts the value from the first command positional parameter and puts it in the character string **option**.

```
character*12 default character*12 option default = ' ' ok = mccmdstr(' ', 1, default,
option) if (ok .lt. 0)then goto 999 endif
```

When the TEST command runs, the variable **option** will contain the string ***LIST***. If the user enters the string ABCDEFGHIJKLMN, an error occurs because **option** doesn't have room to store this string. The **mccmdstr** function prints the error message below and returns an error status.

```
TEST: first positional argument is too big --> ABCDEFGHIJKLMN TEST: Must be a character
string of no more that 12 chars.
```

The code below extracts the value of the second keyword positional parameter for the keyword HEAD and puts it in the character string **mnhead**.

```
character*12 default character*24 mnhead default = ' ' ok = mccmdstr('HEA.D', 2, default,
mnhead) if (ok .lt. 0)then goto 999 endif
```

When the TEST command runs, the variable **mnhead** will contain the string ***Dane County***. This string includes whitespace because the parameter is surrounded by single quotes on the command line. Note the mandatory and optional keyword components. This call will ignore a keyword named HEAP, while it will read a keyword named HEADER.

Extracting an integer value

Use the **mccmdint** function to extract a value from the command line as an integer value. The following code fragment extracts the value of the second positional parameter as an integer. The call to **mccmdint** also performs range checking to verify that the value entered by the user is within the range 1 to 9999. If the user doesn't enter a value for the second positional parameter, the default value of 1 is used.

```
parameter (MINFILE = 1, MAXFILE = 9999) integer fileno ok = mccmdint(' ', 2, 'File
Number', 1, MINFILE, MAXFILE, fileno) if (ok .lt. 0)then goto 999 endif
```

When the TEST command runs, the variable **fileno** will contain the value 4. If the user enters the value 10000 for the second positional parameter, the range allowed in this **mccmdint** call is exceeded. Thus, **mccmdint** prints the following error message and returns an error status.

```
TEST: Invalid File Number. TEST: 2nd positional argument is too big --> 10000 TEST: Must
be valid 'File Number' integer value within range 1 thru 9999.
```

Extracting time information

The functions to extract time information, **mccmddhr** and **mccmdihr**, accept input in a variety of formats. To get a list of the acceptable formats, type the command ARGHELP TIME from the McIDAS-X Text and Command Window.

The following code fragment extracts the value of the second keyword positional parameter as an integer for the keyword START. Note that the call to **mccmdihr** also performs range checking to verify that the value entered by the user falls within the range 0 to 23:59:59. This is not done automatically, so you can also use these functions to extract an interval of time such as -18:45.

```
integer starttime ok = mccmdihr('STA.RT', 2, 'Starting Time', 120000, 0, 235959,
starttime) if (ok .lt. 0)then goto 999 endif
```

When the TEST command runs, the variable **starttime** contains the value 123000. If the user doesn't enter a value for the second keyword positional parameter for the START keyword, the value returned in **starttime** is the default of 120000.

Be careful when setting ranges and default values for the **mccmdihr** function and using the returned values. The format expected is *hhmmss*. To specify 15:00:00 as the default for a command, you must enter 150000 as the fourth parameter in **mccmdihr**. If you specify 15 for the default, the value returned will be 00:00:15.

The following code fragment extracts the value of the second keyword positional parameter as a double precision number for the keyword END. The default value is five hours. Note that range checking is turned off in this example by setting the minimum value, 99.0, greater than the maximum value, -99.0.

```
double precision endtime ok = mccmddhr('END', 2, 'Ending Time', 5.0d0, 99.0d0, -99.0d0,
endtime) if (ok .lt. 0)then goto 999 endif
```

When the TEST command runs, the variable **endtime** will contain the value 4.5.

Extracting date information

The function to extract date information, **mccmdiyd**, accepts input in a variety of formats. To get a list of the acceptable formats, type the command ARGHELP DATE from the McIDAS-X Text and Command Window.

The following code fragment extracts the value of the first keyword positional parameter for the keyword START. This call sets the default to the current day and does not perform range checking.

```
integer currentday integer startday ok = mcgetday(currentday) ok = mccmdiyd('STA.RT', 1,
'Starting Day', currentday, 99, -99, startday) if (ok .lt. 0)then goto 999 endif
```

When the TEST command runs, the variable **startday** will contain the value 1996094.

Extracting a character string value from a quote field

Use the **mccmdquo** function sparingly when extracting a quote string that contains whitespace. The preferred method for retrieving parameters containing whitespace is to use the **mccmdstr** function and have the user enter the parameter with single quotes (') as shown in the **mccmdstr** section above.

The following code fragment extracts the value of the quote string.

```
character*256 quote ok = mccmdquo(quote) if (ok .lt. 0)then goto 999 endif
```

When the TEST command runs, the variable **quote** will contain the string ***Home of the Black Wolf***. If this field is missing from the user command, the variable is set to blank characters.

Functions for interrogating the command line structure

Sometimes applications need to get information from the command line beyond the actual values entered by the user. The functions described in the table below provide these services.

C function	Fortran function	Description
Mccmd	**mccmd**	returns the entire user command in one string

Mccmdkey	**mccmdkey**	verifies that the keywords entered by the user are valid
Mccmdnam	**mccmdnam**	returns a list of the keywords entered by the user
Mccmdnum	**mccmdnum**	returns the number of parameters entered for a particular keyword

Each function is described below, along with an example. The sample output is based on the user-entered command string for the program TEST, as shown in the section titled *Sample TEST command* above.

Verifying the validity of a keyword

Because of the complexity of the McIDAS-X command line, users will sometimes enter an incorrect keyword that is ignored and then not be able to determine why their command doesn't run properly. The **mccmdkey** function will verify that the keywords entered by the user are valid for the command. It compares the keywords the user enters with a list of acceptable keywords for a command and prints an error message if a discrepancy occurs. Additionally, **mccmdkey** checks your application to verify that it contains no ambiguous keywords; for example, COL.OR and COL.UMN.

The global keywords described earlier in this section are implicitly included in **mccmdkey**, so you don't need to include them in your list of keywords to be tested. Your cannot use the **mccmdkey** function if the keyword names in your command are not fixed; for example, cases where keyword names are related to data structures. Your program cannot predetermine the names the user may have entered; therefore, it cannot use **mccmdkey**.

The code fragment below shows how to validate the sample TEST command for the keywords END, FILE, PARAMETER, START and TITLE.

```
parameter (MAXKW = 5) character*12 validkw(MAXKW) data
validkw/'END','FIL.E','PAR.AMETER','STA.RT','TIT.LE'/ ok = mccmdkey(MAXKW, validkw) if (ok
.lt. 0)then goto 999 endif
```

If the user enters the additional keyword argument **COUNTRY=UK**, the TEST command will output the following line to the McIDAS-X Text and Command Window and exit.

```
TEST: Invalid command keywords: COUNTRY=UK
```

Note that it is the programmer's responsibility to keep the keyword list used in **mccmdkey** consistent with the keywords actually used by the application. This checking can be turned off by the user by setting **syskey** word 15 to a non-zero value.

Retrieving a list of user-entered keywords

Sometimes an application needs to know exactly which keywords the user entered. The **mccmdnam** function will return this information. The code fragment below shows how to retrieve the list of user-entered keywords.

```
parameter (MAXKW = 5) character*12 entered_kw(MAXKW) n_ent = mccmdnam (MAXKW, entered_kw)
if (n_ent .lt. 0)then goto 999 endif
```

When the TEST command runs, the variables in this code fragment will contain the following values:

```
n_ent = 4 entered_kw(1) = 'END' entered_kw(2) = 'STA' entered_kw(3) = 'PAR' entered_kw(4)
= 'TITLE'
```

Note that **mccmdkey** returns abbreviated values for the START and PARAMETERS keywords, since that is how they were entered on the command line.

Verifying the number of keyword parameters

When applications allow for multiple keyword positional parameters, you must be able to tell how many parameters the user actually included in the command. The **mccmdnum** function returns this information. The following code fragment extracts the number of keyword positional parameters that the user entered for the PARAMETER keyword.

```
integer n_parms n_parms = mccmdnum('PAR.AMETER')
```

When the sample TEST command runs, **n_parms** will contain the value 2.

To find out how many command positional parameters the user entered, the code fragment would look like this:

```
n_parms = mccmdnum(' ')
```

Status codes

The command-line functions all return status codes. A non-negative value indicates a successful return. A value greater than zero indicates a successful return, but with additional information such as the parameter returned is the default value. A value less than zero indicates a problem, for example in the formatting or range checking. The argument fetching status codes are defined in the table below.

The status codes use the format *abcd.* The value in the *a* position indicates the parameter's location; for example, the command line or string table. The values in the *b* and *c* positions indicate the form of the value; for example, time or angle. The value in the *d* position indicates the status.

Status code	**Definition**
[-]0bcd	argument comes from the default
[-]1bcd	argument comes from the command line
[-]2bcd	argument comes from the McIDAS-X string table
[-]a00d	character string argument
[-]a01d	quote field string argument
[-]a10d	integer argument
[-]a11d	integer hexadecimal argument
[-]a20d	double decimal argument
[-]a21d	double hexadecimal argument
[-]a30d	date argument
[-]a31d	current date argument
- *a32d*	year within date argument is invalid
- *a33d*	mon month within date argument is invalid
- *a34d*	mm month within date argument is invalid
- *a35d*	day of month (dd) within date argument is invalid
- *a36d*	day of year (ddd) within date argument is invalid
[-]a40d	integer time argument
[-]a41d	current integer time argument
- *a42d*	hours within integer time argument are invalid
- *a43d*	minutes within integer time argument are invalid
- *a44d*	seconds within integer time argument are invalid
[-]a45d	double time argument
[-]a46d	current double time argument
- *a47d*	hours within double time argument are invalid
- *a48d*	minutes within double time argument are invalid
- *a49d*	seconds within double time argument are invalid
[-]a50d	integer lat/lon argument

- *a52d*	degrees within integer lat/lon argument are invalid
- *a53d*	minutes within integer lat/lon argument are invalid
- *a54d*	seconds within integer lat/lon argument are invalid
[-] a55d	double lat/lon argument
- *a57d*	degrees within double lat/lon argument are invalid
- *a58d*	minutes within double lat/lon argument are invalid
- *a59d*	seconds within double lat/lon argument are invalid
[-] 90d	keyword status
abc0	argument is ok
- *abc1*	argument has an invalid character
- *abc2*	integer argument can't contain a fraction
- *abc3*	argument exceeds system limits for desired format
- *abc4*	out-of-range argument < given min
- *abc5*	out-of-range argument > given max

Pointing device

The standard pointing device is usually a three-button mouse. The leftmost button is used by the window manager and the middle and right buttons are used by the McIDAS-X mouse interface.

If you have a two-button mouse, the left button is used by the window manager and the right button performs the tasks normally performed by the right button on a three-button mouse. Press both buttons to perform the tasks normally performed by the middle mouse button. Check the mouse settings in your X-server configuration if you see unexpected behavior.

Users can interact with the McIDAS-X Image Window and McIDAS-X commands via the mouse-driven cursor. McIDAS-X defines the cursor's size, color and shape as long as the cursor resides on the McIDAS-X Image Window. If the cursor is moved outside the window, the cursor's size, color and shape revert to the workstation's settings.

The McIDAS-X function for interfacing with the mouse is **mcmoubtn**, which performs the following tasks:

- Reports the state of the mouse buttons when a mouse button event occurs and tracks any cursor movement while a command is running
- Reports the status of the keyboard toggles Alt G and Alt Q, which you can use as a signal for your program; for example, to signal a measuring or ending condition

You can use the McIDAS-X functions **putcur**, **mcgetcur**, **sizcur**, **colcur** and **typcur** in your applications to define the cursor's location, size, color and type. The function **mcgetcur** will return the cursor position of the cursor when your command starts.

Reporting the state of the mouse buttons

The **mcmoubtn** function reports the state of the mouse buttons as soon as a mouse button event occurs. Button events include the following:

- Mouse button was up, but is now down
- Mouse button is being held down
- Mouse button was down, but is now up

Applications specify which of the above events triggers a response via the first argument of the **mcmoubtn** function. Once the event occurs, the state of each mouse button and the cursor's line and element position in the McIDAS-X Image Window are returned to the calling application.

Calling **mcmoubtn** inherently causes the application to sleep or idle. Thus, you can call it within tight, intensive loops without fear of saturating the processor. If, at any time, the desired mouse event does not occur within a two-minute time span, **mcmoubtn** returns a timeout status to the calling application. The application is free to call **mcmoubtn** again if that two-minute time span is not considered unusual.

The code fragment below, from the McIDAS-X ZLM command, monitors the mouse event status.

```
parameter button_release=3 .... c---wait for mouse release 10
status=mcmoubtn(button_release,button(1),button(2),tvlin,tvele) c---end mouse button
sampling if((button(1).ne.0.and.button(2).ne.0).or.status.ne.0) then call beep(50,10) goto
100 c---on left mouse button press else if(button(1).ne.0) then ......
```

Note that the **mcmoubtn** function neither tests for nor protects against two applications using it at the same time. If several applications that call **mcmoubtn** are started concurrently, the results may be unpredictable.

Reporting the status of Alt G and Alt Q

The **mcmoubtn** function also indicates the status of the keyboard toggles Alt G and Alt Q, which are monitored by McIDAS-X similar to the mouse buttons. If you hold down the Alt key on the keyboard and press either the G or Q key, **mcmoubtn** will report that Alt G or Alt Q was pressed. These keyboard toggles are useful when you need to precisely position the cursor on the McIDAS-X Image Window and the act of clicking a mouse button may cause the mouse to move.

Defining cursor location, size, color and type

Use the functions below to change the appearance of the McIDAS-X cursor.

Fortran function	Description
putcur	moves the cursor to a specified location
sizcur	changes the cursor size
colcur	changes the cursor color
typcur	changes the cursor type; for example, crosshair or box

The values that these functions place into User Common are used by the **mcimage** program to define the appearance of the McIDAS-X cursor. The example below illustrates the use of these cursor functions.

```
integer line, element integer height, width integer color integer type .. C Define the new
location of the cursor line = 200 element = 351 C Define the new height and width; note:
these values must be ODD height = 23 width = 13 C Reassign the cursor to be displayed in
color level 3 color = 3 C Define the new type; 1=box, 2=cross-hair, 3=both, 4=solid, C
5=bullseye type = 3 C Now make the change: call putcur ( line, element ) call sizcur (
height, width ) call colcur ( color ) call typcur ( type ) ...
```

Display utilities

The McIDAS-X library provides many functions for displaying text on the McIDAS-X Text and Command Window and displaying images and graphics on the McIDAS-X Image Window.

- *Text* refers to three types of text messages that applications programs use to communicate status information to the user: standard text messages, error messages and debug messages.
- *Images* are information that can be represented as shades of gray in a two-dimensional matrix, such as satellite images, radar images or images derived from grids.
- *Graphics* are line drawings such as text, symbols or line segments; a plot of meteorological surface observations, for example, is made up of these elements.

This section provides descriptions and examples of the functions that you can use in your programs to display text, images and graphics on the McIDAS-X windows, and to obtain information about the McIDAS-X display.

For additional information about the functions described in this section, see the online man pages provided with the McIDAS-X software.

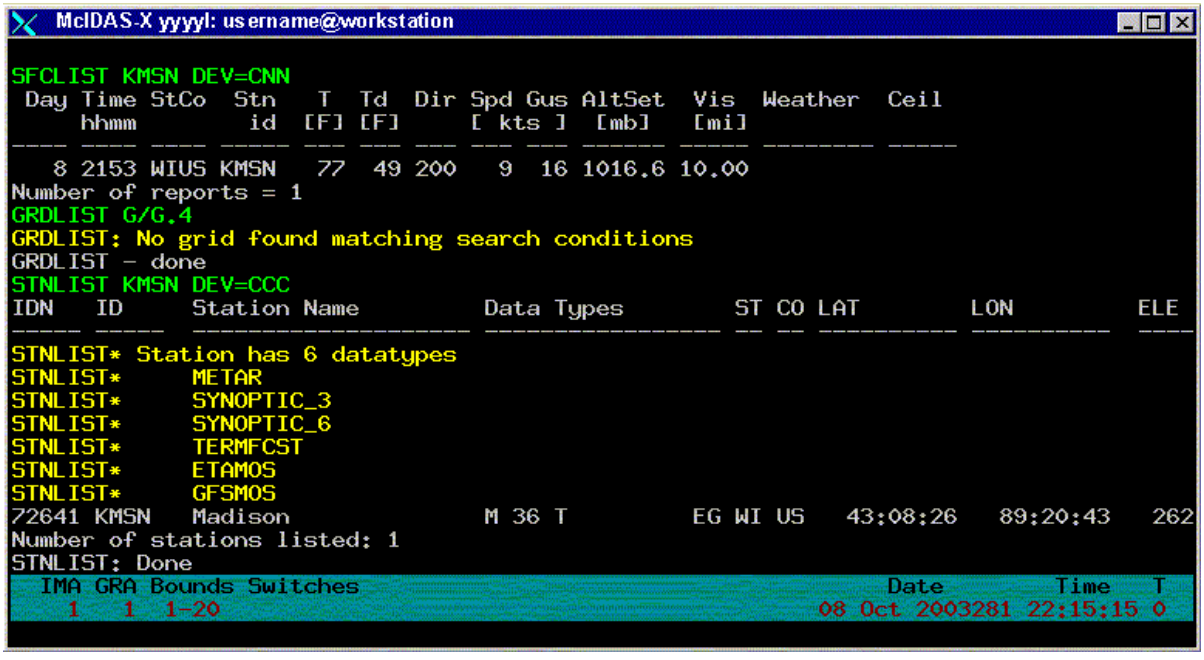
Text messages

McIDAS-X has three types of text messages that you can use in your applications:

- *Standard text* messages display normal text output, such as listings or user instructions, as shown with the McIDAS-X SFCLIST command in Figure 4-2 below.
- *Error* messages describe the failure of an application to perform a specific task. They are automatically prefixed with the program name followed by a colon, as shown with the McIDAS-X GRDLIST command in Figure 4-2.
- *Debug* messages contain details about the internal state of an application at runtime. Although debug messages aren't normally of interest to a user, they can identify why an application fails to perform a task. Debug messages are prefixed with the program name followed by an asterisk; see the McIDAS-X STNLIST command in Figure 4-2.

You should insert debug messages to help trace progress in applications, avoiding extraneous or misleading messages. Also avoid placing debug messages in code where thousands of lines of output could be generated. You will probably not have sufficient room on the McIDAS-X Text and Command Window to view that much output and even if the debug messages are suppressed, they will increase CPU usage unnecessarily.

Figure 4-2. *The McIDAS-X Text and Command Window displays standard, error and debug text messages*



Routing text

Text can be sent, or routed, to several destinations, as shown in the table below. Users enter the McIDAS-X global keyword, DEV, to specify the destination device of the text output generated by a command.

This DEV=	Sends the output to

option	
DEV=C	McIDAS-X Text and Command Window (default for standard and error messages)
DEV=P	printer
DEV=F	disk file; 80 characters per line
DEV=N	all output is discarded (default for debug messages)
DEV=T	text file

The user enters the disposition of the text messages as a series of three, contiguous disposition types in this order: *standard, error, debug*. For example, if the user enters DEV=CCC, all standard, error and debug messages are written on the McIDAS-X Text and Command Window. Entering DEV=CNN displays only the standard messages on the window; the error and debug messages are suppressed. The default is DEV=CCN, meaning the standard and error messages are written on the McIDAS-X Text and Command Window and the debug messages are suppressed.

For more information about McIDAS-X global keywords, see the *McIDAS User's Guide*.

Generating text

To display standard, error and debug messages in C or Fortran applications, use the library functions shown in the table below.

C function	Fortran function	Message type
Mcprintf	**sdest**	standard text
Mceprintf	**edest**	error
Mcdprintf	**ddest**	debug

The sample code below illustrates the use of each message type in a Fortran application.

```
subroutine main0 C $ ABCD - illustrates message disposition implicit none c --- external
functions integer mccmdint ! integer arg fetch c --- internal variables integer parameter
! 1st parameter c --- get the first positional parameter from the command line if(
mccmdint(' ',1,'1st Parameter',1,1,0,parameter).lt.0) & return call sdest('STANDARD --
First parameter =',parameter) call edest('ERROR -- First parameter =',parameter) call
ddest('DEBUG -- First parameter =',parameter) return end
```

The following table shows where text would be routed, based on the application, ABCD, presented in the code above.

Entering this command	Results in	Output goes to
ABCD 100	STANDARD -- First parameter =100 ABCD: ERROR -- First parameter =100	both messages go to the McIDAS-X Text and Command Window
ABCD 200 DEV=CCC	STANDARD -- First parameter =200 ABCD: ERROR -- First parameter =200 ABCD*DEBUG -- First parameter =200	all messages go to the McIDAS-X Text and Command Window
ABCD 300 DEV=PPF YY	STANDARD -- First parameter =300 ABCD: ERROR -- First parameter =300 ABCD*DEBUG -- First parameter =300	printer printer disk file named YY
ABCD 400 DEV=NNN		no output

If you don't have an integer value to be printed at the end of **sdest**, **edest** or **ddest** calls, specify zero as the second argument. These functions don't print an integer value of zero. To print a zero, format it into the string. The lines below show the use of a Fortran write to format a text message.

```
write(cline,fmt='(a18,1x,f9.2)')'the temperature is',temp call sdest(cline,0)
```

You can create the same formatted output with the following C call.

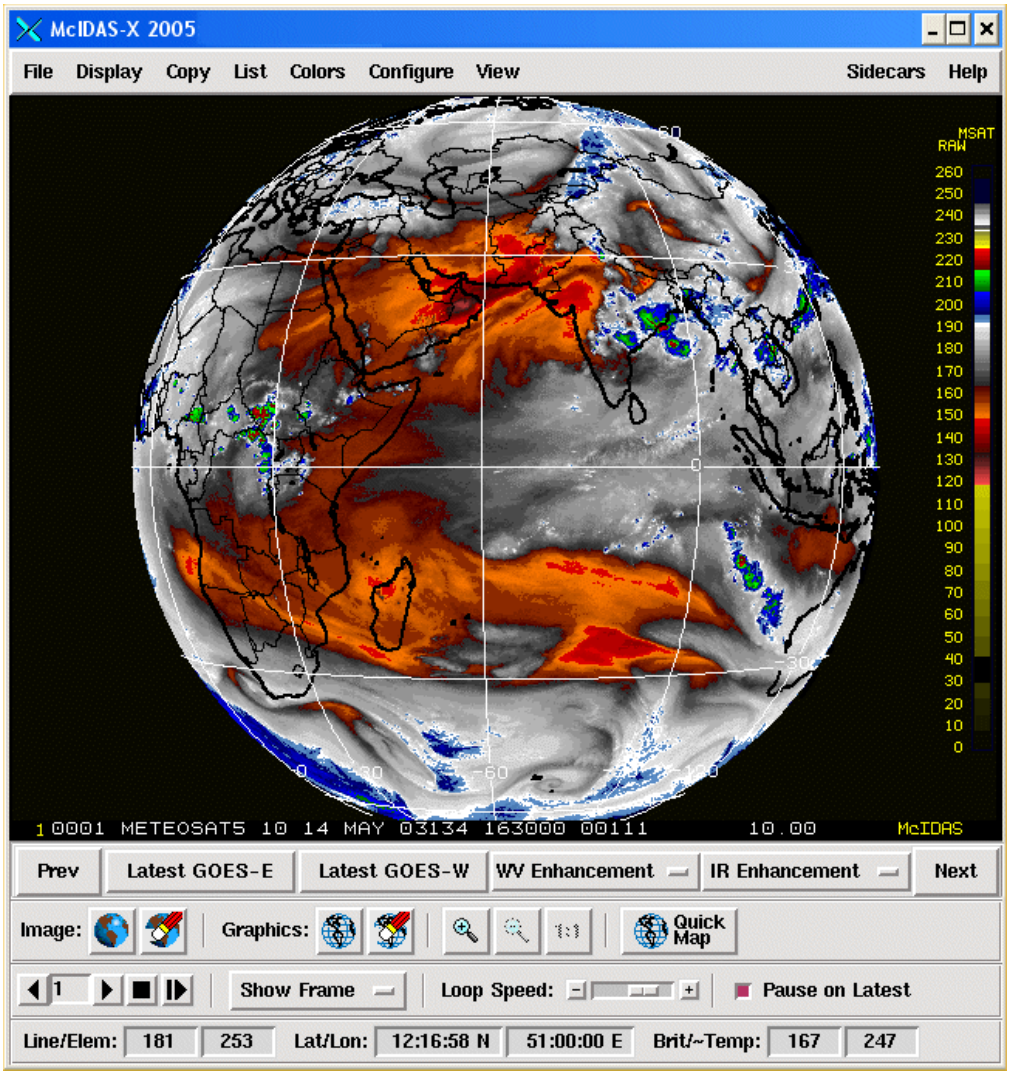
```
(void) Mcprintf ("the temperature is %9.2f\n ",temp);
```

Although **sdest**, **edest** and **ddest** automatically insert a new line when they are called, you must explicitly include the new line character, **\n**, in applications calling **Mcprintf**, **Mceprintf** and **Mcdprintf**.

Images

An image is information that is usually represented as shades of gray in a two-dimensional matrix, such as a satellite image, a radar image or an image derived from grids. Images are displayed on the McIDAS-X Image Window, as shown in Figure 4-3.

Figure 4-3. *The McIDAS-X Image Window displays McIDAS-X-generated images.*



You will use the **mcline** function to write a line of image data into a frame object for display on the McIDAS-X Image Window. A frame object is a memory-based collection of information that completely describes the contents and appearance of a frame to the **mcimage** process, which creates a visible picture. A frame object contains the actual image data, along with navigation and color enhancement information. Frame objects are stored in McIDAS-X shared memory. The number and size of the frame objects allowed per McIDAS-X session are user-configurable.

The sample code below illustrates how to use **mcline** to write a frame object. Note that the **mcline** function writes an entire line. Use **mclineseg** to write only a partial line.

```
integer array(1000) . . . C --- get the image frame number frame = luc(51) C --- get the
size of the frame status = mcfsize(frame, num_lines, num_elems) if( status.lt.0 ) then
call edest('Invalid image frame number=',frame) return endif C --- fill the output array
with a bounded gray scale do 10 elem = 1,num_elems 10 array(elem) = MOD( elem-1, 255 ) C -
```

```
-- write array to the frame object do 100 line = 1,num_lines 100 call mcline( frame, line,
array )
```

The output from this code displays a series of black-to-white gray shades. It may not map to the McIDAS-X Image Window quite as you expect, however. McIDAS-X applications envision the McIDAS-X Image Window as being *n* lines by *m* elements, and capable of displaying 256 colors. Each element in the array holds a 1-byte value (0 to 255) representing a brightness value.

McIDAS-X applications use the **mcline** function to write image data, or brightness levels, into a frame object. The **mcimage** program then takes this 8-bit image data and maps it into the McIDAS-X Image Window. Because hardware configurations impose limitations on the number of displayable brightness levels, the image data in the frame object must pass through a filter that maps the brightness levels into the range of display levels. Consequently, the image brightness levels in the frame object rarely correspond one-to-one with the display levels in the McIDAS-X Image Window.

- For more information about McIDAS-X image display characteristics, see the McIDAS-X TERM command with the SCALE option in the *McIDAS User's Guide*. Also see the section titled *Displaying Images and Graphics* in the Introduction chapter of the McIDAS User's Guide.
- For more information about McIDAS-X shared memory and frame objects, see the section titled *Shared memory* in *Chapter 2, Learning the Basics*.

Graphics

McIDAS-X graphics appear to the user as text, symbols and line segments drawn in color on the McIDAS-X Image Window. Though displayed as part of the same frame object as images, graphics are manipulated through a separate, vector-based API. Vector-based means that graphics are produced by an imaginary drawing pen that can be moved about on the frame by function calls. The pen traces a line segment of the specified color as it moves, and the graphics subsystem converts this into the appropriate pixels and places it in the frame object. McIDAS-X also provides a simple interface for writing text and numbers as vector graphics.

This section describes how to use the graphics subsystem, providing both basic and advanced McIDAS-X graphics techniques. An extended example, GRAF.PGM, and the output it produces, accompanies much of the discussion. The listing for GRAF.PGM appears later in this section.

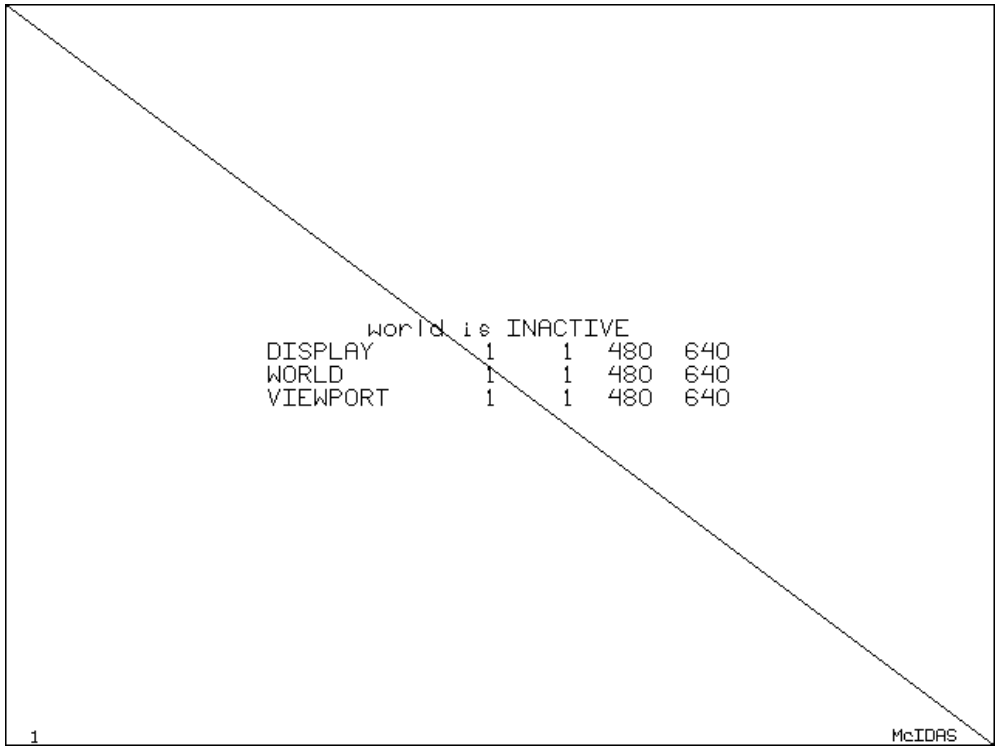
The table below alphabetically lists the McIDAS-X graphics functions used on the following pages. Additional graphics functions for displaying wind barbs, wind vectors, and weather symbols on a frame are provided later in this section.

Fortran function	Description
dshoff	turns the dash mode off
dshon	turns the dash mode on
endplt	closes or binds off the graphics frame
enpt	flushes the graphics buffer
initpl	initializes the graphics package to write to a frame object
newplt	erases the current graphics frame
page	resizes the viewport or world
plot	draws a line
pltdig	writes an integer number
qgdash	returns the current dash mode
qscale	returns the current scaling mode
sclhgt	converts text height from world to frame coordinates; use only if scaling is turned on; does not reduce the height of the text
scloff	turns scaling off; subsequent graphics calls use frame coordinates
sclon	turns scaling on; subsequent graphics calls use world coordinates
sclpnt	converts a point from world to frame coordinates
wrtext	writes a text string

Basic McIDAS-X graphics techniques

This section describes the techniques that you will most often use when working with the McIDAS-X graphics subsystem. Figure 4-4 below shows a graphic produced by the sample GRAF application when run from a McIDAS-X session with no arguments and a frame size of 480 lines by 640 elements.

Figure 4-4. *This GRAF.PGM output is the result of a McIDAS-X session run with no arguments and a frame size of 480 lines by 640 elements.*



To produce such a graphic, an application must do the following:

- Initialize the graphics subsystem
- Draw lines
- Write text and numbers
- Flush the graphics subsystem's buffers
- Close, or bind off, the graphics subsystem

These procedures are described below.

Initializing the graphics subsystem

Before you can use the McIDAS-X graphics subsystem, you must initialize it with a call to **initpl** as shown in the following code fragment from GRAF.PGM, lines 88 to 93; **`frame`** is the number of the frame where the graphics will appear and **`width`** is the line width, in pixels. Specify zero to use the current frame and width.

```
C // Initialize the plot and fetch the color frame = luc(-1) call initpl( frame, width )
if( mccmdint('COL.OR', 1, 'Graphic Color', 3, 1, 3, & color ) .lt. 0 ) return
```

Drawing lines

All graphics objects are composed of line segments built using the **plot** function. The graphics subsystem lets you specify the following line segment attributes:

- Line color
- Line style, dash or solid, along with the dashing pattern
- Line width

Line color

The diagonal line in Figure 4-4 is generated by these lines of code (91 to 132) in GRAF.PGM:

```
call initpl( frame, width ) ... call plot( w_ullin, w_ulele, PEN_UP ) call plot( w_lrlin,
w_lrele, color ) call enpt
```

The second **plot** call draws the line in color level **color** from the previous cursor position to line **w_lrlin** and element **w_lrele**, here the lower-right corner of the frame. The previous position is always the one specified by the most recent **plot** call; the **PEN_UP** defined constant, with a value of zero, moves the pen from its previous, undefined location to the start of the line at (**w_ullin,w_ulele**) without drawing.

To use a particular graphics form repeatedly, you can write a routine to generate it at the desired location with desired attributes. A simple example from GRAF.PGM, lines 263 to 286, is shown below.

```
subroutine box( ullin, ulele, lrlin, lrele, color ) <...variable declarations deleted...>
call plot( ullin, ulele, PEN_UP ) call plot( ullin, lrele, color ) call plot( lrlin,
lrele, color ) call plot( lrlin, ulele, color ) call plot( ullin, ulele, color ) call enpt
...
```

This example draws rectangles on the display with a single call and makes applications easier to write and understand.

Dash mode

The dash mode is set by the **dshon** and **dshoff** functions. To determine the current mode at any time, call the **qgdash** function.

When a McIDAS-X session starts, the dash pattern is preset to 10 pixels on and 10 pixels off. You can change this pattern using the McIDAS-X GD command, either from the command line before running any graphics applications, or within an application using the **keyin** or **Mckeyin** functions. These values are stored in User Common words 47 to 49.

Lines 146 to 151 in GRAF.PGM draw a dashed box to outline the viewport, which is described later, using the calls below.

```
if( border .gt. 0 ) then call qgdash( old_dash ) call dshon call box( v_ullin, v_ulele,
v_lrlin, v_lrele, color ) if( old_dash .eq. 0 ) call dshoff end if
```

Though not strictly necessary in this context, this sequence also illustrates a helpful idiom: capturing an aspect of graphics status (here the dashing state **old_dash**), changing it to a known state (**dshon**) and then restoring the original state when done. Such behavior is especially important when writing library functions, since they should be careful to leave things the way they found them.

Line width

The line width is established when the graphics subsystem is initialized with **initpl**. To change the line width, you must close the current graphic and restart another, specifying a new width. Previously drawn graphics remain on the frame with the former line width and new graphics are drawn with the new line width.

Displaying text and numbers

The McIDAS-X library provides two graphics functions, **wrtext** and **pltdig**, for writing text and numbers on a graphics frame. When using these functions, you must designate the following:

- Height of the characters
- Color level
- Location for the output: upper-left justified for **wrtext** and lower-right justified for **pltdig**

Lines 361 to 386 in GRAF.PGM use **wrtext** to produce the alphanumeric characters shown in Figure 4-4 through its own **plttext** subroutine, as shown below.

```
C --- Determine the center point in frame coordinates. Because C --- sclpnt() always works
whether the world is active or not, C --- make this call only if scaling is actually on.
call qscale( sclstat ) if( sclstat .eq. 1 ) then call sclpnt( clin, cele, txtclin, txtcele
) else txtclin = clin txtcele = cele end if C --- Determine the length of the text message
in characters. C --- This information is then used, together with the actual C ---
heights, to compute the upper left corner in frame coordinates nchar = len_trim( text )
ullin = txtclin - txthgt/2 ulele = txtcele - txthgt*nchar/2 + txthgt/10 C --- Display the
text. Turn scaling off first (we have figured C --- the text height and location in frame
coordinates ourselves, C --- right?) but be sure to restore it to its original state C ---
according to the result 'sclstat' of the qscale() call above call scloff call wrtext(
ullin, ulele, txthgt, text, nchar, color ) if( sclstat .eq. 1 ) call sclon
```

For titles and labels, it is often easier to decide where text should be centered rather than where its left margin should be. The above code uses the number of characters **nchar** and height in pixels **txthgt** to compute the offsets in line and element from the center to the upper-left. The internal **plttext** subroutine encapsulates the logic and other computations to account for graphics scaling so GRAF.PGM generates a centered line of text with a single call, just as **box** draws a graphics object in a clear, abstract way.

The **pltdig** function is similar to **wrtext** except that it also allows the user to specify the number of spaces. If the value to be plotted contains fewer digits, the result is left-padded with zeros.

Flushing the buffers

Because the graphics subsystem is buffered, graphics may not appear as they are being drawn. Interactive applications may need to force graphics to appear at various stages in the process. To do this, you must enter the call below to flush the buffers without ending the application's ability to draw additional graphics.

```
call enpt
```

In the listing for GRAF.PGM, this is done at the end of the **box** and **plttext** subroutines (lines 286 and 388). All graphics drawn to that point become visible and the graphics package, including all graphics options the application may have set, remains active.

Binding off graphics

When you're done creating graphics, call the **endplt** function (line 224 in GRAF.PGM) to bind off the graphics as shown below.

```
call endplt
```

Calling **endplt** displays all graphics on the McIDAS-X Image Window and frees all associated resources. To create more graphics on this or other frames, you will have to call **initpl** again.

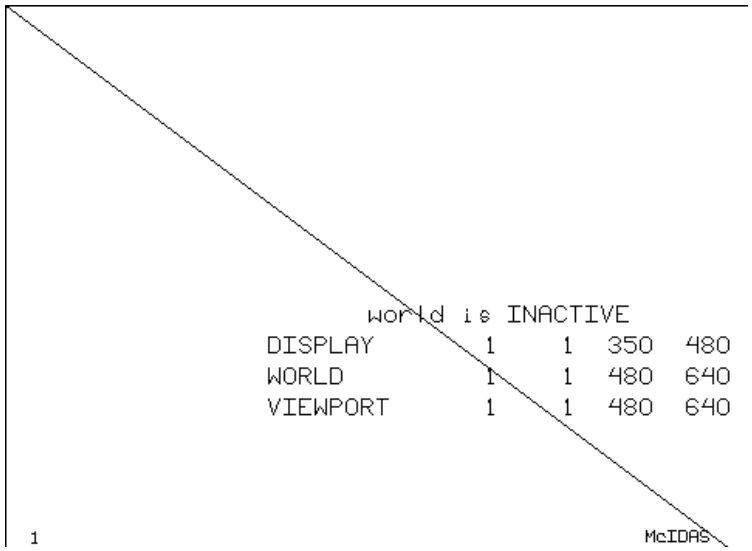
Advanced McIDAS-X graphics techniques

This section describes some advanced techniques that you can use with the McIDAS-X graphics subsystem. It builds upon the example and the McIDAS-X session started in the previous section, *Basic McIDAS-X graphics techniques*.

Using world coordinates

The example presented in the previous section positioned and drew all graphics in terms of pixel locations on the screen, assuming the typical frame size of 480 lines by 640 elements. However, McIDAS-X frames don't have a fixed size. If you restart the previous McIDAS-X session with a frame size of 350 by 480 and rerun GRAF.PGM, the display will look like Figure 4-5 below.

Figure 4-5. *This GRAF.PGM output is the result of a McIDAS-X session run with no arguments and a frame size of 350 lines by 480 elements.*



Although the text is still centered on (240,320), this is no longer the center of the frame. The text message describing the frame size gets its values from this line of code (line 195) in GRAF.PGM:

```
call mcfsize( frame, f_nlins, f_neles )
```

The upper-left corner is always (1,1) and the lower-right corner (**f_lrlin, f_lrele**) can be readily computed from the size.

The problem of positioning graphics in frames of various sizes remains, however. An obvious solution is for the application to create its graphics objects in some normal coordinate system in which the frame always occupies the same range, whatever its physical size in pixels. McIDAS-X refers to such systems as *scaled* or *world* coordinates because they let graphics applications run in their own little world, which is always the size that the application finds convenient. McIDAS-X supports the use of world coordinates with the **page**, **sclon**, **scloff**, **sclpnt** and **sclhgt** functions.

To use world coordinates properly, you must understand the concepts of *defining* and *activating* worlds. The **sclon** function activates world coordinates (scaling) and **scloff** deactivates them. You can use only one coordinate system, either world or frame, at a time. All functions that generate graphics using a *line* or *element* pixel coordinate (**plot**, **pltdig**, **wrtext**) use the current system, whether frame or world. Frame coordinates are determined entirely by the hardware and/or the McIDAS-X session; world coordinates, on the other hand, may be defined to be convenient for the application. The **initpl** function defines a default world of 480 lines by 640 pixels; this world may be redefined using the **page** function. Keep in mind that **page** also activates the world it defines, whereas **initpl** does not.

In Figure 4-5, the world is defined to be 480 by 640 by **initpl** but not activated, as noted on the display. If you instead enter this command:

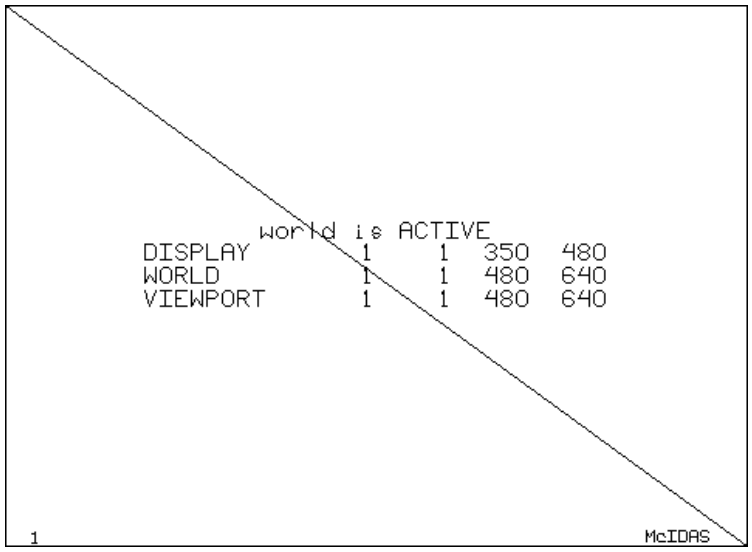
GRAF WORLD=1 1 480 640

GRAF.PGM looks at the four WORLD parameters and defines and activates a world using the code below (lines 99 to 117).

```
C // If the user has specified the necessary four arguments to C // WORLD= read them and
define the world; page() C // automatically activates it. C // The defaults are 1 1 480
640 but there is no significance C // to this here; the argfetchers will trigger a return
C // if the user doesn't actually enter a legal value for C // each of the four positions
if( mccmdnum('WORLD') .eq. 4 ) then if( mccmdint('WORLD',1, 'World upper left line', & 1,
1, 0, w_ullin ) .lt. 1100 ) return ...three more argfetches deleted... call page( w_ullin,
w_ulele, w_lrlin, w_lrele, SET_WORLD ) end if
```

The resulting output is shown in Figure 4-6 below.

Figure 4-6. *This GRAF.PGM output uses world coordinates to position the text.*



The diagonal now goes from corner to corner as in Figure 4-4 and the text is again centered even though the frame sizes are different. Making applications that can generate attractive, properly positioned output regardless of the physical size of the frame is the primary use of world coordinates and the **page** function. If a 480 by 640 world is suitable, you can just enter the calls below to activate **initpl**'s default world.

```
call initpl( frame, width ) call sclon
```

In general, you should use a world that is as large as or larger than your largest frame size to minimize truncation errors resulting from a single world pixel occupying multiple pixels of the frame.

One complication of world coordinates is the positioning and sizing of text. The **wrtext** function can be given positions and a height in either frame or world coordinates. If the world is smaller than the frame, the text is enlarged; if the world is larger than the frame, the text is not reduced. This prevents the text from becoming too small to read. The difficulty is that the application must be able to account for both cases when centering text. One solution is to do the positioning and output directly in frame coordinates even if scaling is on; the **sclpnt** and **sclhgt** functions allow explicit conversion of points and text heights from world to frame coordinates. The **plttext** subroutine in GRAF.PGM, lines 354 to 386, illustrates this technique as shown in the code samples below.

The actual text size in frame coordinates is determined first:

```
C --- Determine the actual (frame coordinates) height of the C --- text. (The use of local
variable 'txthgt' is needed to keep C --- plttext() from modifying its inputs.) txthgt =
hgt call sclhgt( txthgt )
```

The center point is then transformed to frame coordinates:

```
C --- Determine the center point in frame coordinates. Because C --- sclpnt() always works
whether the world is active or not, C --- make this call only if scaling is actually on.
```

```
call qscale( sclstat ) if( sclstat .eq. 1 ) then call sclpnt( clin, cele, txtclin, txtcele
) else txtclin = clin txtcele = cele end if
```

Then the upper-left corner is computed, again in frame coordinates:

```
C --- Determine the length of the text message in characters. C --- This information is
then used, together with the actual C --- heights, to compute the upper-left corner in
frame C --- coordinates. nchar = len_trim( text ) ullin = txtclin - txthgt/2 ulele =
txtcele - txthgt*nchar/2 + txthgt/10
```

Given that the position is in frame coordinates already, you must verify that scaling is off before calling **wrtext** and then be sure to restore the scaling to its original setting, whether off or on, before returning control to the caller:

```
C --- Display the text. Turn scaling off first (we have figured C --- the text height and
location in frame coordinates C --- ourselves, right?) but be sure to restore it to its C
--- original state according to the result 'sclstat' of the C --- qscale() call above.
call scloff call wrtext( ullin, ulele, txthgt, text, nchar, color ) if( sclstat .eq. 1 )
call sclon
```

In the above examples, you used world coordinates to make graphics applications independent of frame size. You can also change the orientation and origin of world coordinates as well as their extent.

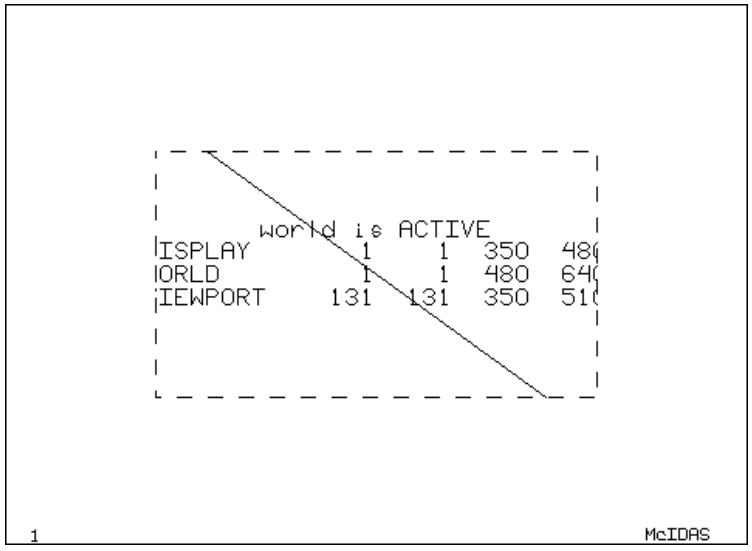
Some graphics such as scatter plots are easier to generate in an (*x,y*) coordinate system in which *x* increases leftward and *y* increases upward. The **page** function allows this, but there are two caveats. First, you should make your world domains as large as, or larger than, the frame to avoid truncation. Second, inverting the world by making either coordinate decrease, rather than increase, from the upper-left corner of the frame to the lower-right may cause confusion if you want to display text in world coordinates or use *viewports* (see below). The difficulty with text is that a specific, negative scale factor for **line** may result in text becoming very small or invisible.

For more information about frame coordinates, see the section titled *Coordinate systems* in *Chapter 2, Learning the Basics*.

Clipping and viewports

Sometimes it is desirable to limit graphics to a particular region of the frame. Consider Figure 4-7 below, which is identical to Figure 4-6 except the line segment is only drawn when it is more than 130 pixels, in world coordinates, from the edge of the world.

Figure 4-7. *This GRAF.PGM output shows a viewport, or clipping region.*



One solution is to determine the intersection of the diagonal with the 130-pixel border to get two new endpoints for the line segment. This algorithm is cumbersome because the diagonal may intersect with any two of the borders. The simple solution, supported by the McIDAS-X graphics subsystem, is to use a *viewport*. Also called a clipping region, a viewport defines a region of the frame outside of which graphics will not appear even if drawn. Like worlds, viewports are defined using **page**, but with a fifth argument of zero instead of one. Lines 139 to 151 in GRAF.PGM define the above viewport and draws its extent, as shown below.

```
if( mccmdint('BOR.DER', 1, 'Border Width', 0, 0, & (w_lrlin-w_ullin)/3, border ).lt. 0 )
return v_ullin = w_ullin + border v_ulele = w_ulele + border v_lrlin = w_lrlin - border
v_lrele = w_lrele - border call page( v_ullin, v_ulele, v_lrlin, v_lrele, SET_VIEW ) if(
border .gt. 0 ) then call qgdash( old_dash ) call dshon call box( v_ullin, v_ulele,
v_lrlin, v_lrele, color ) if( old_dash .eq. 0 ) call dshoff end if
```

Note that the viewport also clips text. The main use of viewports in McIDAS-X programming is to generate graphical output in regions set by the LINE and ELEM keywords, as is done by the McIDAS-X commands PTLIST and GRDDISP. Some users have written simple, frame-based graphical interfaces that set aside a portion of the frame as a toolbar and use a viewport to prevent meteorological data from plotting there when the rest of the frame is generated.

Listing for GRAF.PGM

```
1 C THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED. 2 3 C *** McIDAS Revision History *** 4 C *** McIDAS Revision History
*** 5 6 C ? GRAF -- Demo of basic McIDAS graphics 7 C ? GRAF <keywords> 8 C ? Parameters: 9 C ? (none) 10 C ? Keywords: 11 C ?
BORDER = | border width, in pixels 12 C ? COLOR = | graphics color level 13 C ? WORLD = | mn_lin mn_ele mx_lin mx_ele to define and
14 C ? | activate a world 15 C ? Remarks 16 C ? This demo nominally draws a diagonal line from 17 C ? the upper left to the lower
right of the display. 18 C ? 19 C ? If WORLD= arguments are not specified, the initpl() 20 C ? default world of 480 lines by 640
elements is defined 21 C ? but not activated. The diagonal line will be correct 22 C ? only if the display size is 480 by 640. If
WORLD= arguments 23 C ? are specified the world will be defined and activated and 24 C ? the diagonal correct regardless of world
size. 25 C ? 26 C ? BORDER= defines a viewport (clipping region) the indicated 27 C ? number of pixels (world coordinates) from the
edge of the 28 C ? world and outlines it with a dashed line. 29 C ? ---------- 30 subroutine main0 31 32 implicit NONE 33 34 C ---
Constants and shared variables 35 36 integer PEN_UP ! move pen without drawing 37 integer SET_VIEW ! define a viewport 38 integer
SET_WORLD ! define a world 39 parameter ( PEN_UP = 0, SET_VIEW = 0, SET_WORLD = 1) 40 41 C --- local variables 42 43 character*80
text ! output text buffer 44 45 integer border ! border (view port) width 46 integer cele ! text center element 47 integer clin !
text center line 48 integer color ! color level 49 integer frame ! frame number 50 integer f_lrele ! frame lower right element 51
integer f_lrlin ! frame lower right line 52 integer f_neles ! number of elements in frame 53 integer f_nlins ! number of lines in
frame 54 integer f_ulele ! frame upper left element 55 integer f_ullin ! frame upper left line 56 integer hgt ! text height 57
integer linspac ! spacing between text lines 58 integer old_dash ! dashing mode 59 integer sclstat ! scaling (world) mode 60 integer
v_lrele ! lower left viewport element 61 integer v_lrlin ! lower left viewport line 62 integer v_ulele ! upper right viewport
element 63 integer v_ullin ! upper right viewport line 64 integer w_lrele ! lower right world element 65 integer w_lrlin ! lower
right world line 66 integer w_ulele ! upper left world element 67 integer w_ullin ! upper left world line 68 69 real elew2f !
element frame:world ratio 70 real linw2f ! line frame:world ratio 71 72 C --- External functions 73 74 integer luc ! user common
peek 75 integer mccmdint ! integer argument fetch 76 integer mccmdnum ! number of args for keyword 77 78 C --- Initialized variables
79 80 integer width ! line width (session current) 81 82 data width / 0/ 83 84 C ----------------------------------------------------------
------------ 85 C INITIALIZE 86 C ------------------------------------------------------------------ 87 88 C // Initialize the plot and
fetch the color 89 90 frame = luc(-1) 91 call initpl( frame, width ) 92 if( mccmdint('COL.OR', 1, 'Graphic Color', 3, 1, 3, 93 &
color ) .lt. 0 ) return 94 95 C ------------------------------------------------------------------ 96 C DEFINE THE WORLD AND DRAW
BORDER AND DIAGONAL 97 C ------------------------------------------------------------------ 98 99 C // If the user has specified the
necessary four arguments to 100 C // WORLD= read them and define the world; page() 101 C // automatically activates it. 102 C // The
defaults are 1 1 480 640 but there is no significance 103 C // to this here; the argfetchers will trigger a return 104 C // if the
user doesn't actually enter a legal value for 105 C // each of the four positions 106 107 if( mccmdnum('WORLD') .eq. 4 ) then 108
if( mccmdint('WORLD',1, 'World upper left line', 109 & 1, 1, 0, w_ullin ) .lt. 1100 ) return 110 if( mccmdint('WORLD',2, 'World
upper left element', 111 & 1, 1, 0, w_ulele ) .lt. 1100 ) return 112 if( mccmdint('WORLD',3, 'World lower right line', 113 & 1, 1,
0, w_lrlin ) .lt. 1100 ) return 114 if( mccmdint('WORLD',4, 'World lower right element', 115 & 1, 1, 0, w_lrele ) .lt. 1100 ) return
116 call page( w_ullin, w_ulele, w_lrlin, w_lrele, SET_WORLD ) 117 end if 118 119 C // get the world definition. Note that there 120
C // is always a defined world, whether by initpl() or 121 C // page(). 122 123 call world( w_ullin, w_ulele, w_lrlin, w_lrele ) 124
125 C // draw the border and diagonal 126 127 call qgdash( old_dash ) 128 call dshoff 129 call box ( w_ullin, w_ulele, w_lrlin,
w_lrele, color ) 130 call plot( w_ullin, w_ulele, PEN_UP ) 131 call plot( w_lrlin, w_lrele, color ) 132 call enpt 133 if( old_dash
.ne. 0 ) call dshon 134 135 C ------------------------------------------------------------------ 136 C DEFINE THE VIEWPORT AND DRAW
BORDER 137 C ------------------------------------------------------------ 138 139 if( mccmdint('BOR.DER', 1, 'Border Width', 0,
0, 140 & (w_lrlin-w_ullin)/3, border ).lt. 0 ) return 141 v_ullin = w_ullin + border 142 v_ulele = w_ulele + border 143 v_lrlin =
w_lrlin - border 144 v_lrele = w_lrele - border 145 call page( v_ullin, v_ulele, v_lrlin, v_lrele, SET_VIEW ) 146 if( border .gt. 0
) then 147 call qgdash( old_dash ) 148 call dshon 149 call box( v_ullin, v_ulele, v_lrlin, v_lrele, color ) 150 if( old_dash .eq. 0
) call dshoff 151 end if 152 153 C ------------------------------------------------------------------ 154 C GENERATE AND DISPLAY THE
TEXT MESSAGES 155 C ------------------------------------------------------------ 156 157 C // Fetch the text height and use it
and the scale factors 158 C // to determine the separation between text lines. If the 159 C // line scale factor is less than one,
the text locations 160 C // will be moved closer together by the scaling but the 161 C // text height will be unchanged. So increase
the separation 162 C // between lines (in world coordinates) to prevent overwrite. 163 164 if( mccmdint('HGT', 1, 'Text Height', 10,
5, 20, 165 & hgt ) .lt. 0 ) return 166 call qscale( sclstat ) 167 call sclfac( linw2f, elew2f ) 168 if( sclstat.eq. 1 .and.
linw2f.lt. 1.0 ) then 169 linspac = 1.5*hgt / linw2f 170 else 171 linspac = 1.5*hgt 172 end if 173 174 C // Determine the center of
the topmost five lines. 175 C // Center the third line in the world; this will illustrate 176 C // the main use of world coordinates
to preserve centering 177 C // even when display sizes differ. 178 179 call world( w_ullin, w_ulele, w_lrlin, w_lrele ) 180 clin =
w_ullin + (w_lrlin-w_ullin)/2 - 2*linspac 181 cele = w_ulele + (w_lrele-w_ulele)/2 182 183 C // Display scaling status. 184 185 call
qscale(sclstat) 186 if( sclstat.eq. 1) then 187 text = 'world is ACTIVE' 188 else 189 text = 'world is INACTIVE' 190 end if 191 call
plttext( clin, cele, hgt, color, text ) 192 193 C // Get and display frame size. 194 195 call mcfsize( frame, f_nlins, f_neles ) 196
f_ullin = 1 197 f_ulele = 1 198 f_lrlin = f_nlins + f_ullin - 1 199 f_lrele = f_neles + f_ulele - 1 200 write(text,30) f_ullin,
f_ulele, f_lrlin, f_lrele 201 30 format('DISPLAY ',4I5) 202 clin = clin + linspac 203 call plttext( clin, cele, hgt, color, text )
204 205 C // Display world size. 206 207 call world( w_ullin, w_ulele, w_lrlin, w_lrele ) 208 write(text,32) w_ullin, w_ulele,
w_lrlin, w_lrele 209 32 format('WORLD ',4I5) 210 clin = clin + linspac 211 call plttext( clin, cele, hgt, color, text ) 212 213 C //
Display viewport size. 214 215 write(text,34) v_ullin, v_ulele, v_lrlin, v_lrele 216 34 format('VIEWPORT ',4I5) 217 clin = clin +
linspac 218 call plttext( clin, cele, hgt, color, text ) 219 220 C ------------------------------------------------------------------
221 C CLEAN-UP 222 C ------------------------------------------------------------ 223 224 call endplt 225 226 call
sdest('done',0) 227 return 228 end 229 230 231 ** Name: 232 ** box - draw a box on the display 233 ** 234 ** Interface: 235 **
subroutine 236 ** box(integer ullin, integer ulele, integer lrlin, 237 ** integer lrele, integer color) 238 ** 239 ** Input: 240 **
ullin - upper left line 241 ** ulele - upper left element 242 ** lrlin - lower right line 243 ** lrele - lower right element 244 **
color - color of box 245 ** 246 ** Input and Output: 247 ** none 248 ** 249 ** Output: 250 ** none 251 ** 252 ** Return values: 253
** none 254 ** 255 ** Remarks: 256 ** Input coordinates will be interpreted as either frame or 257 ** world system depending upon
whether the caller has activated 258 ** a world or not. 259 ** 260 ** Categories: 261 ** graphic 262 263 subroutine box( ullin,
ulele, lrlin, lrele, color ) 264 265 implicit NONE 266 267 C --- interface variables 268 269 integer ullin 270 integer ulele 271
integer lrlin 272 integer lrele 273 integer color 274 275 C --- symbolic constants and shared values 276 277 integer PEN_UP ! move
pen without drawing 278 parameter ( PEN_UP = 0) 279 280 281 call plot( ullin, ulele, PEN_UP ) 282 call plot( ullin, lrele, color )
283 call plot( lrlin, lrele, color ) 284 call plot( lrlin, ulele, color ) 285 call plot( ullin, ulele, color ) 286 call enpt 287 288
return 289 end 290 291 ** Name: 292 ** plttext - Center a line of text on the indicated location 293 ** 294 ** Interface: 295 **
subroutine 296 ** plttext(integer clin, integer cele, integer hgt, 297 ** integer color, character*(*) text) 298 ** 299 ** Input:
300 ** clin - center line 301 ** cele - center element 302 ** hgt - text height 303 ** color - text color 304 ** text - text string
305 ** 306 ** Input and Output: 307 ** none 308 ** 309 ** Output: 310 ** none 311 ** 312 ** Return values: 313 ** none 314 ** 315 **
Remarks: 316 ** The text will be centered on (clin,cele) in either world 317 ** or frame coordinates, depending upon whether the
```

```
caller has 318 ** activated a world or not. If a world is active, the actual 319 ** text will be enlarged if the world is smaller than the frame. 320 ** If the world is larger, the text size will be unchanged. 321 ** In order to keep the text centered, this routine does all 322 ** of its own scaling (using sclhgt and sclpnt) and actually 323 ** writes the text in frame coordinates. The display is then 324 ** returned to the initial state set by the caller. 325 ** 326 ** Categories: 327 ** graphic 328 329 subroutine plttext( clin, cele, hgt, color, text ) 330 331 implicit NONE 332 333 C --- interface variables 334 335 integer clin 336 integer cele 337 integer hgt 338 integer color 339 character*(*) text 340 341 C --- local variables 342 integer ullin ! upper left of text (frame) 343 integer ulele ! upper left of text (frame) 344 integer txthgt ! text height, frame coords 345 integer txtclin ! text center, frame coords 346 integer txtcele ! text center, frame coords 347 integer sclstat ! scaling status 348 integer nchar ! number of characters in text 349 350 C --- external functions 351 352 integer len_trim ! compute length of text 353 354 C --- Determine the actual (frame coordinates) height of the 355 C --- text. (The use of local variable 'txthgt' is needed to keep 356 C --- plttext()from modifying its inputs.) 357 358 txthgt = hgt 359 call sclhgt( txthgt ) 360 361 C --- Determine the center point in frame coordinates. Because 362 C --- sclpnt() always works whether the world is active or not, 363 C --- make this call only if scaling is actually on. 364 365 call qscale( sclstat ) 366 if( sclstat .eq. 1 ) then 367 call sclpnt( clin, cele, txtclin, txtcele ) 368 else 369 txtclin = clin 370 txtcele = cele 371 end if 372 373 C --- Determine the length of the text message in characters. 374 C --- This information is then used, together with the actual 375 C --- heights, to compute the upper left corner in frame coordinates. 376 nchar = len_trim( text ) 377 ullin = txtclin - txthgt/2 378 ulele = txtcele - txthgt*nchar/2 + txthgt/10 379 380 C --- Display the text. Turn scaling off first (we have figured 381 C --- the text height and location in frame coordinates ourselves, 382 C --- right?) but be sure to restore it to its original state 383 C --- according to the result 'sclstat' of the qscale() call above. 384 call scloff 385 call wrtext( ullin, ulele, txthgt, text, nchar, color ) 386 if( sclstat .eq. 1 ) call sclon 387 388 call enpt 389 390 return 391 end 392 393 394 ** Name: 395 ** sclfac - return the world to frame scale factors 396 ** 397 ** Interface: 398 ** subroutine 399 ** sclfac(real linw2f, real elew2f) 400 ** 401 ** Input: 402 ** none 403 ** 404 ** Input and Output: 405 ** none 406 ** 407 ** Output: 408 ** linw2f - number of frame lines per world line 409 ** elew2f - number of frame elements per world element 410 ** 411 ** Return values: 412 ** none 413 ** 414 ** Remarks: 415 ** An increment in world coordinates multiplied by the 416 ** appropriate scale factor will become an increment in 417 ** frame coordinates. 418 ** 419 ** This routine also illustrates a helpful design principle: 420 ** if your application needs access to the inner workings of 421 ** a subsystem (in this case a common block), write an access 422 ** routine to extend the API rather than just including the 423 ** common block in your application. 424 ** 425 ** Categories: 426 ** graphic 427 428 subroutine sclfac( linw2f, elew2f ) 429 430 implicit NONE 431 432 C --- interface variables 433 434 real linw2f 435 real elew2f 436 437 C --- constants and shared variables 438 439 integer ifrm ! frame number for graphics 440 integer iwdth ! line width 441 integer mnele ! upper left world element 442 integer mnlin ! upper left world line 443 integer mxele ! lower right world element 444 integer mxlin ! lower right world line 445 integer sclfg ! scaling on ? 446 real scele ! frame elems per world elem 447 real sclin ! frame lines per world line 448 449 common /M0FRACOM/ ifrm, iwdth, mxlin, mnlin, mxele, mnele, 450 & sclin, scele, sclfg 451 452 453 C // Implementation is trivial; just capture the common 454 C // block values in the formal arguments. 455 456 linw2f = sclin 457 elew2f = scele 458 459 return 460 end 461 462 ** Name: 463 ** world - return the world coordinates of display corners 464 ** 465 ** Interface: 466 ** subroutine 467 ** world(integer w_ullin, integer w_ulele, 468 ** integer w_lrlin, integer w_lrele ) 469 ** 470 ** Input: 471 ** none 472 ** 473 ** Input and Output: 474 ** none 475 ** 476 ** Output: 477 ** w_ullin - upper left line of world 478 ** w_ulele - upper left element of world 479 ** w_lrlin - lower right line of world 480 ** w_lrele - lower right element of world 481 ** 482 ** Return values: 483 ** none 484 ** 485 ** Remarks: 486 ** See sclfac() for a design note. 487 ** 488 ** Categories: 489 ** graphic 490 491 subroutine world( w_ullin, w_ulele, w_lrlin, w_lrele) 492 493 implicit NONE 494 495 C --- interface variables 496 497 integer w_ullin 498 integer w_ulele 499 integer w_lrlin 500 integer w_lrele 501 502 C --- constants and shared variables 503 504 integer ifrm ! frame number for graphics 505 integer iwdth ! line width 506 integer mnele ! upper left world element 507 integer mnlin ! upper left world line 508 integer mxele ! lower right world element 509 integer mxlin ! lower right world line 510 integer sclfg ! scaling on ? 511 real scele ! frame elems per world elem 512 real sclin ! frame lines per world line 513 514 common /M0FRACOM/ ifrm, iwdth, mxlin, mnlin, mxele, mnele, 515 & sclin, scele, sclfg 516 517 518 C // Implementation is trivial; just capture the common 519 C // block values in the formal arguments. 520 521 w_ullin = mnlin 522 w_ulele = mnele 523 w_lrlin = mxlin 524 w_lrele = mxele 525 526 return 527 end
```

Contouring gridded fields

The mcgrdcon function computes and draws contours on a grid. Since contour drawing requires navigation, you must make a call to **mapdef** prior to calling **mcgrdcon.** The simple example below will draw contours using the value entered with the keyword CINT on the command line. If CINT=0, the **mcgrdcon** function will compute a contour interval and report it, drawing about 10 contour intervals.

```
integer nr !number of rows parameter (nr=22) integer nc !number of columns parameter
(nc=32) integer color !color of contours integer idash !line dashing control integer iwrap
!wrap contours around edges? integer lint !label interval l integer lsize !size (in
pixels) of labels integer smooth !smoothing factor for contouring real cint !contour
interval real grid (nr, nc) !grid to contour double precision dint !contour interval
character*80 lfmt !label format external ctcvcf !external function call for strings . . .
C---pick up cint from command line if (mccmddbl (' CIN.T' ,1,'Contour Interval'
,0,-1.D10,1.D10,dint) & .lt. 0) return !if mccmddbl returns less than !zero and error has
occurred cint = sngl (dint) . [get grid to contour] . [initialize navigation with mapdef
call] . lsize = 6 !labels are 6 pixels high color = 4 !use color 4 (green) idash = 0 !dash
only negative values lint = 4 !label very 4th contour lfmt = ' (''Height = ' ,F6.0,'
meters'')' !format statement !that controls !how the contour !labels look smooth = 15
!smoothing factor iwrap = 0 iret = mcgrdcon (grid,nr,nc,cint,lsize,color,idash,lint &
lfmt,smooth,iwrap,ctcvcf)
```

The **mcgrdcon** function also has the capability to draw at specific contour values if those values are entered in the string table, and the non-numeric string name is entered as the contour interval. For example, if the user enters the two commands below:

Type: **TE TEST1 "540 562 598**
GRDDISP ETA/00 PARAM=Z LEV=500 CINT=TEST1 UNIT=DM

An ETA model 500 mb Z field is displayed with contours drawn at 5400, 5620, and 5980 m.

The code that handles this case, taken from **grddisp.pgm**, is shown below and continued on the next page.

```
1 C --- a common block that passes the string name for the external 2 C contour intervals
to the external function is defined: 3 4 external ctcvcf !external function call for
```

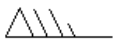
```
string 5 !table entry 6 . 7 . 8 . 9 common/extcomm/cint !cint is a character*12 and will
10 . !contain the string table entry 11 . !holding the contour values 12 . 13 14 C ---
when GRDDISP picks up the cint string, 15 16 if (mccmdstr ('CIN.T',1,' X',cint) .lt 0)
goto 2000 17 18 n = nchars (cint,begin,end) 19 20 rc = mcstrtodbl (cint (begin:end), dint)
21 22 C --- if cint is not a number, then rc will be <0 and dint will be 23 C zero. 24 C
This will be the case if a string table entry is being used to 25 C set the contour
values. 26 27 if (rc.lt.0 .and. dint.eq.0)then 28 intc = lit ('EXT') 29 xint = alit
('EXT') 30 else !cint is an integer 31 xint = sngl (dint) 32 intc = ifix (xint) 33 endif
34 . 35 . 36 . 37 C --- The call to mcgrdcon is done after the gridded data have been 38 C
read in from the server. Grid contains floating point data, not 39 C the scaled integers
as stored in the grid array. Because the 40 C data is floating point, an I-type format in
the lfmt string 41 C generally will not work. 42 lsize = 6 !label size is 6 pixels 43
icolor = 4 !draw in color 4 (green) 44 idash = 0 !dash negative contours 45 lint = 4
!label every 4th contour 46 lfmt = ' (''' Height = ',F6.0, ' meters''')' !format statement
that 47 !controls the contour 48 !labels 49 50 ismoth =15 51 iwrap = 0 52 maxlev = 0 53 54
iret=mcgrdcon (grid, nr, nc, xint, lsize, icolor, idash, lint, 55 & lfmt,ismoth, iwrap,
maxlev, ctcvcf)
```

Wind barbs and wind vectors

The McIDAS-X library provides seven functions for plotting wind direction and speed using wind barbs and also provides a function for plotting vectors. The table below describes these seven functions.

Fortran function	C function	Description
mcplotwind	**Mcplotwind**	plots wind information using barbs and flags at the designated line/element
mcplotwindnavll	**McPlotWindNavLL**	plots wind information using flags and barbs at a designated latitude/longitude; wind direction is corrected for projection
mcplotskywind	**McPlotSkyWind**	plots wind and sky cover information at a designated line/element
mcplotskywindnav	**McPlotSkyWindNav**	plots wind and sky cover information at a designated line/element; wind direction is corrected for projection
mcplotskywindnavll	**McPlotSkyWindNavLL**	plots wind and sky cover information at a designated latitude/longitude; wind direction is corrected for projection
mcplotwmocalm	**McPlotWMOCalm**	plots the WMO calm wind symbol at the designated line/element
pltwnv		plots wind information using wind vectors at the designated latitude/longitude

The **plotwind** functions plot wind speed without converting any units. A wind barb uses triangular flags and long and short lines to represent wind speed. For example, the wind flag here represents approximately 75 knots.

- 
- A flag represents 50.
- A long barb represents 10.
- A short barb represents 5.

The functions that plot sky cover (**mcplotskywind**, **mcplotskywindnav**, **mcplotskywindnavll**) use the WMO code value for sky cover, which indicates the sky cover in eighths.

The **pltwnv** function plots wind speed using a vector. The vector length can be proportional to the wind speed or a fixed length. To set a proportional vector length, specify a length less than zero in the call. To set a specify a fixed length, specify a length greater than zero. In addition, a previous call to **mapdef** is required so the navigation functions accurately compute the wind position and direction.

The example below demonstrates the use of the **mcplotwind**, **mcplotskywindnavll**, **and pltwnv** functions.

```
subroutine main0 double precision direction ! wind direction in degrees double precision
speed ! wind speed in knots double precision latitude ! latitude in degrees N double
precision longitude ! longitude in degrees W real latlon(4) ! arrays needed by mapdef
integer tvbounds(4) real navparms(4) integer length ! unit length of wind flag/ ! vector
```

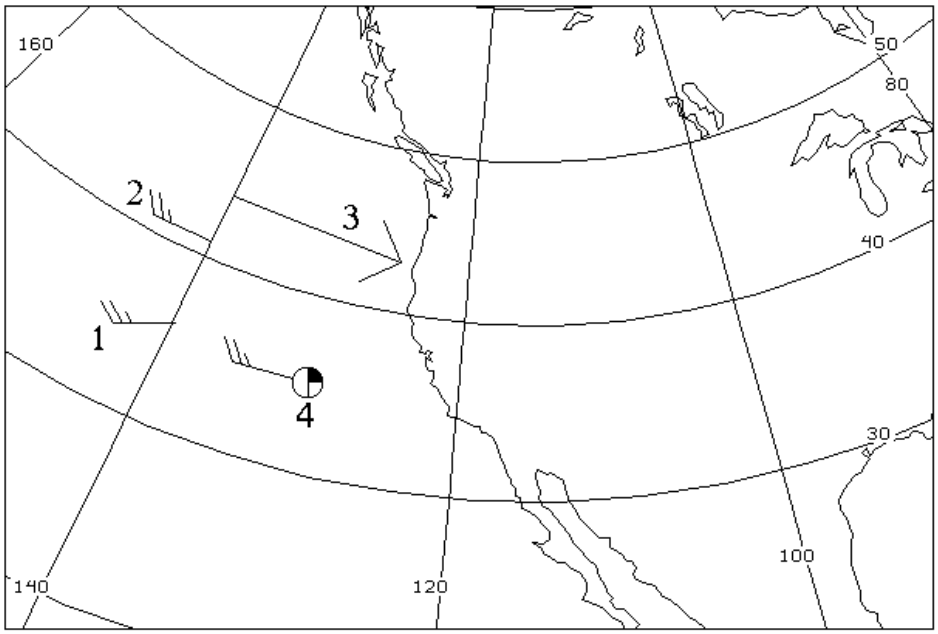
```
integer color_lev ! graphics color to use integer sky_cover ! sky cover WMO code value
integer frame ! current frame number integer hemisphere ! hemisphere flag integer ok data
length/40/ data color_lev/3/ data IWID/1/ c---Initialize the graphics routines IGDEV =
luc(-5) CALL INITPL(IGDEV,IWID) c---Define the wind direction and speed. direction = 270.0
speed = 27. frame = luc(-1) c---Now plot the non-corrected wind flag at line 225, element
130. c---Note in the display that the direction 270 degrees is plotted c---straight left-
right -- that is, it has not been corrected for c---the navigation/projection. hemisphere
= 1 ok = mcplotwind(225, 130, length, color_lev, hemisphere, & direction, speed) c---
Define a new latitude/longitude for plotting a wind flag c--- with a corrected angle. Note
that the plotted direction is c--- correctly shown as an earth-relative 270-degree angle.
latitude = 41. longitude = 140. ok = mcplotwindnavll (frame, latitude, longitude, length,
& color_lev, direction, speed) c---Now give an example of a wind vector. Note that this c-
--is always corrected for the navigation. c--- c---Note that a call to mapdef is required
before calling c---pltwnv; only one call per frame is needed. ok = mapdef(latlin,
tvbounds, 'SAT', navparms) call pltwnv(nint(speed), nint(direction), 440000, 1400000, &
color_lev, length) c---Plot the wind flag and sky cover symbol at 35N 130W. The angle c---
of the wind flag is corrected for the navigation. latitude = 35.0 longitude = 130.0
sky_cover = 3 ok = mcplotskywindnavll(frame,latitude,longitude,length, & color_lev,
sky_cover, direction, speed) call endplt return end
```

Figure 4-8 shows the results of the above example. For this example, the MAP command below was run to produce the background map in a conformal projection and navigation for the frame.

MAP DEF X LALO LAT=20 45 LON=95 165 SLAT=60 SLON=115 PRO=CONF

Note that wind flag1, generated by the **mcplotwind** call, makes no angle correction based on the navigation for the map projection. However, wind flag 2, generated by the **mcplotwindnavll** call, correctly rotates the wind direction relative to the earth to compensate for the map direction. Wind flag 3 was generated by the **pltwnv** call. Note the preceding **mapdef** call in the code. Windflag 4, generated by the **mcplotskywindnavll,** plotted both an angle-corrected wind flag and a sky cover symbol, as if constructing a station model plot.

Figure 4-8. *Three wind barbs and a wind vector are plotted on a conformal projection.*



Weather Symbols

The McIDAS-X library provides eight functions for plotting weather symbols. The table below describes these eight functions.

Fortran function	C function	Description
mcplotwmoweather	**McPlotWMOWeather**	plots present weather symbols from WMO code
mcplotwmopastwxmand **mcplotwmopastwxauto**	**McPlotWMOPastWxMand** **McPlotWMOPastWxAuto**	plots past weather symbols from WMO manned and automated code
mcplotwmosky	**McPlotWMOSky**	plots sky cover symbol from WMO code

mcplotwmolow **mcplotwmomid** **mcplotwmohigh**	**McPlotWMOLow** **McPlotWMOMid** **McPlotWMOHigh**	plots cloud symbols for low, middle, and high level clouds from WMO code
mcplotwmopressuretendency	**McPlotWMOPressureTendency**	plots pressure tendency symbols from WMO code
pltxtn **wxsplt**		old routines for plotting weather symbols

Display characteristics

The McIDAS-X display is the device used to output image and graphical data. Use the two functions below to inquire about the characteristics of the display.

Fortran function	C function	Description
itrmch	not available	obtains the value of a McIDAS-X display characteristic
mcfsize	**Mcfsize**	obtains the size of a frame, in lines and elements

As a developer, you should use this information for validating display-related command parameters and for making environment-dependent programming decisions. For example, you can find information about the display type, the minimum and maximum valid color levels, or the number of default display lines and elements.

Both the **itrmch** and **mcfsize** functions isolate programs from directly accessing User Common, so you don't need to know the User Common index to find the value of a display characteristic.

Obtaining the value of a display characteristic

Applications use the **itrmch** function to do the following:

- Inquire about the default value of a display characteristic
- Check if a feature is supported by the display

To obtain a value, use the appropriate character string from the table below as the first argument to the function.

Strings ending in a question mark return 1 for true and 0 for false.

Valid itrmch strings	Description
TERMINAL_TYP	terminal display type as an integer literal of four characters, for example: "XWIN"
OPER_SYSTEM	operating system of the current McIDAS-X session on this workstation as an integer literal of four characters, for example: "UNIX"
DSP_LINES	number of default display lines
DSP_ELEMENTS	number of default display elements
MAX_COLORS	maximum color level
MIN_COL_LEV	minimum valid color level
MAX_COL_LEV	maximum valid color level
MX_CUR_SIZ_H	maximum cursor height
MX_CUR_SIZ_W	maximum cursor width
MN_CUR_SIZ_H	minimum cursor height

MN_CUR_SIZ_W	minimum cursor width
GRAY_LEVELS	number of gray levels available
ANNOTAT_SIZE	size of the annotation
DWELL_RATE_K	dwell rate constant
BITS_PER_PIX	number of bits per pixel
IND_GRAPHIC?	Does the display have independent graphics?
VAR_FR_SIZE?	Does the display have variable frame sizes?
DISP_TOGGLE?	Is a display toggle required to show a frame?
ZOOM?	Does the display have zoom capability?
STRETCHING?	Is contrast stretching supported?
CUR_BOX?	Is the box cursor type supported?
CUR_XBOX?	Is the crossbox cursor type supported?
CUR_XHAIR?	Is the crosshair cursor type supported?
CUR_BULL?	Is the bullseye cursor type supported?
CUR_SOLID?	Is the solid box cursor type supported?

The **itrmch** function can be used to validate user-entered command parameters that relate to a display. The example below is from the ZA command. It exits if the color for the current terminal is out of range. The second argument is no longer used.

```
C--Get the number of color levels for the terminal C--and see if the user asked for a level
out of the C--accepted range. LEVEL=PP(1,1) MAXLVL= ITRMCH('MAX_COL_LEV',-1) IF (LEVEL.LT.0
.OR. LEVEL.GT.MAXLVL) THEN CALL EDEST('Graphics color invalid ',LEVEL) REYTURN endif
```

Obtaining the size of a frame

Frame sizes can vary within a McIDAS-X session. Your applications should never assume a size but instead call the **mcfsize** function to return the size of a given frame in lines and elements, as shown in the code fragment below.

```
integer frame integer lines integer elems C--- get size of the frame frame=4 call
mcfsize(frame, lines, elems) C--- upon successful completion, lines and elems will contain
C--- frame 4's screen dimensions
```

System utilities

The McIDAS-X library provides a group of functions for performing system-related tasks such as:

- Allowing applications to determine and alter the terminal state
- Permitting an application to sleep for a while
- Starting a McIDAS-X command from an application and informing the caller of its success or failure
- Allowing applications to use shared resources, such as files, without interfering with each other

This section provides descriptions and examples of the functions that you can use in your applications to perform these tasks.

For additional information about the functions described in this section, see the online man pages provided with the McIDAS-X software.

String tables

A McIDAS-X string table is a collection of names and associated strings. String tables serve two purposes:

- Users can use string tables to simplify command entry.
- Programmers can use string tables to pass information between commands run at different times.

Users can run the commands TU, TL, TE, TD and REPEAT to manage string tables and simplify command entry. For information about these commands, see the *McIDAS-X Learning Guide* and the *McIDAS User's Guide*.

As a programmer, you will find string tables useful because they can serve as a scratch pad for applications to communicate with each other. One application writes information into a character string and saves it in the table where it is available to all subsequent applications, until either the string is changed or the string table is replaced with another.

Use the three functions below to read from or write to a string table.

Fortran function	**Description**
lbget	gets a string from the current string table
lbput	writes a string to the current string table
lbputc	forms a character string from a sequence of character tokens and writes it to the string table

Although no separate operation exists to delete a string name from the string table, you can delete a string name using **lbput** with an argument string consisting of one or more blank characters.

Note that **lbputc** takes an input array of tokens and concatenates them. The resulting string is then associated with the string name only if it is less than 160 characters long. The **lbput** function will associate only a single string with the string name, subject to the same 160-character limit.

Limitations

You must observe the following limitations when developing applications that access string tables.

- A string name may not exceed 12 alphanumeric characters.
- You can't use Y or H as string names because they are reserved for the present date and time and can't be reassigned.
- A question mark as the first character of a string name indicates a *global string*, which does not change when the workstation's current string table is replaced with another.
- Strings may not exceed 160 characters.
- A single string table may contain a maximum of 256 entries.

Cautions

String tables provide convenient, global storage for passing information in and out of applications without using the command line, text display or file I/O. Although the technique for implementing a string table is easy, using string tables for passing information between commands is not always desirable. If the applications using string tables do not provide a facility for reinitializing the strings used, unpredictable results may occur during subsequent use of the applications.

Storing or sharing information using the string table functions can make applications more convenient for users if used sparingly and documented clearly. It can just as easily lead to confusion, since a string name can't be protected for an application's exclusive use. Other applications or the user (via the string table editor TE) can modify or delete the string, causing the application relying on that string to behave unpredictably. To minimize the risk of name collisions in the string table, choose an unlikely name for the stored string.

Use the McIDAS-X disk file system if your application needs to save more than a few pieces of information that will be used by other applications at a later time. Use a text file, whenever practical, as it simplifies the editing and viewing of the file and makes debugging your application easier.

Examples

The code segments below demonstrate the McIDAS-X string table API. Sample user-level input, via the TE and ECHO commands, is also provided to show how string table entries can be written to and read from both the command line and within an application.

Reading a string from a string table

The **lbget** function extracts the contents of a string from within an application and places it in a character variable. If the string table value is longer than the character variable receiving it, the value stored in the character variable is truncated. For example, a user enters the following command from the McIDAS-X Text and Command Window.

TE HITTER "MICKEY MANTLE

The code segment below reads the contents of the string HITTER.

```
character*12 string_name ! name of McIDAS string to ! extract value from character*24
out_string ! variable to store resulting ! value in string_name = 'HITTER' ok = lbget
(string_name, out_string) if (ok .lt. 0)then goto 999 endif c--- upon successful
completion the contents of the variable c--- out_string will be 'MICKEY MANTLE'
```

Writing a string to a string table

The **lbput** function stores the contents of a character string in the current McIDAS-X string table associated with a specified string name, as shown in the code fragment below.

```
character*12 string_name ! name of McIDAS string to ! place new value in character*24
in_string ! variable to store resulting ! value in string_name = 'HITTER' in_string =
'Willie Mays' ok = lbput (string_name, in_string) if (ok .lt. 0)then goto 999 endif c---
upon successful completion the contents of the McIDAS c--- string HITTER will be 'Willie
Mays'
```

If a user subsequently runs the following command from the McIDAS-X Text and Command Window:

ECHO "The Greatest Centerfielder of all time was #HITTER

The output displayed on the text screen will appear as shown below.

```
The Greatest Centerfielder of all time was Willie Mays
```

Writing character tokens to a string table

The **lbputc** function allows a user to enter a group of character tokens into one string. The code fragment below demonstrates this function.

```
parameter (MAXTOK = 3) character*12 string_name ! name of McIDAS string to ! place new
value in character*12 tokens(MAXTOK) ! string tokens to write into ! the McIDAS string
700CLUB string_name = '700CLUB' tokens(1) = 'Hank Aaron' tokens(2) = 'and' tokens(3) =
'Babe Ruth' ok = lbputc (string_name, MAXTOK, tokens) if (ok .lt. 0)then goto 999 endif c-
-- upon successful completion the contents of the McIDAS c--- string 700CLUB will be 'Hank
Aaron and Babe Ruth'
```

If a user subsequently runs the following command from the McIDAS-X Text and Command Window:

ECHO "#700CLUB are the two greatest sluggers of all time

The output displayed on the text screen will appear as shown below.

```
Hank Aaron and Babe Ruth are the two greatest sluggers of all time
```

User Common

McIDAS-X applications run in an environment consisting of both static and dynamic parts.

- The static part of the environment defines the constraints imposed by the hardware or specified when the McIDAS-X session was started. These constraints remain constant throughout the session; for example, the number of graphics color levels.
- The dynamic part of the environment is controlled by both the user, via the keyboard and mouse, and other McIDAS-X processes including applications; for example, the image frame displayed.

User Common contains the dynamic state of the session. All processes, including applications, can read and modify User Common. You will use it in applications to alter the display and make the applications interact with each other in predictable ways.

For information about the **itrmch** function, which provides details about the static part of the display, see the section in this chapter titled *Display characteristics*. For a detailed listing of the contents of User Common, see the file **uc.doc** in the McIDAS-X source directory.

Positive and negative User Common

User Common is divided into positive and negative regions. You must understand the differences between the two and use the appropriate region so that your applications will behave predictably. Note that each session running on a workstation has its own private copy of User Common.

- Use positive User Common when writing interactive applications. Positive User Common contains the instantaneous state of the session. At any instant, it is the same for all processes. All processes, including applications, may change it at any time, and these changes are immediately visible to all other McIDAS-X processes and to the user via the display.
- Use negative User Common when writing applications that need to know the state of the system when the user starts a command by pressing the Enter key. Negative User Common contains a copy of the session state taken at the time a process or *process chain* starts, plus any modifications made to it by the process or chain. A process chain is a series of McIDAS-X processes run synchronously.

Negative User Common is initialized when the process chain starts. All subsequent processes inherit negative User Common and any changes made to it by processes in the chain.

Batch files, McBASI scripts, and commands separated from each other by a semicolon on a single command line all initiate a chain. Because each process chain has its own copy, Negative User Common cannot be changed from outside the process chain and changes to it are not visible except to the owning process chain, including the display.

The following sequence of commands illustrates why both positive (instantaneous) and negative (process-dependent) User Common are necessary and how applications use them to achieve predictable interactive behavior. If the user enters the three commands below:

```
SF 2
IMGDISP EASTS/CONUS STA=MSN BAND=4
SF 4
```

The first command sets the current frame to 2; the second command requests that the latest GOES-EAST 4 km IR image, centered on Madison, Wisconsin, be displayed on the current frame. Now suppose the user wants to view a graphic already displayed on frame 4, and enters the third command before IMGDISP begins displaying the image. The display immediately switches to frame 4.

What happens to the IMGDISP command trying to display to the current frame? Is the current frame the frame the user intended (frame 2) or the frame presently displayed (frame 4)? The ambiguity is resolved with positive and negative User Common.

When SF 2 changes the appropriate word in User Common, in this case word 51, to 2, the display immediately reflects it. When IMGDISP starts, the display state, including the current frame number 2, is copied into negative User Common. IMGDISP can then examine the appropriate User Common word, in this case -1, to get the frame number that was current when it started. This value is always 2; whereas the true current frame (word 51) changes from 2 to 4 when SF 4 runs.

User Common API functions

The User Common functions are described in the table below.

C function	Fortran function	Description
Mcluc	**luc**	returns a value from User Common
Mcpuc	**puc**	changes a value in User Common

The most common error in using these functions is specifying the wrong User Common index value; the second most common error is transposing a new value with the User Common index value in **puc** and **Mcpuc**.

The **`m0glue.h`** include file has many of the User Common indexes enumerated using #define statements.

Several APIs are provided in addition to the basic **Mcluc** and **Mcpuc** functions. They are described in the table below.

C function	Fortran function	Description
McGetGraphicsFrameNumberI	**mcgetgraphicsframenumberi**	returns the current graphics frame number for an interactive application
McGetImageFrameNumberI	**mcgetimageframenumberi**	returns the current image frame number for an interactive application
McGetGraphicsFrameNumber	**mcgetgraphicsframenumber**	returns the current graphics frame number
McSetGraphicsFrameNumber	**mcsetgraphicsframenumber**	sets the current graphics frame number
McGetImageFrameNumber	**mcgetimageframenumber**	returns the current image frame
McSetImageFrameNumber	**mcsetimageframenumber**	sets the current image frame number
McGetMaxImageFrameNumber	**mcgetmaximageframenumber**	returns the maximum image frame number for the current session
McGetMaxGraphicsFrameNumber	**mcgetmaxgraphicsframenumber**	returns the maximum graphics frame number for the current session
McIsImageLooping	**mcisimagelooping**	returns current image looping state
McIsImageFrameOn	**mcisimageframeon**	returns whether or not the current image frame is visible
McSetImageFrameOn	**mcsetimageframeon**	sets current image frame to visible
McIsImageConnectedToLoop	**mcisimageconnectedtoloop**	indicates if the image frames are connected to the looping system
McIsGraphicsLooping	**mcisgraphicslooping**	returns current graphics looping state
McIsGraphicsFrameOn	**mcisgraphicsframeon**	returns whether the current graphics frame is visible
McSetGraphicsFrameOn	**mcsetgraphicsframeon**	sets current graphics frame to visible
McIsGraphicsConnectedToLoop	**mcisgraphicsconnectedtoloop**	indicates whether the graphics frames are connected to the looping system
McGetStdOutputDevice	**mcgetstdoutputdevice**	returns the destination device for standard output messages
McGetStdErrorDevice	**mcgetstderrordevice**	returns the destination device for standard error messages
McGetStdDebugDevice	**mcgetstddebugdevice**	returns the destination device for standard debug messages
McSetStdOutputDevice	**mcsetstdoutputdevice**	sets the destination device for standard output messages
McSetStdDebugDevice	**mcsetstddebugdevice**	sets the destination device for standard debug messages
McSetStdErrorDevice	**mcsetstderrordevice**	sets the destination device for standard error messages
McIsMcIDASRunning	**mcismcidasrunning**	indicates whether a

		McIDAS-X session is active

User Common functions are often used to verify that an item, such as a frame, etc., is within a valid range. The example below, from the MAP command, illustrates the use of one of these functions.

Determining if the current frame is within a valid range

The MAP command fragments below define what the highest numbered image frame is and compares that number to the desired frame to be used in the command.

```
c--- maximum image frame MAX_IMAGE = mcgetmaximageframenumber() . . . c--- get the current
image frame def_image_frame = mcgetimageframenumber() c--- get the user desired frame and
check to see if it is c--- out of range. Note that the user may not have input c--- a
frame number, in which case the default frame is used. status = mccmdint('IMA.GE',1,'Image
Frame',def_image_frame, & 1,MAX_IMAGE,nfri) if( status.lt.0 ) return
```

Note that **mcgetimageframenumber,** not **mcgetimageframenumberi**, determines the current frame. This ensures that the current frame is the one displayed at the time MAP is started, not the frame displayed at the time the above code fragment actually runs.

Restoring the display to its original state

The sample code below performs these three tasks:

- Reads the roam and cursor state from User Common into variables
- Performs interactions involving the mouse
- Restores the original roam and cursor states when done

Note the use of **luc** and **puc** since no other APIs are currently available.

```
c --- freeze the roam roam = luc( 178 ) call puc( 1, 178 ) c --- connect cursor to mouse
mouse = luc( 67 ) call puc( 1, 67 ) <deleted code> 100 continue status = mcmoubtn(3,
button(1), button(2), line, elem) altG = 0 if(status.eq.2) altG=1 altQ = 0 if(status.eq.1)
altQ=1 c --- check for program termination if( altQ.ne.0 ) then call beep(200,100)
interact = -3 return endif <deleted code> goto 100 <deleted code> c --- reset the mouse
call puc( mouse, 67 ) c --- reset the roam call puc( roam, 178 )
```

Starting McIDAS-X commands from applications

When developing an applications program, check to see if a McIDAS-X command performs a needed task. If so, you should start that command from your application rather than duplicating the logic inside your program. If fixes are made to the command, your program will automatically contain the updated information. If keywords are removed from a command, it's your responsibility to make the necessary changes in your calling application.

McIDAS-X commands can run synchronously or asynchronously, with or without extended format.

- Commands that run *synchronously* will run to completion before control is returned to the original calling program.
- Commands that run *asynchronously* return control to the original calling program before they have run to completion.
- Commands that run with an *extended format* may contain a semicolon (;) indicating the start of a sequence of commands, or one or more pound signs (#) indicating a required string substitution.

You can start any McIDAS-X command from an application using the functions defined in the table below.

C function	Fortran function	Description
not available	**keyin**	starts a command asynchronously with extended format
not available	**skeyin**	starts a command synchronously with extended format
Mckeyin	**mckeyin**	starts a command asynchronously
Mcskeyin	**mcskeyin**	starts a command synchronously

Use the **keyin**, **mckeyin** and **Mckeyin** functions to start commands asynchronously. For example, the McIDAS-X time scheduler, **sked**, starts user commands asynchronously. If the scheduler started a long-running command synchronously, other scheduled commands couldn't start until that command finished.

Use the **skeyin**, **mcskeyin** and **Mcskeyin** functions to start commands synchronously. Using these functions, the application stops and will not continue until the specified command is done. For example, a user can run the ERASE command to erase a frame and then run the IMGDISP command to display an image on that frame. If these applications aren't run synchronously, some of the image load could occur before the ERASE command finishes, erasing part of the image.

Starting commands synchronously

The **mckeyin** and **mcskeyin** functions expect a single command in McIDAS-X format. The asynchronous version, **mckeyin**, returns the status of the command if one was started, or an error code if it wasn't started. The synchronous version, **mcskeyin**, returns the exit status of the started command. The status code returned from **mcskeyin** is set by calling the function **Mccodeset**, which is described in the next section titled *Error handling*.

Below is a code fragment from an application that uses **Mcskeyin**. The first call attempts to position the cursor at a certain latitude and longitude. If this call returns a successful status, a second call is made to report the position in all the relevant coordinate systems.

```
/* * if there is exactly one match * try to put the cursor there */ if(found==1) { char
command[100]; sprintf( command, "PC E %f %f DEV=NNN", slat, slon); /* * if the cursor
positioning was successful * output the position */ if(Mcskeyin( command )==0)
(void)Mcskeyin("E"); }
```

Missing Value codes

It is not uncommon to have data reports with missing values. The McIDAS-X system uses specific values that flag parameters as missing. Integer values for code written in C use the macro MCMISSING4 defined in mcidas.h. Integer values for code written in FORTRAN use the value HEX80 stored in hex80.inc.

The McIDAS-X library provides a group of functions for assigning and verifying floating-point missing value codes. The functions are described in the table below.

C function	Fortran function	Description
McGetMissingDbl	**mcgetmissingdbl**	returns the missing value code for a 64-bit floating point number
McGetMissingFlt	**mcgetmissingreal**	returns the missing value code for a 32-bit floating point number
McIsMissingDbl	**mcismissingdbl**	tests whether a 64-bit floating point number contains the missing value code
McIsMissingFlt	**mcismissingreal**	tests whether a 32-bit floating point number contains the missing value code

Error handling

Most functions use their return value to inform the calling program of their success or the nature of their failure. So it is with McIDAS-X commands, though in a more general way. When run from the command line, error messages generated by the **edest** or **Mcprintf** function are sufficient. These functions are described in the *Text messages* section earlier in this chapter.

McIDAS-X also allows scripts or McIDAS-X applications to run other applications, and depending upon the results, modify subsequent actions. For example, if you run a command to get data and then want to run a transformation on it, you can abandon the second command if the first one fails to acquire data.

This table describes the functions that an application should use to set an error code.

C function	Fortran function	Description
Mccodeset	**mccodeset**	sets the global status to return upon exiting
Mccodeget	**mccodeget**	returns the current value of the global status
not available	**mcabort**	sends an error message and exits

Mciniterr	not available	returns a string explaining why Mcinit failed

Some older McIDAS-X applications are not yet modified to use this status facility, so a zero, or successful status may sometimes be returned erroneously. Although Fortran programs that call **mcabort** return a status of 1, the process that started the command won't know if **mcabort** was called, or just **return** or **exit**. All new McIDAS-X applications are coded to call **Mccodeset** if they encounter a problem when running. Thus, when writing an application:

Do:	Don't:
call **`edest`**, call **`mccodeset`**, and **`return`** a status in a function return code	call **`mcabort`** or **`exit`** since they are not very informative

Currently, SSEC uses the return codes below for McIDAS-X applications.

Value	Definition
0	command ran successfully
1	command failed with an unrecoverable error
2 to 99	command failed with a potentially recoverable error
100 to 126	reserved for locally developed applications
127	error of unknown origin

For example, use a return code of 1 to tell users they entered a command with a syntax error. Use a return code of 2 to indicate the user's request is valid, but the data requested is not available yet.

The code fragment below is from a command that uses the **mccodeset** function.

```
integer ret_val character*12 option c--- get command option ok = mccmdstr(' ', 1, ' ',
option) if (option .ne. 'LIST' .and. option .ne. 'COPY')then ret_val = 1 goto 999 endif :
: c--- get data ok = m0ptget (dataset, nsort, sort, nparms, parms, units, & form, scales,
maxbyte, 1) if (ok .lt. 0)then ret_val = 2 goto 999 endif c--- process data : : ret_val =
0 999 continue call mccodeset (ret_val) call edest('done',0) return end
```

Suspending applications

The **mcsleep** function suspends an action or application for a specified number of milliseconds. It is most often used for polling, or repetitively checking to see if something has changed.

The term *sleep* means that an application tells the operating system that it doesn't want to be considered ready to be dispatched for a period of time. The system does not guarantee that the application will be dispatched again exactly when its sleep interval is over because that depends on system load. However, it does guarantee that the application will not wake up again until at least that period of time expires.

For example, the **sked** application checks the scheduler file every thirty seconds to see if any of its entries should be run. The **Mcmoubtn** function sleeps for 10 milliseconds if it is trying to detect a change in the cursor position. Even though these intervals are very different, their purpose is the same: to allow other processing to continue without slowing down the system by checking something more often than necessary.

The choice of a time interval is a matter of experimentation since you can't know how fast or busy the system will be. Don't test an application only on very fast machines if it will also be run on slow ones. Factors, such as whether file systems or servers are local, can also affect how long a unit of work takes, which in turn affects your choice of a time interval.

Below is a code fragment showing how long-running processes use the **mcsleep** function to perform polling. Notice that they always check the system shutdown User Common word (word 194) using the **McIsMcIDASRunning** function after waking up so they won't continue running when they're no longer needed.

```
c-- go to sleep for 10 seconds call mcsleep(10000) c-- verify that mcidas is still running
if mcismcidasrunning ( ).eq.0) then return endif . . .
```

File locks

An application sometimes needs exclusive use of a resource, such as one or more words in shared memory or one or more bytes in a file. The lock and unlock functions, shown in the table below, control the exclusive use of a resource, guaranteeing that once the program starts to run, nothing will interfere until it's finished.

C function	Fortran function	Description
M0lock	**lock**	blocks another application from using a resource
M0unlock	**unlock**	releases a resource so another application can use it

For example, assume two programs want to update a record in a file of 100-byte records. The first program wants to change the third byte of the first record; the second program wants to change the fourth byte of that record.

If both programs read the record, change their respective byte, and write the record back out, one of the changes will be lost because both programs read the unmodified record before making their changes.

Using the **lock** function will prevent this from happening. If each program locks before it reads the record and unlocks after it writes the record, one process will wait for the other to complete before it begins.

Consider the following recommendations when locking and unlocking a resource.

- When defining a lock, use a text string that is also a legal file name, since the locking mechanism may be implemented via the file system.
- If the lock refers to an actual file, use the name of that file. The file name must be consistent among all the programs that use and modify the resource.
- Only own one lock at a time and keep it for the shortest period of time needed to protect your activity.
- If you must own two or more locks at a time, make sure that all programs which do so request them in the same order.

The sample code below is from the file DDESERVF. The **lock** function is called by the function that updates the file; the **unlock** function is called after the file is closed so another process can access the lock.

```
c--- Open the resolver file, and read contents istat=volnam('RESOLV.SRV', pathname) call
lock('RESOLV.SRV') open(21, file=pathname, status='old', err=100) 1 read(21, 2, err=99,
end=99) work2 2 format(A) c ... (real processing goes on here) 99 close(21) c c--- set
return code based on whether or not the c--- name was resolved c if (valid.eq.0) then
m0sxresolv=-1 else m0scresolv=0 endif call unlock('RESOLV.SRV') 100 return end
```

Conversion utilities

The McIDAS-X library provides many practical conversion utilities for your applications. These functions were developed because the standard Fortran and C libraries didn't provide sufficient routines for performing the conversions needed in McIDAS-X.

This section describes the functions available in for performing these tasks:

- Manipulating information at the byte level
- Handling character strings
- Converting day and time formats
- Converting latitude and longitude formats
- Converting physical units such as speed and temperature

For additional information about the conversion functions described in this section, see the online man pages provided with the McIDAS-X software.

Byte- and word-level manipulation

When writing applications for McIDAS-X, you may find it necessary to manipulate information at the byte level. This section describes the McIDAS-X library functions that provide byte-level manipulation for Fortran routines and gives examples of specific byte-level operations.

When writing C functions, you can usually use the byte manipulation routines provided in the standard C library. However, you may also want to use some of the functions presented here since they have special features. Most of the byte moving routines in this section are Fortran-callable functions written in C that use the byte manipulation routines provided with the standard C library.

This section contains some terms that may be unfamiliar to you. They are defined below.

- A *buffer* is any memory storage.
- A *byte* is an 8-bit memory segment; a *half-word* is a 16-bit memory segment; a *word* is a 32-bit memory segment.
- An *element* refers to a collection of one or more bytes.
- *0-based* is a counting sequence that begins with zero; *1-based* is a counting sequence beginning with the number one.
- *Big-endian* and *network-byte-order* are used interchangeably to mean the most significant byte in a word comes first; *little-endian* means the least significant byte is first. For example, the byte ordering for the integer value 12 appears as 0000000c on big-endian machines and 0c000000 for little-endian machines.
- *Memory overflow* refers to writing beyond the memory allocated for a variable.

Be careful when using the byte manipulation functions below. Most do not protect against memory overflow.

Fortran function	Description
alit	copies four bytes of a character string to a single precision floating-point variable
clit	copies four bytes of an integer value to a 4-character string variable
crack	copies a specified number of consecutive bytes from a source buffer into an array of words; each word is a 32-bit- long memory segment; the resulting byte values are stored in the least significant byte for each element of the word array
dlit	copies eight bytes of a character string to a double precision floating-point variable
ic	returns the value at a specific byte location (0-based) in a buffer
lit	copies four bytes of a character string to a 1-word integer variable
movb	copies a number of bytes from a source buffer to a specified, 0-based offset in a destination buffer
movblk	copies a number of elements of a specified size from a source buffer to a destination buffer with incremental offsets for both the source and destination buffers (0-based)
movc	copies a number of bytes beginning at a specified byte location in a source buffer (0-based) to a specified offset in a destination buffer (0-based)
movcw	copies the entire contents of a character string buffer to an integer buffer

movh	copies a number of half-words from a source buffer to a destination buffer with half-word increments
movpix	copies a number of elements from a source buffer to a destination buffer with incremental offsets for both the source and destination buffers
movw	copies a number of words from a source buffer to a destination buffer
movwc	copies the contents of an integer buffer to a character string buffer, copying as many bytes as can be stored in the destination string
mpixel	copies elements of a specified size in a source buffer to a destination buffer with elements of a specified size
mpixtb	like mpixel, but additionally converts variable-sized elements in a buffer based on a lookup table
pack	moves the least significant byte from each element of a word array and compresses it into consecutive bytes in a buffer
pack2	moves the least significant byte from each element of a half-word array and compresses it into a consecutive byte string
stc	places a byte value into a specified, 0-based byte location in a buffer
swbyt2	switches the byte ordering of a half-word to big-endian; this function has no effect on a machine that uses big-endian as its native storage format
swbyt4	switches the byte ordering of a word to big-endian; this function has no effect on a machine that uses big-endian as its native storage format
zeros	sets a number of bytes in an array to zero

Below are some code segments demonstrating the byte manipulation utilities. Similar functions are grouped together. The more commonly used functions are discussed first. Note that some of the byte manipulation functions will behave differently on big- and little-endian machines. All the examples in this section assume big-endian.

For more information about these byte manipulation functions, see the online man pages provided with the McIDAS-X software.

Extracting and inserting a byte value

The **ic** function lets you extract the value of individual bytes within a memory segment, as shown in the sample code below. It is useful when checking for ASCII control characters in a byte stream, for example.

```
c--- function ic example integer oldval integer val data oldval/z'abcdef01'/ val = 0 c---
extract the second byte, 0-based, from 'oldval' and c--- place the result in 'val' val =
ic (oldval, 2) c--- the resulting value in 'val' will be: c--- hex c--- 000000ef
```

The **stc** function lets you insert a byte value in a particular location in a memory segment. It is useful for substituting control characters with a blank before sending the output to a printer, for example.

```
c--- function stc example integer src data src/z'000000ba'/ val = 0 c--- place the byte
value from 'src' into the second byte, c--- 0-based, in 'val call stc (src, val, 2) c---
the resulting value in 'val' will be: c--- hex c--- 0000ba00
```

Switching the byte ordering

When you develop applications that read data from a remote data file or data stream, you usually need to know the byte ordering of the source data. In general, IBM-RISC, SUN, SGI and HP are big-endian machines, and DEC and Intel are little-endian machines.

The **swbyt2** and **swbyt4** functions have no effect on big-endian machines; however, they flip the bytes on little-endian machines, making the least significant byte the most significant byte. Because this byte ordering is transparent, you only have to maintain one set of source code for both types of architecture. When transferring data between systems, call **swbyt#** before writing and after reading.

Use the **swbyt#** calls on known integer values only; don't use them to process character strings.

```
c--- function swbyt4 example integer src(2) data src/Z'abcdef01',Z'23456789'/ c--- if
applicable, flip the 2 elements of the array 'src' c--- into network-byte ordering. call
swbyt4 (src, 2) c--- Because swbyt4 is a machine-dependent operation the c--- outcome of
this call will vary. On big-endian machines, c--- swbyt4 will not modify the values in the
buffer 'src'. c--- On little-endian machines it will. c--- big-endian little-endian c---
src() input value input value output value c--- 1 abcdef01 01efcdab abcdef01 c--- 2
23456789 89674523 23456789 c--- function swbyt2 example integer*2 src(4) data
src/Z'abcd',Z'ef01',Z'2345',Z'6789'/ c--- if applicable, flip the 4 elements of the array
'src' c--- into network-byte ordering. call swbyt2 (src, 4) c--- Because swbyt2 is a
machine-dependent operation the c--- outcome of this call will vary. On big-endian
machines, c--- swbyt2 will not modify the values in the buffer 'src'. c--- On little-
endian machines it will. c--- big-endian little-endian c--- src() input value input value
output value c--- 1 abcd cdab abcd c--- 2 ef01 01ef ef01 c--- 3 2345 4523 2345 c--- 4 6789
8967 6789
```

Copying bytes

The McIDAS-X library contains several functions for copying bytes. Some are simple copy routines, while others provide for more complicated byte moving operations.

movb and movc

The **movb** and **movc** functions copy a specified number of bytes from one buffer to another. The only difference between the two is that **movb** only lets you specify a destination buffer offset, while **movc** also lets you specify an offset from the source buffer.

```
c--- function movb example integer src(4) integer dest(4) data
src/Z'abcdef01',Z'23456789',Z'fedcba98',Z'09080706'/ call zeros (dest, (4 * 4)) c--- copy
15 bytes from the array 'src' into the array 'dest' c--- beginning at byte 1. call movb
(15, src, dest, 1) c--- The resulting value in 'dest' will be: c--- dest() hex c--- 1
00abcdef c--- 2 01234567 c--- 3 89fedcba c--- 4 98090807 c--- function movc example
integer src(4) integer dest(4) data src/Z'abcdef01',Z'23456789',Z'fedcba98',Z'09080706'/
call zeros (dest, (4 * 4)) c--- copy 14 bytes from the array 'src' beginning at byte 5 c--
- into the array 'dest' beginning at byte 2. call movc (14, src, 5, dest, 2) c--- The
resulting value in 'dest' will be: c--- dest() hex c--- 1 00004567 c--- 2 89fedcba c--- 3
98090807 c--- 4 06050403
```

movcw

The byte copying function **movcw** lets you move a character string into an integer buffer. This function doesn't convert the value stored in a string to its integer representation. It merely moves the bytes into an integer buffer.

Before using this function, verify that the destination buffer is as least as large as the source string, as **movcw** will move the entire source string to the destination address without protecting against buffer overflow.

```
c-- function movcw example integer dest(3) character*12 cvar12 call zeros (dest, (4 * 3))
cvar12 = 'June Cleaver' call movcw (cvar12, dest) c--- The resulting elements of the array
'dest' will be: c--- dest() hex string c--- 1 4a756e65 June c--- 2 20436c65 Cle c--- 3
61766572 aver
```

movblk and movpix

Use the byte-copying functions **movblk** and **movpix** for complicated byte copying operations. These functions let you specify the following:

- Segment sizes to copy
- Number of bytes to skip between source bytes
- Number of bytes to skip between destination bytes

Both the **movblk** and **movpix** functions let you copy bytes from one buffer to another with incremental offsets for both the source and destination buffers. However, the **movblk** function has an additional feature that lets you specify the length of each source element to move. The **movpix** function has a fixed, source-byte length of one and includes parameters for sampling bytes and moving a buffer in reverse order.

The **movblk** function is shown in the sample code below.

```
c--- function movblk example integer src(16) integer dest(32) data
src/Z'abcdef01',Z'23456789',Z'fedcba98',Z'09080706',
Z'05040302',Z'010a0b0c',Z'0d0e0f11',Z'22334455',
Z'66778899',Z'aabbccdd',Z'eeff1213',Z'14151617',
Z'18191a1b',Z'1c1d1e1f',Z'21232425',Z'26272829'/ call zeros (dest, (4 * 32)) c--- copy 4
elements that are 3 bytes long beginning at byte 1 c--- in 'src' into 'dest' beginning at
byte 6. The increment c--- between elements in the source buffer is 2 bytes, the c---
increment between bytes in the destination buffer is c--- 7 bytes. call movblk (4, 3, src,
1, 2, dest, 6, 7) c--- The resulting elements of the array 'dest' will be: c--- dest() hex
```

```
c--- 1 00000000 c--- 2 0000cdef c--- 3 01000000 c--- 4 00012345 c--- 5 00000000 c--- 6
45678900 c--- 7 00000089 c--- 8 fedc0000
```

The **movpix** function is shown in the sample code below.

```
c--- function movpix example integer src(16) integer dest(32) data
src/Z'abcdef01',Z'23456789',Z'fedcba98',Z'09080706',
Z'05040302',Z'010a0b0c',Z'0d0e0f11',Z'22334455',
Z'66778899',Z'aabbccdd',Z'eeff1213',Z'14151617',
Z'18191a1b',Z'1c1d1e1f',Z'21232425',Z'26272829'/ call zeros (dest, (4 * 32)) c--- copy 5
elements beginning at byte 1 in 'src' into 'dest' c--- beginning at byte 6. the increment
between elements in c--- the source buffer is 2 bytes, the increment between bytes c--- in
the destination buffer is 7 bytes. call movpix (5, src, 1, 2, dest, 6, 7) c--- The
resulting elements of the array 'dest' will be: c--- dest() hex c--- 1 00000000 c--- 2
0000cd00 c--- 3 00000000 c--- 4 00010000 c--- 5 00000000 c--- 6 45000000 c--- 7 00000089
c--- 8 00000000 c--- 9 0000dc00
```

Converting byte values

The **mpixtb** function converts the contents of a buffer to new values based on a lookup table included in the calling sequence. Simultaneously, it optionally converts the 1-, 2- or 4-byte data to a different length. This function is used in satellite calibration routines when a lookup table is generated to convert data from one physical quantity to another, such as raw values to temperature. The valid length conversions are 1, 2 or 4 bytes.

```
c--- function mpixtb example integer src(16) integer lookup(16) data src
Z'abcdef01',Z'23456789',Z'fedcba98',Z'09080706',
Z'05040302',Z'010a0b0c',Z'0d0e0f11',Z'22334455',
Z'66778899',Z'aabbccdd',Z'eeff1213',Z'14151617',
Z'18191a1b',Z'1c1d1e1f',Z'21232425',Z'26272829'/ data lookup/15, 14, 13, 12, 11, 10, 9, 8,
0, 1, 2, 3, 4, 5, 6, 7/ c--- convert 8 1-byte elements beginning at src array element 4 c-
-- into 2-byte elements that have been modified by the lookup c--- table. call mpixtb (8,
1, 2, src(4), lookup) c--- The elements of the array 'src' will be: c--- original new c---
value hex value hex c--- 09080706 00010000 c--- 05040302 00080009 c--- 010a0b0c 000a000b
c--- 0d0e0f11 000c000d
```

The **mpixel** function expands and packs the data values in place. If the source of the data is one byte, but the application expects it as four bytes, **mpixel** converts the data without needing an additional buffer.

Moving byte streams

The literal conversion functions **alit**, **dlit** and **lit** move the byte contents of character strings to the basic Fortran variable types.

```
c--- functions alit, clit, dlit and lit examples character*4 cvar4 character*8 cvar8
character*4 cval integer int_dest double precision dbl_dest real rel_dest cvar4 = 'FRED'
cvar8 = '& GINGER' rel_dest = alit (cvar4) int_dest = lit (cvar4) dbl_dest = dlit (cvar8)
cval = clit (int_dest)
```

The **clit** function moves the contents of an integer variable to a 4-character variable. These routines do not convert the string to its integer representation. They merely move the byte stream into the new variable. The resulting value may not even be a valid representation on the machine.

Occasionally, you may need to move a byte stream into whole or half-word integer variables and back into a byte stream. You can use the **crack**, **pack** and **pack2** functions for this purpose, as shown in the sample code below.

```
integer src(2) integer dest(8) data src /Z''abcdef01',Z'23456789'/ c--- function crack
example call zeros (dest, (4 * 8)) call crack (7, src, dest) c--- The resulting elements
of the array ''dest' will be: c--- dest() hex c--- 1 000000ab c--- 2 000000cd c--- 3
000000ef c--- 4 00000001 c--- 5 00000023 c--- 6 00000045 c--- 7 00000067 c--- function
pack example integer src(4) integer val data
src/Z'010000cd',Z'020000ef',Z'03000001',Z'04000023'/ val = 0 call pack(4, src(2), val) c--
- The resulting value in ''val' will be: c--- hex c--- cdef0123 c--- function pack2
example integer*2 src(4) integer val data src /Z'0f0e',Z'0d0c',Z'0b0a',Z'0908'/ val = 0
call pack2 (4, src, val) c--- The resulting value in 'val' will be: c--- hex c--- 0e0c0a08
```

Character string manipulation

This section describes the primary interfaces that you will use with character strings in McIDAS-X. These character-string functions were developed to do the following:

- Provide better error codes than those available in the standard language libraries
- Accommodate a McIDAS-X-specific syntax for some commonly used parameters unknown to the standard language libraries
- Control differences in the way that the C and Fortran languages treat character strings

This section contains some terms that may be unfamiliar to you. They are defined below.

- *Blank-padded* describes the practice of replacing unused characters at the end of a string with space characters.
- *Double-precision* usually describes a two-word storage representation for floating-point numbers.
- *Whitespace* is a subset of the ASCII (American Standard Code for Information Interchange) character set that includes space, end-of-line, vertical tab, horizontal tab and form-feed characters.
- *Null-terminated* describes the practice of placing a zero (ASCII NULL character) at the end of a character string; this is the standard representation in the C language.

The string manipulation functions are listed alphabetically in the table below with a brief description.

C function	Fortran function	Description
fsalloc	not available	copies the contents of a Fortran string to a C string, dynamically allocating the memory necessary to store the resulting string
fslen	not available	returns the logical length of a C string needed to contain a blank-padded Fortran string
not available	**ischar**	indicates whether a string of four characters contains all printable characters
not available	**isdgch**	determines whether a Fortran string contains all digits, all letters, a mixture of letters and digits, or other characters
Mclocase	**mclocase**	converts uppercase characters in a string to lowercase
Mcstricmp	not available	performs a case-independent comparison of two C character strings
Mcstrnicmp	not available	performs a case-independent comparison of a specified number of characters in two C strings
Mcstrtodbl	**mcstrtodbl**	converts a character string in decimal point format to a double-precision value
Mcstrtodhr	**mcstrtodhr**	converts a character string in time format to a double- precision value
Mcstrtodll	**mcstrtodll**	converts a character string in lat/lon format to a double-precision value
Mcstrtohex	**mcstrtohex**	converts a character string in hexadecimal to an integer value
Mcstrtohms	**mcstrtohms**	converts a character string in time format to its components: hours, minutes, seconds
Mcstrtoihr	**mcstrtoihr**	converts a character string in time format to an integer value in units of hours/minutes/seconds
Mcstrtoill	**mcstrtoill**	converts a character string in lat/lon format to an integer value in units of degrees/minutes/seconds
Mcstrtoint	**mcstrtoint**	converts a character string to an integer value
Mcstrtoiyd	**mcstrtoiyd**	converts a character string representing a day to an integer representation of Julian day
Mcstrtofs	not available	copies a C character string to a Fortran character string
Mcupcase	**mcupcase**	converts lowercase characters in a string to uppercase
not available	**nchars**	indicates the starting and ending character positions (1-based) and length of a Fortran string
stralloc	not	concatenates a variable number of C character strings into a single

	available	string pointer

These string manipulation functions are further described below with sample code fragments. Functions that perform similar tasks are grouped together, based on whether they convert strings, analyze strings, build strings or serve as string utilities.

For more information about these string manipulation functions, see the online man pages provided with the McIDAS-X software.

Converting string formats to integer values

The **Mcstrtoint** function converts a variety of string formats to integer values and returns an error status value less than zero when the calling routine enters an invalid string. To get a list of the string formats, enter the command ARGHELP INTEGER from the McIDAS-X command line.

To convert a hexadecimal string with **Mcstrtoint**, prefix the string with a dollar sign ($). When converting exponential representations with **Mcstrtoint**, be aware that it returns an error status if the string can't be completely represented as an integer. The sample code below converts several string formats to integer values.

```
{ char src_string[20]; int val; int status; /* * -- convert the character string '4321' to
an integer */ (void) strcpy (src_string, "4321"); status = Mcstrtoint (src_string, &val);
/* upon exit, the integer value stored in 'val' will be 4321 */ /* * -- convert the
hexadecimal character string '2f' to an integer */ (void) strcpy (src_string, "$2f");
status = Mcstrtoint (src_string, &val); /* upon exit, the integer value stored in 'val'
will be 47 */ /* * -- convert the exponential character string '234e3' to an integer */
(void) strcpy (src_string, "234e3"); status = Mcstrtoint (src_string, &val); /* upon exit,
the integer value stored in 'val' will be 234000 */ /* * -- convert the exponential
character string 2340e-1 to an integer */ (void) strcpy (src_string, "2340e-1"); status =
Mcstrtoint (src_string, &val); /* upon exit, the integer value stored in 'val' will be 234
*/ /* * -- convert the exponential character string 2340e-2 to an integer */ (void) strcpy
(src_string, "2340e-2"); status = Mcstrtoint (src_string, &val); /* * this call will
return an error because the actual value '23.4' * cannot be represented as an integer
variable */ }
```

Converting string formats to double-precision values

The **Mcstrtodbl** function converts a variety of string formats to double- precision values and returns an error status value less than zero when the calling routine enters an invalid string. To get a list of the string formats, enter the command ARGHELP DECIMAL on the McIDAS-X command line. The sample code below shows how to use **Mcstrtodbl**.

```
{ char src_string[20]; double val; int status; /* * -- convert the character string
'-4321.1' to a double */ (void) strcpy (src_string, "-4321.1"); status = Mcstrtodbl
(src_string, &val); /* upon exit, the value stored in 'val' will be -4321.1 */ /* * --
convert the exponential character string '321.2-e3' to a double */ (void) strcpy
(src_string, "321.2e-3"); status = Mcstrtodbl (src_string, &val); /* upon exit, the value
stored in 'val' will be .3212 */ }
```

Converting time and latitude/longitude formats

The four functions below convert a variety of McIDAS-X time and latitude/longitude formats to double-precision and integer values.

- **Mcstrtodhr** returns time values in units of hours as a double- precision value.
- **Mcstrtoihr** returns an integer representation of time in the format *hhmmss*.
- **Mcstrtodll** returns latitude or longitude in double-precision values.
- **Mcstrtoill** returns a scaled integer representation of latitude and longitude in the integer format *dddmmss*.

To get a list of the time formats, enter the command ARGHELP TIME on the McIDAS-X command line. To get a list of the latitude/longitude formats, enter the command ARGHELP LATLON.

If your input string for **Mcstrtodhr** or **Mcstrtoihr** is the NULL string or colon (:), the value returned is the current system time. If you always want the current system time, use the McIDAS-X library function **Mcgettime**.

The calling sequence and valid string formats for **Mcstrtodll** and **Mcstrtoill** are identical to **Mcstrtodhr** and **Mcstrtoihr**, except the former will not return the current system time if the NULL string or colon string are passed as input.

```
{ char src_string[20]; double dval; int val; int status; /* * -- convert the string
'421.1321' to a time */ (void) strcpy (src_string, "421.1321"); status = Mcstrtodhr
(src_string, &dval); status = Mcstrtoihr (src_string, &val); /* * upon exit, 'dval' will
contain 421.321 and 'val' will contain * 4210756 */ /* * -- convert the string '21:32:12'
to a time */ (void) strcpy (src_string, "21:32:12"); status = Mcstrtodhr (src_string,
&dval); status = Mcstrtoihr (src_string, &val); /* * upon exit, 'dval' will contain 21.536
and 'val' will contain 213212 */ /* * -- get current system time by entering the special
character ":" */ (void) strcpy (src_string, ":"); status = Mcstrtodhr (src_string, &dval);
```

```
status = Mcstrtoihr (src_string, &val); /* * upon exit, 'dval' and 'val' will contain the
current system time * in their respective formats */ }
```

Converting time to hours, minutes, seconds

The **Mcstrtohms** function converts a character string of the format *hh:mm:ss* into its components. It returns the current time if the input string is the NULL string or colon (:) and returns an error status value less than zero when the calling routine enters an invalid string. To list the valid time formats, enter the command ARGHELP TIME on the McIDAS-X command line.

The sample code below shows how to use **Mcstrtohms**.

```
{ int hour, minute, sec; int status; char src_string[20]; /* * -- convert the string
'21:32:12' to its components; hours, * minutes and seconds */ (void) strcpy (src_string,
"21:32:12"); status = Mcstrtohms (src_string, &hour, &minute, &sec); /* * upon exit, hour
= 21, minute = 32 and sec = 12 */ }
```

Converting a date in string format to a Julian day

The **Mcstrtoiyd** function converts a character string for a given day to a Julian day with integer value representation.

The format is *ccyyddd*, where:

- *cc* is the century
- *yy* is the year in the century
- *ddd* is the day number for the year, 1-based

It returns the current Julian day if your input string is the NULL string or slash (/) and returns an error status value less than zero when the calling routine enters an invalid string.

If your entered string does not include the century, the current century is assumed. If you always want the current system day, use the McIDAS-X library function **Mcgetday**.

To list the valid date formats, enter the command ARGHELP DATE on the McIDAS-X command line.

The sample code below shows the use of the **Mcstrtoiyd** function.

```
{ int day; int status; char src_string[20]; /* * -- convert the string "03-APR-1995" to a
Julian day */ (void) strcpy (src_string, "03-APR-1995"); status = Mcstrtoiyd (src_string,
&day); /* * upon exit, 'day' will contain 1995093 */ /* * -- convert the string "96/04/13"
to a Julian day */ (void) strcpy (src_string, "96/04/13"); status = Mcstrtoiyd
(src_string, &day); /* * upon exit, 'day' will contain 1996104, assuming you are *
currently in the 1900s */ /* * -- convert the string "1995366" to a Julian day */ (void)
strcpy (src_string, "1995366"); status = Mcstrtoiyd (src_string, &day); /* * status will
be less than 0 because there was no day 366 * in 1995 */ }
```

For more information about dates and time, see the section *Day and time* in this chapter.

Converting a C character string to Fortran

Mcstrtofs converts a C character string to a Fortran character string. If the source string is shorter than the destination string, the remaining bytes in the destination string are filled with the ASCII space character. If the C string is longer than the Fortran string, it is truncated in the Fortran string.

```
character*24 fullname call getname (fullname) call sdest (fullname,0) : :
```

```
void getname_ (char *fname, FsLen len_name) { char string[72]; int status; (void) strcpy
(string, "Ted Williams"); status = Mcstrtofs (fname, string, len_name); return; }
```

Converting a Fortran character string to C

The **fsalloc** function, which is called from a C routine, performs these tasks:

- Determines the amount of memory required to store the string
- Allocates that amount of memory
- Converts the Fortran character string to a C character string
- Assigns the new string to a character pointer variable

The **fsalloc** function does not allocate memory to store whitespace bytes at the end of the original source string. Because this function dynamically allocates memory, the calling program is responsible for freeing up the memory when finished with it. If allocation fails, the returned value is **`(char *) NULL`**. Calling routines should always test for this condition.

The code fragment below shows you how to use **fsalloc**.

```
character*24 fullname data fullname/'Joe Dimaggio'/ call printhof (fullname) : : void
printhof_ (char *fname, FsLen len_name) { char *cname; cname = fsalloc (fname, len_name);
/* * if the memory allocation is successful, 'cname' will point * to 13 bytes of memory
that have been filled with the * characters 'Joe Dimaggio' for the first 12 bytes and the
* NULL byte for the final value. note that it will not * allocate the full 24 bytes of
memory for this example * because the final 12 bytes of the original character * string
are whitespace characters. */ if (cname != (char *) NULL) { Mcprintf ("Hall of Famer:
%s\n", cname); free (cname); } return; }
```

Determining the contents of a string

The **ischar** function tells the calling routine if a string of four characters is from the printable ASCII character set, including the space character. The **isdgch** function tells the calling routine if a string of characters contains all digits, all letters, a mixture of digits and letters, or other characters. The sample code below shows the use of both functions.

```
character*4 cvar4 character*12 cvar12 integer status cvar4 = 'AbcD' status = ischar
(cvar4) c--- upon return, 'status' will contain the value 1 indicating c--- that all 4
characters are printable. cvar12 = '3214as231G' status = isdgch (cvar12) c--- upon
return,'status' will contain the value 0 indicating c--- that the string contains a
mixture of letters and digits.
```

Determining the number of characters in a string

The **nchars** function returns the location of the first and last non-blank characters (1-based) along with the length of the printable characters in a string. This function does not consider space characters printable when determining the values to return for starting and ending positions.

```
character*12 cvar12 integer status integer begchr integer endchr integer length cvar12 = '
Ty Cobb' length = nchars (cvar12, begchr, endchr) c--- upon exit, 'length' contains the
value 7, 'begchr' c--- contains the value 3, and 'endchr' contains the value 9.
```

Comparing two strings

Mcstricmp performs a case-insensitive string comparison between two entire strings. **Mcstrnicmp** performs a similar comparison, but only for the number of bytes specified by the calling routine. Both functions return a value less than zero if the lexical value of the first string is less than that of the second string. They return a value greater than zero if the lexical value in the first string is greater than that of the second string. They return the value zero if the strings are identical.

```
{ char string1[20]; char string2[20]; int status; (void) strcpy (string1, "BOB GIBSON");
(void) strcpy (string2, "bob feller"); /* compare the contents of 'string1' and `string2'
*/ status = Mcstricmp (string1, string2); /* * because the string "bob feller" occurs
earlier in an * alphabetic listing than "BOB GIBSON" the value of * 'status' would be
greater than 0. */ /* * compare the first 3 characters of the variables * 'string1' and
'string2' */ status = Mcstrnicmp (string1, string2, 3); /* * because the first 3
characters of both strings contain * the word "Bob", the value of 'status' would be 0. */
}
```

Concatenating a series of strings

The **stralloc** function has a variable number of calling parameters that concatenate a series of strings into a new string. The final parameter in the calling sequence must be the NULL pointer.

This function dynamically allocates memory for the resulting string. The calling function must free that block of memory when it is finished.

```
{ char t_string[20]; char *string1; (void) strcpy (t_string, "WISCONSIN"); string1 =
stralloc ("ST", "=", t_string, (char *) NULL); /* * upon successful exit the 'string1'
would point to a block * of memory that is 13 bytes long and contain the following *
string: "ST=WISCONSIN" */ if (string1 != (char *) NULL) { Mcprintf ("%s\n", string1); free
(string1); } string1 = stralloc ("CO=", "US", " ", "MX", " ", "UK", (char *) NULL); /* *
upon successful exit the 'string1' would point to a block * of memory that is 12 bytes
long and contain the following string: * "CO=US MX UK" */ if (string1 != (char *) NULL) {
Mcprintf ("%s\n", string1); free (string1); } return; }
```

Converting lowercase and uppercase characters

The **Mclocase** and **Mcupcase** functions convert the letters of a character string to all lowercase or uppercase respectively, as shown in the sample code below. Non-alphabetic characters in the string are unmodified.

```
{ char name[20]; (void) strcpy (name, "Hank Aaron!!"); /* convert the contents of the
character array 'name' * to lowercase */ Mclocase (name); /* * upon exit, the content of
the character array 'name' * will be: "hank aaron!!" */ /* convert the contents of the
character array 'name' * to uppercase */ Mcupcase (name); /* * upon exit, the content of
the character array 'name' * will be: "HANK AARON!!" */ }
```

Day and time

Day and Time conversion utilities consist of the follwing:

- Retrieving the current system day and time
- Converting a 5-digit Julian day to 7-digit, and vice versa
- Determining if a year is a leap year
- Converting to and from dates in IYD and CCYYDDD format
- Converting a Julian day and time to seconds, and vice versa
- Converting a Julian day to day/month/year, and vice versa
- Converting a Julian day to a character string
- Incrementing day and time

All McIDAS-X applications use the Julian day format. To correctly represent both the twentieth and twenty-first centuries, the McIDAS-X library provides Julian day manipulation routines using the format *ccyyddd* where *cc* represents the century, *yy* is the year of the century and *ddd* is the day of the year, with January 1 being day one. For example, 17 January 1997 is represented as 1997017.

In addition, there is one more variant of the *yyyddd* format. This format represents the julian day of the year 1900+yyy. As examples, 99119 is day 119 of the year 1999, and 102231 is day 231 of the year 2002. This format is used in many McIDAS-X data structures for internal storage, but is usually coverted to *ccyyddd* for user listings.

Some of the functions described in this section still expect the older Julian day format, *yyddd*. However, they will be replaced with the new format.

The time utilities described in this section use the integer representation *hhmmss* for time and return the time in UTC, Coordinated Universal Time.

Below is an alphabetical listing of the day and time functions available in the McIDAS-X library, along with a short description.

C function	Fortran function	Description
Mccydok	**mccydok**	verifies that a Julian day in the form *ccyyddd* is correct
Mccydtodmy	**mccydtodmy**	converts a Julian day in the form *ccyyddd* to day, month and year
Mccydtodow	**mccydtodow**	converts date to day of the week
Mccydtoiyd	**mccydtoiyd**	converts date in *ccyyddd* format to *yyyddd*
Mccydtostr	**mccydtostr**	converts the Julian day in the form *ccyyddd* to a variety of character string formats
Mccydtoyd	**mccydtoyd**	converts a 7-digit Julian day to 5 digits
Mcdaytimetosec	**mcdaytimetosec**	converts a Julian day in the form *ccyyddd* and the time of day in the form *hhmmss* to seconds since 1 January 1970 at 00 UTC
Mcdhrtoihr	not available	converts hours stored in double precision to hours stored in an integer of the form *hhmmss*
Mcdmytocyd	**mcdmytocyd**	converts day, month and year to a Julian day in the form *ccyyddd*
Mcgetday	**mcgetday**	gets the current system Julian day in the form *ccyyddd*
Mcgetdaytime	**mcgetdaytime**	gets the current Julian day in the form *ccyyddd* and the current time of day in the form *hhmmss*

Mcgettim	**gettime**	gets the current time of day in the form *hhmmss*
Mchmsok	**mchmsok**	verifies that a time value in the form *hhmmss* is correct
Mchmstoihr	not available	converts hours, minutes and seconds to *hhmmss* format
Mchmstostr	**mchmstostr**	converts a time in the form *hhmmss* to a variety of character string formats
Mcincday	**mcincday**	increments/decrements a Julian day value
Mcinctime	**mcinctime**	increments/decrements a Julian day and time value
Mcisleap	**mcisleap**	checks to see if the four-digit field represents a leap year
Mcistimeofday	**mcistimeofday**	verifies the value given to be a valid time of day
Mciydtocyd	**mciydtocyd**	converts date in *yyyddd* format to *ccyyddd*
Mcsectodaytime	**mcsectodaytime**	converts seconds since 1 January 1970 to a Julian day in the form *ccyyddd* and a time in the form *hhmmss*
Mcydtocyd	**mcydtocyd**	converts a 5-digit Julian day to 7 digits

These functions are further defined below along with examples of sample code. The more commonly used functions are described first.

For more information on these day and time functions, see the online man pages provided with the McIDAS-X software.

Retrieving the current system day and time

The McIDAS-X library contains these three current day/time functions:

- **Mcgetday** retrieves the current system Julian day.
- **Mcgettime** returns the current system time in UTC.
- **Mcgetdaytime** retrieves both.

If you write applications that need both current day and time, use **Mcgetdaytime**. This function doesn't have the potential timing problem associated with retrieving the information in separate calls. This timing problem surfaces just before the Julian day changes. For example, if you call **Mcgetday** at 23:59:59 on a particular day but don't call **Mcgettime** until two seconds later, your day and time won't match because your time will read 00:00:01.

The sample code below shows the use of all three functions.

```
{ int CurrentDay; int CurrentTime; int status; /* get current system day and time. this is
the undesirable * manner */ status = Mcgetday (&CurrentDay); status = Mcgettime
(&CurrentTime); /* * CurrentDay will contain the current system day in the form * ccyyddd.
CurrentTime will contain the current time in the * form hhmmss */ /* get current system
day and time. this is the preferred * manner */ status = Mcgetdaytime (&CurrentDay,
&CurrentTime); }
```

Converting a 5-digit Julian day to 7-digit, and vice versa

You can use the **Mccydtoyd** and **Mcydtocyd** functions to transition your code from the previous 5-digit Julian day format, *yyddd*, to the new 7-digit format, *ccyyddd*, and vice versa.

- **Mccydtoyd** converts the 7-digit Julian day format to the 5-digit representation.
- **Mcydtocyd** converts the 5-digit Julian day format to the 7-digit version.

The **Mcydtocyd** function assumes that if the year of the century (*yy*) in the 5-digit format is less than or equal to 69, it is the 21st century; if the year of the century is greater than or equal to 70, it assumes the 20th century.

See the sample code below.

```
{ int status; int new_day; /* convert the day 1996017 to the 5 digit form of the Julian
day */ status = Mccydtoyd (1996017, &new_day); /* if successful, the contents of new_day
will be 96017 */ /* convert the day 96100 to the 7 digit form of the Julian day */ status
= Mcydtocyd (96100, &new_day); /* if successful, the contents of new_day will be 1996100
```

```
*/ /* convert the day 11100 to the 7 digit form of the Julian day */ status = Mcydtocyd
(11100, &new_day); /* if successful, the contents of new_day will be 2011100 */ }
```

Converting to and from dates in IYD and CCYYDDD format

When servers present grid image or point data, the date fields are usually passed in a *yyyddd* format, where *yyy* represents years since 1900 and *ddd* represents the day count from the beginning of the year (1-based).

When the data is presented, the dates are in *ccyyddd* format, where *cc* is the century field, *yy* is the year of the century field and *ddd* is the julian day of the year.

Two functions are used to convert the data: **mciydtocyd**, which converts from the *yyyddd* format to the *ccyyddd* format, and **mccydtoiyd**, which converts from the *ccyyddd* format to the *yyyddd* format.

The sample code below, from **m0ptrdhdr**, shows how to use **Mciydtocyd**.

```
c c--- modify schema and creation date fields c istat=mciydtocyd(header(3),temp_date)
header(3)=temp_date istat=mciydtocyd(header(26),temp_date) header(26)=temp_date
```

The sample code below, from **m0makara**, shows how to use **mccydtoiyd**.

```
C --- Get, convert current ccyyddd to yyyddd C --- Store time directly.
ival=mcgetdaytime(century_date,aradir(18)) if(ival.lt.0) then call edest('Unable to write
creation date',0) m0makara=-3 endif ival = mccydtoiyd(century_date,adir_date) aradir(17) =
adir_date
```

Converting a Julian day and time to seconds, and vice versa

Mcdaytimetosec and **Mcsectodaytime** convert Julian day and time to absolute times based from 1 January 1970 at 00 UTC, and vice versa. This standard is different from the McIDAS-X functions **sksecs** and **skhms**, which use 1 January 1972 as the base.

```
#include <time.h { int day; int time; time_t seconds; int status; /* * get the number of
seconds since 1 January 1970 for Julian * day 1997017 at 12:30UTC */ day = 1997017; time =
123000; status = Mcdaytimetosec (day, time, &seconds); /* upon successful exit, the value
of 'seconds' will be * 853584200 */ /* * convert 853584200 seconds since 1 January 1970 to
a * Julian day and time */ status = Mcsectodaytime (seconds, &day, &time); /* * upon
successful exit, the value of 'day' will be * 1997017 and the value of 'time' will be
123000 */ }
```

Converting a Julian day to day/month/year, and vice versa

Mccydtodmy converts a Julian day to day, month and year, including the century. The month number returned is 1-based, meaning January is 1. **Mcdmytocyd** converts a day, month, year combination to a Julian day. Examples of both functions are shown in the code fragment below.

```
{ int dayofmonth; int month; int year; int day; int status; /* convert the day 9 May 1996
to a Julian day */ dayofmonth = 9; month = 5; year = 1996; status = Mcdmytocyd
(dayofmonth, month, year, &day); /* upon successful completion 'day' will contain the *
value 1996130 */ /* convert the day 29 February 1995 to a Julian day */ dayofmonth = 29;
month = 2; year = 1995; status = Mcdmytocyd (dayofmonth, month, year, &day); /* this
conversion will fail because there was no * 29 February 1995 */ /* convert the Julian day
1996060 to a day, month and year */ day = 1996060; status = Mccydtodmy (day, &dayofmonth,
&month, &year); /* * upon successful completion 'dayofmonth' will contain 29, * 'month'
will contain 2 and 'year' will contain 1996 */ }
```

Converting a Julian day to a character string

Mccydtostr converts a Julian day to a variety of character strings representing the date, as shown in the code sample below.

```
{ int status; char *day_string; /* * convert the Julian day 1996017 to a character string
of the * form nn mmm, ccyy */ status = Mccydtostr (1996017, 4, &day_string); /* * upon
successful completion, the contents of the variable * day_string will be '17 Jan, 1996'.
For a complete listing * of the output formats available, see the McIDAS API * Reference
Manual. */ }
```

Determining if a year is a leap year

You can use **Mcisleap** to determine if a four-digit year is a leap year. See the sample code below.

```
```

```
iam=-1 c --- get the current day i=mcgetday(kurrent_day) kurrent_year=kurrent_day/1000
iam=mcisleap(kurrent_year) if(iam.eq.0) then call sdest('Current year is not a leap
year',0) elseif(iam.eq.1) then call sdest('Current year is a leap year',0) endif
```

Incrementing day and time

When writing applications for real-time data, you may need to increment day and time parameters to locate data for an application. However, when you work with the beginning or ending day of a particular year, you can't just add a *one* to a Julian day and expect the correct resulting date. Use the **Mcincday** function to increment and decrement a Julian day by days. Use the **Mcinctime** function to increment and decrement a day/time pair by a time increment. A positive increment value results in a future time; negative numbers result in a past time.

```
{ int new_day; int new_time; int status; /* increment the Julian day 1996364 by 3 days */
status = Mcincday (1996364, 3, &new_day); /* * upon successful completion, 'new_day' will
contain * the value 1997001 */ /* decrement the day/time pair 1996002/12UTC by 78 hours */
status = Mcinctime (1996002, 120000, -780000, &new_day, &new_time); /* * upon successful
completion, 'new_day' will contain the * value 1995364 and 'new_time' will contain the
value 60000 */ }
```

Latitude and longitude

The McIDAS-X library contains two utilities for converting latitude and longitude values:

- **flalo** converts an integer representation of latitude or longitude in the format *dddmmss* to a single-precision representation, in degrees.
- **ilalo** converts a single-precision latitude or longitude to a scaled- integer value in the format *dddmmss.*

Because workstations can represent floating-point values differently, most data requiring fractional representation, such as latitude and longitude, is stored as scaled integers. While scaled integers are adequate for data storage, performing mathematical operations on these values is difficult.

Use the **flalo** function to convert the scaled-integer representation of latitude and longitude into single-precision floating point numbers. Use the **ilalo** function to convert single-precision floating-point values to scaled-integer representation. The sample code below demonstrates the latitude and longitude conversion functions.

```
integer lat real flat c--- convert 59 degrees 30 minutes to a single precision c---
floating point value lat = 593000 flat = flalo (lat) c--- upon successful completion,
'flat' contains the value 59.5 c--- convert the latitude value 45.75 to a scaled integer
value flat = 45.75 lat = ilalo (flat) c--- upon successful completion, 'lat' contains the
value 454500
```

For more information about these conversion functions, see the online man pages provided with the McIDAS-X software.

Physical units

The McIDAS-X library contains two utilities for converting physical units:

- **mcucvtd** converts a list of double-precision values from one physical unit to a different physical unit.
- **mcucvtr** converts a list of single-precision values from one physical unit to a different physical unit.

Unit conversion is a integral part of any user application because data is often stored in units that a user won't typically display. The **mcucvtd** and **mcucvtr** functions can convert the physical units shown in the table below; all McIDAS-X core datasets use this standard.

Attribute	Valid units	Interface representation
length	meters kilometers decameters centimeters millimeters miles nautical miles yards feet inches degrees of latitude	M KM DM CM MM MI NMI YD FT IN DEGL

speed	miles per hour knots meters per second feet per second kilometers per hour	MPH KT or KTS MPS FPS KPH
temperature	Kelvin Fahrenheit Celsius	K F C
pressure	millibars inches of Mercury pascals hectopascals	MB INHG PA HPA
time	hours minutes seconds days years	HR MIN SEC DAY YR
weight	grams kilograms pounds ounces tons	G KG LB OZ TON

The code segment below demonstrates these unit conversion routines.

```
integer numval character*4 inunit character*4 outunit integer status double precision
outval numval = 1 c--- Convert 50 degrees fahrenheit to celsius inunit = 'F' outunit = 'C'
status = mcucvtd (numval, inunit, 50.d0, outunit, outval, 0) c--- Upon successful
completion, 'outval' will contain the c--- value 10.0. c--- Convert 50 degrees fahrenheit
to feet outunit = 'FT' status = mcucvtd (numval, inunit, 50.d0, outunit, outval, 0) c---
the value of 'status' will be -1 because you cannot c--- convert degrees fahrenheit to
feet.
```

For more information about these conversion functions, see the online man pages provided with the McIDAS-X software.

Scientific utilities

The McIDAS-X library provides a set of scientific utilities for computing meteorological parameters such as potential temperature, equivalent potential temperature, and mixing ratio.

The table below provides an alphabetical listing of the scientific utilities provided in McIDAS-X.

C function	Fortran function	Description
not available	**lab**	computes potential and equivalent potential temperature and mixing ratio, given the temperature, dew point, pressure, and station elevation
McAdvectParm	**mcadvectparm**	advects a gridded parameter
McBeta	**mcbeta**	computes beta parameter $\frac{\partial f}{\partial y} = \frac{2\Omega cos\phi}{a}$
McCape	**mccape**	computes Convective Available Potential Energy
McCoriolis	**mccorfor**	computes the coriolis parameter ($f=2\Omega \sin\Phi$)
McDewpt	**mcdewpt**	computes dewpoint
McDirec	**mcdirec**	computes meteorological direction of wind for u- and v-components
McDiver	**mcdiver**	computes divergence $\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$
McDivergeParm	**mcdivergeparm**	compute divergence of gridded parameter
McHeatIndex	**mcheatindex**	computes the heat index, given the temperature and dewpoint
McHelic	**mchelic**	computes helicity
McHypsoP	**mchypsop**	integrates hypsometric equation to compute a pressure at a given height
McHypsoZ	**mchypsoz**	integrates hypsometric equation to compute a new height at a given pressure
McLatentHeat	**mclatvap**	latent heat of vaporization
McLiftCLevel	**mclcl**	parcel temperature and pressure at LCL
McMixing	**mcrmix**	mixing ratio
McPFromTheta	**mcpfromtheta**	returns pressure when given theta and temperature
McRelativeHumidity	**mcrelativehumidity**	computes the relative humidity, given the temperature and dewpoint
McSatVapor	**mcsatvap**	saturation vapor pressure over water
McSatVaporIce	**mcsatvapi**	saturation vapor pressure over ice
McSheard	**mcsheard**	computes shear deformation $\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}$
McSndAnl	**mcsndanl**	computes various stability parameters, given a

		sounding
McSpeed	**mcspeed**	computes wind speed from u- and v-components
McStationPres	**mcstnpres**	station pressure, given altimeter and station elevation
McStretd	**mcstretd**	computes the stretching deformation $\frac{\partial u}{\partial x} - \frac{\partial v}{\partial y}$
McTDFromMixing	**mctamr**	computes dew point from mixing ratio
McTempAtThetae	**mctasa**	temperature along saturated adiabats
McTFromTheta	**mctfromtheta**	returns temperature when given theta and pressure
McTheta	**mctheta**	potential temperature
McThetae	**mcthetae**	equivalent potential temperature
McThetaw	**mcthetaw**	wet bulb potential temperature
McUandV	**mcuandv**	computes wind u- and v-components
McVirtTemp	**mcvirttemp**	virtual temperature
McVort	**mcvort**	computes vorticity $\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$
McWetBulb	**mcwetbulb**	wet bulb temperature
McWindChill	**mcwindchill**	computes the wind chill, given the temperature and wind speed (old formula)
McWindChill2001	**mcwindchill2001**	computes the wind chill, given the temperature and wind speed (uses the 2001 algorithm)
not available	**rmix**	determines the mixing ratio, given the temperature and pressure

For more information about these scientific functions, see the online man pages provided with the McIDAS-X software.

Computing isentropic surfaces

Given the temperature, dew point and pressure, the **lab** function computes the following:

- Potential temperature
- Equivalent potential temperature
- Mixing ratio

To compute these meteorological parameters using the pressure at the station instead of the reported pressure, which has been corrected to sea level, the **lab** function will automatically adjust for the pressure differences due to elevation.

The following example computes potential and equivalent potential temperatures, and mixing ratio with adjustments made to the pressure based on the height.

```
double precision derived(3) ! derived parameters double precision dewpt ! dewpt in K
double precision elev ! elevation in meters double precision miss ! missing data code
double precision press ! pressure in mb double precision temp ! temperature in K temp =
292.d0 dewpt= 289.d0 press=1018.29 miss=1.e35 elev=214 C the adjustment based on elevation
is done call lab(temp,dewpt,press,elev,1,miss,derived,3) C this will return theta-e in
derived(1), theta in derived(1) C and mixing ratio in derived(3) C the values returned
```

```
will be: C derived(1) = 323.115 C derived(2) = 292.612 C derived(3) = 11.521 C the
adjustment based on elevation is not done call lab(temp,dewpt,press,elev,0,miss,derived,3)
C the values returned will be: C derived(1) = 319.961 C derived(2) = 290.491 C derived(3)
= 11.226
```

Computing Heat Index

Given the temperature and dew point, the functions **McHeatIndex** and **mcheatindex** will compute the heat index. The value returned will be given in the same units as the input temperature. Below is an example of the function **mcheatindex**.

```
integer ok double precision temperature double precision dewpoint double precision heatindex
temperature = 30.d0 dewpoint = 20.d0 ok = mcheatindex (temperature, dewpoint, 'C',
heatindex) if (ok .lt. 0)then call sdest('error calculating the heat index',0) return endif

c--- upon successful completion the value of heatindex will c--- be 32.5 degrees Celsius
```

Computing Relative Humidity

Given the temperature and dew point, the functions **McRelativeHumidity** and **mcrelativehumidity** will compute the relative humidity. The value returned will be given in the dimensionless units of percentage. Below is an example of the function **mcrelativehumidity**.

```
integer ok double precision temperature double precision dewpoint double precision rh

temperature = 30.d0 dewpoint = 20.d0

ok = mcrelativehumidity (temperature, dewpoint, 'C', rh) if (ok .lt. 0)then call
sdest('error calculating the relative humidity',0) return endif

c--- upon successful completion the value of rh will be 55.1
```

Computing Wind Chill

Given the temperature and wind speed, the functions **McWindChill** and **mcwindchill** will compute the wind chill. The value returned will be given in the same units as the input temperature. Below is an example of the **mcwindchill** function.

```
integer ok double precision temperature double precision windspeed double precision
windchill

temperature = 20.d0 windspeed = 10.d0

ok = mcwindchill (temperature, windspeed, 'F', 'MPS', & windchill)

if (ok .lt. 0)then call sdest('error calculating the wind chill',0) return endif

c---upon successful completion the value of windchill will c---be -15 degrees Fahrenheit
```

Computing mixing ratio

Given a temperature, in Kelvin, and the pressure, in mb, the **rmix** function will return the mixing ratio. The returned value, ws, is defined as the mass of water vapor per mass of dry air. It is dimensionless, with units of g/kg. The sample code below computes the saturation mixing ratio at 0° Celsius for all pressures 100, 150, 200, 250, ... 950, 1000 mb, resulting in 40.44, 26.39, 19.59, 15.57, ... 4.02, 3.82 g/kg.

```
integer i real temperature real pressure real mix temperature=273.15 do 100 i=1000,100,-50
pressure=float(i) mix=rmix(temperature,pressure) 100 continue
```

Computing stability parameters

Given vertical profiles of pressure, temperature, dew point, wind speed and direction, the **sndanl** function computes these stability indices:

- Parcel dew point, potential temperature, equivalent potential temperature and mixing ratio
- Precipitable water
- Convective temperature and forecast maximum temperature (for data from 1200 UTC only)
- Lifted index, total-totals index, K-index and sweat index
- Equilibrium pressure level

Below is an example of the **sndanl** function.

```
parameter (NLEV = 9) real press(NLEV) real temp(NLEV) real dewpt(NLEV) real dir(NLEV) real
spd(NLEV) real stabil(12) c--- get sounding data : c--- assume the arrays; press, temp,
dewpt, dir and spd c--- have been initialized with the following values c--- press(mb)
temp(K) dewpt(K) dir(DEG) spd(MPS) c--- 988.0 300.0 299.8 250 1.5 c--- 925.0 303.0 280.0
270 3.8 c--- 850.0 297.7 269.0 285 5.6 c--- 700.0 277.5 276.1 290 12.3 c--- 500.0 262.1
252.1 265 13.3 c--- 400.0 253.6 240.5 280 29.8 c--- 300.0 237.6 226.0 280 33.4 c--- 250.0
229.7 215.7 285 41.6 c--- 200.0 216.3 206.3 285 45.7 c--- perform the sounding analysis
for 00Z call sndanl (0, NLEV, press, temp, dewpt, dir, spd, stabil) c--- upon return, the
values stored in stabil will be as follows: c--- stabil(1) 282.4 parcel dewpoint (k) c---
stabil(2) 301.1 potential temperature (k) c--- stabil(3) 324.1 equivalent potential
temperature (k) c--- stabil(4) 8.3 mixing ratio (g/kg) c--- stabil(5) 23.7 precipitable
water (mm) c--- stabil(6) 32.7 convective temperature (c) c--- stabil(7) 26.9 forecast
maximum temperature (c) c--- stabil(8) 2.7 lifted index (k) c--- stabil(9) 33.4 total
totals c--- stabil(10) 445.5 equivalent pressure (mb) c--- stabil(11) 20.6 k index c---
stabil(12) 47.6 sweat index
```

Chapter 5

Accessing Data

This chapter provides the information you will need to access and use McIDAS-X data. You'll learn:

- The attributes unique to each of the McIDAS-X data types: disk files, images, grids, point observations and text
- The API functions available for reading, writing and deleting McIDAS-X data
- The blocks of information contained in an image, grid, point observation or text
- How to use selection clauses to restrict a data search
- How the ability to integrate and display a variety of datasets is made possible by its navigation and calibration subsystems

This chapter is organized into these sections:

McIDAS disk files

Reading and writing data to and from disk is fundamental to many applications programs. A McIDAS-X disk file stores information that applications can randomly access by byte address using standard system library calls.

This section describes:

- Special attributes of McIDAS-X disk files that distinguish them from other system files
- API functions you will use to read, write, delete, and copy disk files

Basic concepts

McIDAS-X disk file utilities have several characteristics that distinguish them from other file system utilities.

- *Disk files are always open.* From an application's perspective, a McIDAS-X disk file is always available for use. Since the application doesn't need to open, close or otherwise position the file to perform input or output, it can treat disk files as virtual arrays of bytes.
- *Disk files are byte-oriented.* This attribute applies to both the location of data on disk and the amount of data moved to or from disk. You can use higher-level APIs to transfer words, which are 4-byte groups. Four bytes is the common length of Fortran INTEGER and REAL variables.
- *Disk files use zero-based addressing.* The first byte in every disk file is byte number 0, not byte 1.
- *Disk files contain a unique, missing-data value.* When reading bytes in a disk file that have never been written, a unique value (hexadecimal 0x80) is returned.
- *Zero-length disk files are automatically deleted.* When a McIDAS-X command ends, all writable, zero-length files accessed while the program was running are deleted.
- *Disk file name length.* The McIDAS-X disk file I/O subsystem has no inherent name length limitation.
- *Disk files may exist in a variety of directories*, depending on the settings that the user enters for the session's **`MCPATH`** environment variable, as well as the entries in the REDIRECT table. When accessing files, do not impose a complete pathname. Rather, assign only a file name that can then be converted into a fully qualified pathname/filename by the McIDAS-X file system.

  If the file name passed from the application to the McIDAS-X disk file API functions does not contain a slash character, the McIDAS-X file system will perform the following three steps to determine a file's location on disk.

  1. Check the REDIRECT table entries and try to match the file name.
  2. If that fails, search for the file name in all directories named in the **`MCPATH`** environment variable.
  3. If that fails, choose the pathname of the first writable directory named in the **`MCPATH`** environment variable.

Disk file APIs

The table below describes the McIDAS-X library functions that you will use when programming with disk files. In McIDAS-X, a disk file is called an LW (Large Word) array file. You will notice that many of the Fortran APIs below begin with the letters **lw**.

C function	Fortran function	Description
Mcpathname	**volnam**	converts a disk file name to a fully qualified pathname/filename
Mcread	**lbi**	reads bytes from a disk file into memory
not available	**lwi**	reads words (4-byte groups) from a disk file
Mcwrite	**lbo**	creates a disk file by writing bytes into it from memory
not available	**lwo**	creates a disk file by writing words (4-byte groups) into it
Mcremove	**lwd**	deletes a disk file
Mctruncate	**lwtrunc**	deletes the contents of a disk file without deleting the file itself
not available	**lwfile**	determines if a file exists
not available	**lock**	acquires an exclusive lock on a file

not available	**unlock**	frees the lock on a file

Each function is described in the sections below, along with sample code illustrating its use.

Reading and writing disk files

The McIDAS-X library provides the **Mcread** and **Mcwrite** functions for reading and writing disk files in C. The comparable functions in Fortran are **lbi** and **lbo**, which are shown in the code fragment below. The **lwi** and **lwo** functions read and write words (4-byte groups) from a disk file.

```
c --- an example of writing data integer array_out(3) integer array_in(3) integer nwords,
status, lwo, lwi, first ... c --- initialize first = 0 nwords = 3 do 200 i = 1, nwords 200
array_out(i) = i * 100 c --- write the data to disk -- note that there are four bytes c --
- in each array element status = lbo('testdata', first*4, nwords*4, array_out) if (status
.lt. 0) then call edest('Failed to write testdata', status) call mccodeset(1) return endif
c --- at this point, the file 'testdata' will consist of: c --- word 0 = integer value 100
c --- word 1 = integer value 200 c --- word 2 = integer value 300 c --- words 3 and beyond
are unwritten c --- now, read the data in, skipping the first word: first = 1 status =
lbi('testdata', first*4, nwords*4, array_in) c --- at this point, the array array_in will
consist of: c --- word 1 = integer value 200 c --- word 2 = integer value 300 c --- word 3
= 0x80808080 (the missing value indicating c --- this word was never written)
```

Assigning a system pathname to a disk file

If your application uses Fortran or C library functions for disk I/O, you must use the **volnam** or **Mcpathname** function to convert the name of the disk file into a fully qualified, system *pathname/filename*. This pathname is essential for locating and working on a file.

The code fragment below illustrates the use of **volnam** with a Fortran OPEN statement. The maximum number of characters allowed for the fully-qualified pathname is stored in the constant **MAXPATHLENGTH** in the Fortran INCLUDE file, **fileparm.inc**. The limit for C is in the file **/usr/include/sys/limits.h**.

```
c --- For illustration, assume the user has done a REDIRECT c --- command that looks like
this: c --- REDIRECT ADD MYDATA "/home/me/mcidas/data c --- include 'fileparm.inc'
character*(MAXPATHLENGTH) fullname character*12 filename integer rc, volnam ... filename =
'MYDATA' ... rc = volnam(filename, fullname) if (rc.lt.0) then call edest('Problem
resolving path for '// filename,0) call mccodeset(1) return endif c --- At this point,
fullname will contain the fully-qualified c --- name ('/home/me/mcidas/data/MYDATA')
open(unit=12, file=fullname, mode='share', status='old') ...
```

Determining if a disk file exists

Use the function **lwfile** to determine if a file with a given name already exists. If the file does not exist, **lwfile** returns a zero; it does not create the file for you. The following code fragment illustrates how to use **lwfile**.

```
c --- find out if file 'mystuff' exists status = lwfile('mystuff') if (status .ne. 0) then
call sdest('The file is there!',0) else call edest('The file is NOT there!!!',0) endif ...
```

Deleting the contents of a disk file

To delete the contents of a disk file but not the file itself, use the **lwtrunc** or **Mctruncate** function. These functions remove the contents of the file, leaving one word (4 bytes) of 0x80808080 at the beginning of the file.

Deleting only the contents of a file and not the file itself is important if the file name appears in more than one location in the **MCPATH** tree. For example, if you delete the file ABC from a directory where you have write permissions but the file also exists further down your **MCPATH** in a directory where you have only read permissions, you won't be able to create a writable file by that name. If you use **lwtrunc** or **Mctruncate** to delete only the file's contents, you can write into it again in the future, since it resides in the original, writable directory.

The code fragment below uses the **lwtrunc** function to delete the contents of a file.

```
c --- assume the user's MCPATH contains the directories c --- /home/my/data and
/home/your/data, with a file named c --- DATAFILE residing in both directories; further
assume c --- that I only have write permissions to /home/my/data status =
lwtrunc('DATAFILE') if (status .lt. 0) then call edest('Error occurred trying to truncate
file',0) endif c --- now write the contents of buffer to DATAFILE status = lbo('DATAFILE',
0, bufsiz, buffer) c --- at this point, the contents of buffer were written to c ---
/home/my/data/DATAFILE; if lwd had been called instead of c --- lwtrunc, the lbo call
would have returned an error because c --- I don't have write permissions to
/home/your/data/DATAFILE
```

Deleting a disk file

Use the **lwd** or **Mcremove** functions to delete a disk file, as shown in the code fragment below.

```
... character filename*20 integer status, lwd c --- try to remove the file status =
lwd(filename) if (status .lt. 0) then call edest('Error trying to remove file '//
filename,0) call mccodeset(2) goto 999 else ... c --- do some other processing
```

Locking and unlocking a disk file

As long as a user has read and write permissions, the McIDAS-X disk file I/O subsystem will open all files and permit simultaneous access to these files for both reading and writing. If you write applications that update information in disk files, you must synchronize access to the file to avoid potential file collisions.

McIDAS-X has a locking mechanism called **lock** for coordinating file updates between applications or between copies of the same application. The **lock** function acquires exclusive use of a unique lock. If your application is using the lock, another application trying to lock the file must wait until you free the lock with the **unlock** function before using it.

The **lock** and **unlock** functions do not prevent other applications from reading a file or writing to it. They simply ensure an orderly means of updating a file without losing or overwriting information.

The code fragment below shows how to lock and unlock a file to protect its integrity during updating.

```
... c --- protect against simultaneous attempts to use this lock call lock( 'myfile ') c -
-- when control is returned, this application has exclusive use rc = lbi('myfile',
0,100,array) ... c --- update the values in array ... c --- now save the updated info back
into 'myfile' rc = lbo('myfile',0,100,array) c --- now free the lock call unlock('myfile')
...
```

Copying a disk file

Use the **lwcopy** function to copy one disk file into another, as shown in the code fragment below.

```
c --- set up some variables integer status, lwcopy, lwo integer source(3) integer
destination(3) c --- fill up source array do 200 i=1,3 source(i) = I*100 200 continue c --
- write out array to file "first" status = lwo('first', 0,3,source) if (status.lt.0) then
call edest('Error writing to "first" file',0) call mccodeset(2) goto 999 endif c --- now
copy file 'first' into 'backup' status = lwcopy('first', 'backup') if (status.lt.0) then
call edest('Error during copy...',status) call mccodeset(2) goto 999 endif ...
```

Image data

McIDAS-X images are typically composed of atmospheric and oceanographic data, which are measured from remote sensing platforms such as satellites and radar. Images may contain any data that can be represented in a two-dimensional matrix.

This section describes:

- Attributes that distinguish image data from other types of data
- Blocks of information contained in an image
- API functions available for reading, writing and deleting image data

Basic concepts

Image data has several unique attributes that determine how an image is displayed on the McIDAS-X Image Window. The resolution of the image, the size of the data points and the number of spectral bands all influence the final product.

Image resolution

A satellite observes features by scanning small slices of the earth's surface with each pass of the sensors. The geographic width of each image line helps determine the size of the smallest surface feature the satellite can detect. This concept is called *resolution*.

Image resolution refers to the number of satellite image lines represented in each data point of an image line. If the image resolution is one, the image is stored at *full resolution*. This means that one image data point represents one satellite sensor data point. If the image resolution is four, either sampling or averaging was performed on the image so that one data point in the image represents 16 satellite scan data points. Each satellite has its own scan resolution, so an image resolution of one will mean different geographic resolutions from one satellite to another.

When copying or displaying an image, a user can modify its resolution. Image resolution can be artificially increased, or *blown up*, by replicating data point values, much like enlarging a 3 x 5 photograph to 8 x 10. Image resolution can be decreased, or *blown down*, by sampling or averaging the image data points. For example, if a blowdown factor of two is applied to a McIDAS-X image, every other data point along the line and every other line in the image is dropped out. Each data point on the displayed image represents four data points from the original, scanned image.

Data-point size

Data-point size refers to the number of bytes needed to accurately represent the value of a data point. Although the size varies among sensor sources, data is one, two or four bytes. For example, Meteosat-7 data is 1-byte per data point while GOES-12 data is 2-byte.

Spectral bands

A spectral band is the wavelength in which a scanning instrument measures data; for example, band 4 for the GOES-8 Imager senses 10.7 micron wavelength radiation. These wavelengths are specific to the measuring instrument. Most satellites can measure radiation from many wavelengths simultaneously over the same geographic location.

For more information about the bands for satellite imagery, see *Appendix B, Satellite Information*.

What is an image object?

Image data is quantitatively useless unless it is transformed into physical units (calibration) and oriented relative to time and physical space (navigation). In addition, it is often necessary to know when and how an image was collected and processed. The actual image, along with these *ancillary data*, is collectively called an *image object*. Each image object in McIDAS-X is composed of the following blocks of information:

- The *directory block* contains a list of ancillary information about the image, such as the number of lines and data points, the satellite ID, and the number of spectral bands.
- The *data block* contains the matrix of image data values.
- The *line prefix block* contains information about an image that may vary on a line-by-line basis, such as calibration or documentation information.
- The *navigation block* contains information for determining the location of data points in physical space. More information about navigation is presented in the section titled *McIDAS navigation* later in this chapter.
- The *calibration block* contains the information for converting image data from its internal (stored) units to more meaningful physical units, such as radiance or albedo. More information about calibration is presented in the section titled *McIDAS calibration* later in this chapter.

- The *supplemental (or auxiliary) block* contains additional information that is specific to a data type. For example, information specific to radar data is stored in this block. Also, the latitude/longitude grid for the LALO navigation is stored in this block.
- The *comment block* contains a variety of textual information, such as a list of commands run on the image object to-date.

The API functions and the procedures for reading, writing and deleting image objects follow.

Reading image objects

Most applications for reading image objects will do one of the following:

- Open a connection to read only the directory block of an image object and then read the directory block
- Open a connection to read an image object, including the directory, data, line prefix, navigation, calibration and comment blocks

The table below lists alphabetically the McIDAS-X library functions for performing these tasks.

Function	Description
mcaaux	reads the auxiliary (supplemental) block of an image object
mcacal	reads the calibration block of an image object
mcacrd	reads the comment block of an image object
mcadir	opens a connection to read the directory block from an image object
mcadrd	reads the directory block from an image object
mcafree	frees the handle and memory of a connection opened by mcaget
mcaget	opens a connection to read the data block from an image object
mcalin	reads the data portion of the current image line
mcanav	reads the navigation block of an image object
mcapfx	reads the prefix portion of the current image line
mcasort	gets the parameters from the command line and adds them to the selection array for a future mcaget call
mcpcal	parses a list of valid calibration types from the comment cards
mcpnav	parses out geographic resolution information from the comment cards

These functions are described below along with sample code.

See the online man pages provided with the McIDAS-X software for detailed information about any of the API functions discussed in this section.

Opening a connection to read the directory block

In ADDE, a client has the ability to request only directory blocks from a server. This allows the client to sample information on a server without transferring large amounts of image data that may not be necessary for an application. For example, the McIDAS-X IMGLIST command reads only directory blocks.

The ADDE interface to the image directory is through **mcadir**, which opens a connection based on a set of selection clauses for a given dataset name. The valid selection conditions are provided in the table below.

Selection clause	Description
AUX YES or AUX NO	appends center latitude/longitude, resolution, and calibration types to the comment block (default=YES)
DAY *bday eday*	image day range

SS *ss1 ss2*	SSEC sensor source number range, from 1 to 999
SUBSET *bpos epos*	position range or SUBSET ALL
TIME *btime etime*	image time range

Selection clauses can restrict the search based on the image day, image start time and SSEC sensor source number. With the exception of AUX, you must specify these selection clauses as a range of values.

Note that in a **mcadir** call the dataset name does not contain the position field. The position, if known, is specified with the SUBSET selection clause. For example, if the application requires the first 10 images for a given dataset, the selection clause is SUBSET 1 10. If the application requires all the image directories in a dataset, the selection clause is SUBSET ALL.

If you include the selection clause AUX YES in the 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 latitude and longitude resolution) covered by the center element of the image, and the valid calibration types for the image. These values can subsequently be parsed out with the **mcpnav** and **mcpcal** functions, as shown in the code fragment on the next page.

Reading the directory block

Once the connection is opened with **mcadir**, the application makes repeated calls to **mcadrd** until all image object directory blocks are retrieved. The **mcadir** call must precede the call to **mcadrd**, as shown below.

```
character*4 calkeys(12) character*12 expkeys(12) character real lat real lon real latres
real lonres c --- set selection conditions selects(1) = 'SS 72 72' selects(2) = 'DAY 97001
97001' selects(3) = 'TIME 10:00:00 10:00:00' selects(4) = 'SUBSET ALL' selects(5) = 'AUX
YES' nselect = 5 c --- dataset name dataset = 'RT/GOES-9' c --- turn error reporting on
error_flag = 1 c --- open a connection for the specified dataset status = mcadir(dataset,
nselect, selects, error_flag) if(status .lt.0 )return 100 continue c --- read an image
directory block meeting the selection conditions readstat =
mcadrd(directory,comment_cards) c --- read failed if( readstat.lt.0 ) then call
edest('Failed during directory read of '//dataset,0) return c --- found one else if(
readstat.eq.0 ) then c --- process the data ... c --- get the list of cal types for band 4
band = 4 ok = mcpcal (comment_buffer, ncards, band, calkeys, expkeys, nkeys) c --- get the
navigation information at the center of the image ok = mcpnav (comment_buffer, ncards,
lat, lon, latres, lonres) goto 100 endif
```

The directory block contains a list of ancillary information about the image. The entries in the directory block are described in the table below.

IMPORTANT: When using **mcadir** to read the dataset, a 65 word block is returned, which includes the absolute position number. When using **mcaget**, the directory is 64 words long, including only words 1-64 in the the table below.

Entry	**Description**
0	absolute position of the image object in the ADDE dataset
1	relative position of the image object in the ADDE dataset
2	version number; currently=4
3	SSEC sensor source number; see the Appendices
4	nominal year and Julian day of the image, *ccyyddd*
5	nominal time of the image, *hhmmss*
6	upper-left image line coordinate
7	upper-left image element coordinate
8	reserved
9	number of lines in the image
10	number of data points per line
11	number of bytes per data point

12	line resolution
13	element resolution
14	number of spectral bands
15	length of the line prefix
16	SSEC project number used when creating the file
17	year and Julian day the file was created, *ccyyddd*
18	time the file was created, *hhmmss*
19	spectral band map, bands 1-32
20	spectral band map, bands 33-64
21 - 24	reserved for sensor-specific data
25 - 32	memo field; 32 ASCII characters
33	reserved
34	byte offset to the start of the data block
35	byte offset to the start of the navigation block
36	validity code
37 - 44	PDL (Program Data Load); used for pre-GOES-8 satellites
45	source of band 8; used for GOES AA processing
46	actual image start year and Julian day, *ccyyddd*
47	actual image start time, *hhmmss*; in milliseconds for POES data
48	actual image start scan
49	length of the prefix documentation
50	length of the prefix calibration
51	length of the prefix band list
52	source type; satellite specific (ASCII)
53	calibration type; satellite specific (ASCII)
54 - 56	reserved
57	original source type
58	units
59	scaling
60	byte offset to the supplemental block
61	number of entries in the supplemental block
62	reserved
63	byte offset to the start of the calibration block

64	number of comment cards

For more information about the directory block, see the *AREAnnnn* data structure in *Chapter 6, Format of the Data Files*.

Opening a connection to read an image object

In ADDE, a client can request an entire image object, including the directory, data, line prefix, navigation, calibration and comment blocks, from a server. For example, the McIDAS-X IMGDISP and IMGCOPY commands typically request entire image objects.

To open a connection to read an image object's data block, an application must perform these two steps:

1. Define the selection conditions for the desired image sector in the dataset.
2. Send a request to an image server.

Each is described below.

Defining the selection conditions

Applications use selection clauses to specify the spatial, temporal and spectral limits of the transaction with the server. This eliminates the need for the application to scan the dataset for a particular image object and also limits the size of the transmission to only that needed. The number and format of selection clauses are strictly regulated. Below is a list of the valid selection clause formats for the **mcaget** interface, which passes the request for an image sector from the client to the server. Additional information for each selection clause follows.

Selection clause	**Description**
AUX YES or AUX NO	additional information requested from server
BAND *band*	spectral band, if the image has multiple bands
CAL QTIR	quick calibration switch for POES images
DAY *bday*	image Julian day (no default)
DOC YES or DOC NO	includes the documentation from the line prefix (default=NO)
LOCATE *cor ycor xcor*	sets the coordinate type and the coordinate positions relative to the coordinate type (default=AU 0 0)
MAG *lmag emag*	line and element magnification factor (default=1 1)
POS *pos*	absolute position in the dataset
SIZE *lines elems*	number of image lines and data elements (default=480 x 640)
SU *name*	stretching table name (default=no stretch)
TIME *btime etime*	image time range (no default)

AUX YES
-- Data server: use this clause to insert the unit and scale factor of the image data into entries 58 and 59 of the directory block.
-- Directory server: use this clause to get center point and calibration information from the server as additional comment cards.

BAND -- Use this clause to identify a spectral band of image data. Specify BAND ALL to return all spectral bands in the source image.

CAL (obsolete) -- Use this clause only when the source data type is TIRO. Use CAL QTIR for quick calibration.

DAY -- Use this clause to specify the day of the source. Using DAY implies that the image object position (POS) is not specified in the connection request. If POS is specified, the DAY clause is ignored.

DOC YES -- Use this clause to include the line prefix's documentation when reading an image object.

LOCATE -- Use this clause to position the request spatially. LOCATE specifies a reference point from which sector bounds are determined. To specify the reference point of the image sector, use one of three coordinate systems (I=satellite, A=array or file, E=earth) with one of two standard offsets (U=upper-left, C=center); for example, EC=earth center, IU=satellite upper-left. Following the reference point are two values (*ycor* and *xcor*) that identify the absolute position of the reference point in the chosen coordinate system. For earth coordinates, the values are latitude and longitude; for satellite coordinates, the values are line and element; for array coordinates, the values are array row and column. If you don't specify LOCATE, the default is a satellite upper-left (IU) reference point of the first scan line and pixel of the source image object.

MAG -- Use this clause to specify the resolution magnification factor for the line and element dimensions of the image object. Enter a negative integer for a blowdown; enter a positive integer for a blowup. For example, a magnification factor of -4 will use every fourth data point on the line and every fourth line in the image. A magnification factor of 16 for both line and element dimensions will duplicate each data point in the image 256 times (16 x 16). The **mcaget** function performs all line and element duplication. The application must reduce the size of the data request to allow for a blowup factor. To use a magnification factor other than one, modify the input for SIZE accordingly. For example, to get an image that is 500 x 1000 with a MAG value of 2 2, you must specify SIZE 250 500. You cannot store images with non-integer line/element resolutions in the McIDAS-X image object data structure.

POS -- Use this clause to specify the position of the image in the dataset. If POS isn't specified, the server uses the most current (time-relative) image. Specify POS as either absolute or time relative. Numbers less than one imply time-relative position, with zero as the most current image.

SIZE -- Use this clause to specify the image line and data element limits of the transaction. Specify SIZE ALL to read the entire source image object.

SU -- Use this clause to specify the name of an image data stretching table. These tables are generated with the McIDAS-X SU command.

TIME -- Use this clause to specify a range of image start times identifying specific images in a dataset. Using this clause implies that the image object position (POS) is not part of the connection request. If a POS clause does exist, the TIME clause is ignored.

Server-specific selection clause	**Description**
ID	NEXRAD station ID (NEXRAD server)
TIME *btime etime* C	**C**overage; accesses data for specified time range (realtime POES server)
WL	wavelength (AIRS server)
WN	wavenumber (AIRS server)

Sending a request to an image server

Once the selection conditions for the image sector are defined, the **mcaget** function passes the request for the image sector from the client to the server. The return status shows if the connection is open and if the request can be completed.

The **mcaget** function lets the application specify the units and format of the data points. Since these parameters are necessary to any data transaction, they are specified as separate parameters to the **mcaget** function and are not entered as selection clauses. Units may be any unit identifier valid for the image object type. A list of valid unit identifiers is available to an application through the **mcadir** function by specifying the AUX YES selection clause. Use the format parameter to specify the bytes-per-data point in the return array. The valid formats are I1 (1 byte per data point), I2 (2 bytes per data point) and I4 (4 bytes per data point).

If you request that 2-byte data be returned as a 1-byte representation, without going through a calibration process that converts the 2-byte data to 1-byte, the server will truncate the least significant byte.

Because **mcaget** requires many selection conditions to request image data, the **mcasort** function can be called to translate command line keyword parameters into equivalent **mcaget** selection clauses. Any application-level program may call **mcasort** to retrieve these keywords and return them as **mcaget** selection clauses. The table below lists the keywords for accessing image objects and their equivalent selection clauses.

Command line keyword	**Equivalent mcaget selection clause**	**Remarks**
none	AUX YES	always set by mcasort
BAND=*band*	BAND *band*	only one spectral band in the clause
DAY=*bday*	DAY *bday*	
LATLON=*lat lon*	LOCATE *Eloc lat lon*	loc defaults to center (EC) if keyword PLACE is not specified
LINELE=*line ele sys*	LOCATE *Iloc line ele*	loc defaults to center (IC) if keyword PLACE

		is not specified
MAG=*lmag emag*	MAG *lmag emag*	
PLACE=*loc*	none	sets loc for the LATLON, LINELE, and STATION keywords
RTIME=*bmin emin*	TIME *btime etime*	keyword RTIME overrides keyword TIME
STATION=*stn*	LOCATE *Eloc lat lon*	loc defaults to center (EC) if keyword PLACE is not specified
TIME=*btime etime*	TIME *btime etime*	
WL=*bwl ewl*	WL *bwl ewl*	
WN=*bwn ewn*	WN *bwn ewn*	

Reading the image data

If an image sector in the dataset satisfies the client request, a connection is established between the server and the client and the transaction proceeds. Because applications manipulating image data must sometimes sample data from different sources simultaneously, ADDE allows an application to receive data from multiple datasets in any order required for that application. When a request for data is initiated with **mcaget**, a handle identifying the request is returned. This handle is subsequently used to retrieve each line of data for the request. If your application requires accessing data from multiple requests simultaneously, you only have to manage a handle for each request.

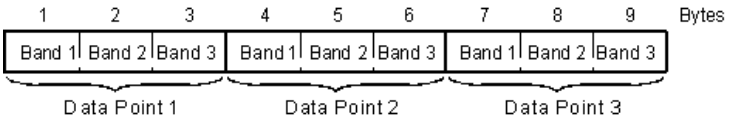
This section describes the functions you should use to get the requested image data from the server and to read the image object's line prefix, navigation, calibration and comment blocks.

Getting the requested image data

The **mcalin** function reads the requested image data from the server one line at a time. When all the data is read, the connection is closed. Then the **mcafree** function readies the environment for the next request by freeing the image handle and the memory allocated to store the image data from the previous request.

If you request multispectral data, each data point in the line will contain the measurements for the bands arranged consecutively. For example, Figure 5-1 shows the arrangement of data points for an image line of 1-byte data containing three spectral bands.

Figure 5-1. *An image line may contain multispectral data stored in bands arranged consecutively.*



For more information about manipulating data at the byte level, see the section titled *Conversion utilities* in *Chapter 4, McIDAS-X Utilities*.

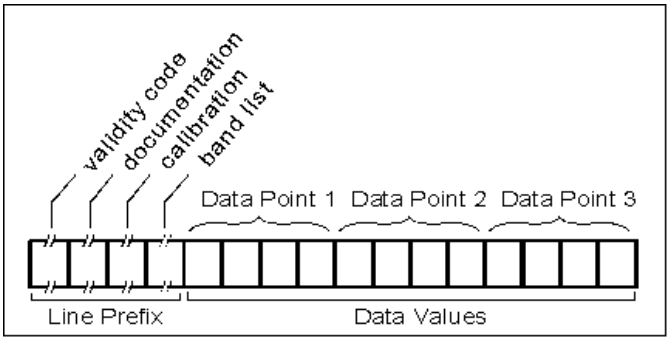
Below is a code fragment showing the steps for reading an image object data block. Note that the **mcaget** call must precede a call to **mcalin**.

```
c --- get selection conditions nselect = 1 selects(nselect) = 'DAY 96300' nselect =
nselect + 1 selects(nselect) = 'LOCATE IC 1000 1000' nselect = nselect + 1
selects(nselect) = 'TIME 19:00 19:15' nselect = nselect + 1 selects(nselect) = 'SIZE 200
300' c --- set the format of the returned data buffer format = 'I4' c --- set the units of
the returned data unit = 'TEMP' c --- specify the number of bytes available in the
data_buffer c --- (note that data_buffer is usually an integer array with c --- four bytes
per array element) max_byte = 300 c --- open a connection status = mcaget(dataset,
nselect, selects, 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 --- got a line of data else if(
status.eq.0 ) then c --- process the data ... goto 100 endif c --- Free the handle status
= mcafree( handle ) ...
```

Reading the line prefix block

If processing the image requires the line prefix, the application must get the prefix using the **mcapfx** function. The line prefix is the set of information that may precede the data on an image line; its maximum size is 1000 bytes. As shown in Figure 5-2, an image line consists of an optional line prefix and the actual data values.

Figure 5-2. *Each image line contains an optional line prefix and data values*.



The line prefix is divided into four regions:

- The *validity code* verifies the existence of the data portion of the image line. It is a constant value within each image and is stored in entry 36 of the image directory block. Comparing the value in the image directory with the value of the validity code determines if data exists for an image line.
- The *documentation* region holds the documentation specific to each satellite. Entry 49 of the image directory block defines the length of the prefix documentation.
- The *calibration* region holds the calibration coefficients for the data and is needed when coefficients vary between image lines. Entry 50 of the image directory block defines the length of the prefix calibration.
- The *band list* contains an ordered list of the spectral bands comprising the data portion of the image line. Entry 51 of the image directory block defines the length of the prefix band list.

The line prefix is the same length for each line in an image. The sensor source determines the size and content of the line prefix and which of its four regions are present.

As shown in the sample code below, the call to **mcapfx** must occur immediately after the call to **mcalin** so that the prefix and data are for the same image line.

```
c --- set up the ADDE transaction ... 100 continue c --- read the data block status =
mcalin(handle, data_buffer) if( status.lt.0 ) then call edest('Data Read failed',0) return
c --- got a line of data else if( status.eq.0 ) then c --- read the line prefix pfxstatus
= mcapfx(handle, prefix_buffer) if( pfxstatus.lt.0 ) then call edest('Prefix Read
failed',0) goto 100 endif c --- process the data ... goto 100 endif ...
```

Reading the navigation block

The **mcanav** function reads an image object's navigation block. 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. See the code segment below. To eliminate the risk of buffer overflow, dimension the **`nav_buffer`** to 4096 bytes.

```
c --- open a connection status = mcaget(dataset, nselect, selects, unit, format, &
max_byte, msg_flag, directory, HANDLE) if( status.lt.0 ) return ... c --- read the
navigation block status = mcanav(HANDLE, nav_buffer) if( status.lt.0 ) then call
edest('Navigation Block Read failed',0) return endif ...
```

Reading the calibration block

The **mcacal** function reads the calibration block of an image object. The call to **mcacal** can occur any time after the connection is opened by the **mcaget** call. The handle returned by **mcaget** is passed to **mcacal**, which returns the associated calibration block. To eliminate the risk of buffer overflow, dimension the **`cal_buffer`** to 40,000 bytes.

```
c --- open a connection status = mcaget(dataset, nselect, selects, unit, format, &
max_byte, msg_flag, directory, HANDLE) if( status.lt.0 ) return ... c --- read the
calibration block status = mcacal(HANDLE, cal_buffer) if( status.lt.0 ) then call
edest('Calibration Block Read failed',0) return endif ...
```

Reading the comment block

To read the comment block, use **mcacrd** and the handle returned by **mcaget**. The **mcacrd** function returns the entire comment block to the application. The call to **mcacrd** can occur only after the calls to **mcalin** are done.

Below is an example using **mcaget**, **mcalin** and **mcacrd** to read a comment block. To eliminate the risk of buffer overflow, dimension the **comment_buffer** to 40,000 bytes.

```
c --- open a connection status = mcaget(dataset, nselect, selects, 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 c ---
process the data ... 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 ...
```

Writing image objects to a dataset

To write an image object to an image dataset, the application must identify a dataset and position, and open a connection with the server. Use the API functions below to write image objects to a dataset.

Function	Description
mcaput	opens a connection to write the directory, navigation and calibration blocks of an image object
mcaout	writes the line prefix and data portions of an image line
mcacou	writes the comment block of an image object

Opening a connection to write an image object

The request to open a connection is performed by the **mcaput** function, which requires the following:

- A valid dataset name
- An image object position
- Directory, navigation, and calibration blocks

Because **mcaput** does not return an object handle, only one image object is written at a time.

The only valid selection clause for writing image objects 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.

Writing the data block

Once the connection is open, the server expects to transfer the number of bytes 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 **mcaout**, which is called as many times as necessary to transfer the bytes. The **mcaout** function has only one argument, which is an array of data points to write to the data block. The call to **mcaput** must occur prior to **mcaout**.

Writing the comment block

The comment block is written after the last byte of the data block is transferred. The number of comment entries is defined in word 64 of the directory block. If this entry is nonzero, the specified 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 sample code fragment below writes image objects to a dataset. For another **mcaput** code example, see the function **imgcopy.f**

```
c --- initialize the directory block call zeros(directory_block, 64) c --- create a
comment card call getday( day ) call gettim( time ) cday = cfu( day ) ctime = cfu( time )
comment = cday(1:5)//' '//ctime(1:6)//' This is a comment ' ncard = ( len_trim(comment) /
80 ) + 1 c --- 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 points/line directory_block(11)= num_bytes
! number of bytes/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 movcw(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 c --- initialize the navigation block
call zeros(nav_block, nav_size) c --- fill the navigation block entries c Note:
"navigation_params" is an array of navigation parameters c that describes the geo-location
of the elements of the c image object. do 10 i = 1,nav_size nav_block(i) =
navigation_params(i) 10 continue c --- initialize the calibration block call
zeros(cal_block, cal_size) c --- fill the calibration block entries c Note:
"calibration_parms" is an array of calibration parameters c that transforms the data
elements to physical units. do 20 i = 1,cal_size cal_block(i) = calibration_params(i) 20
continue c --- fill the selection array nselect = 1 selects(nselect) = 'POS
'//cfu(position) c --- open a connection to write the image object if( mcaput( image,
nselect, selects, 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 "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).
"data_buffer" is a one-dimension array c sized to (num_elems) c Note: this assumes a 4
byte to 1 byte compression of the data. call pack( num_elems, data_array(line, 1),
data_buffer) c --- write a line of data to the image object if( mcaout( data_buffer ).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
```

Deleting image objects

You can delete image objects from a dataset using **mcadel**, as long as you have write permission. The **mcadel** function has one valid selection clause, SUBSET, which you must specify during the connection phase of the transaction or the server request will fail. The code fragment below deletes the image objects at locations **pos1** through **pos2** from the dataset.

```
c --- construct selection clause nselect =1 selects(nselect) = 'SUBSET '//cfu( pos1 )//'
'//cfu( pos2 ) call bsquez( selects(nselect) ) c --- delete image object if( mcadel(
dataset, nselect, selects, msgflg ).ne.0 ) then call edest(' Failed to delete image
objects ',0) else call sdest(' Image objects deleted',0) endif
```

Grid data

McIDAS-X grids are typically composed of atmospheric and oceanographic data, which are produced by numerical models or derived from observational data using an objective analysis scheme. Grids may contain any data that can be represented in a two-dimensional matrix.

Because grid data and image data can both be represented in two-dimensional matrices, it's important to know how they're different. The attributes that distinguish them are listed in the table below.

Attribute	Grid data	Image data
data volume	low	high
data value resolution	high	low
geographic resolution between data points	low	high
representation to users	graphical contours	gray shading

This section describes:

- Attributes unique to grid data
- Blocks of information contained in a grid object
- API functions available for reading and writing grid data
- Selection conditions you can use when requesting grid data

Basic concepts

Grid data has five unique attributes: valid time, level, parameter, origin and navigation. Each is described below.

Valid time

Grid data is often used to store output from numerical models that simulate the atmosphere and ocean. Because these models predict what the atmosphere or ocean will look like at some time in the future, grids must contain two different time attributes when representing model forecasts:

- The primary time attribute is the time the model is initialized.
- The secondary time attribute is the time the field will represent.

For example, if a model run on 17 January at 00 UTC generates forecast fields valid 36 hours later, the primary time attribute is 17 January at 00 UTC and the secondary time attribute is 18 January at 12 UTC. If the grid does not represent a model forecast, no secondary time attribute is needed.

Level

Grids store information that represents data at some vertical location in the atmosphere. The level can be the constant height above the surface, such as 100 meters or 32000 feet; the constant pressure surface, such as 500 millibars; or some other meteorological surface, such as the level of the tropopause or isentropic surface.

Parameter

The grid parameter is the data type the grid represents. Parameters can be any field that can be stored as a number, such as temperature, wind speed or dew point. Grids must also contain the stored units used in the grid.

Origin

A grid's origin refers to the process that created the grid. For example, if a grid is created by a forecast model, the origin is the name of the model, such as ECMWF or GFS. If the grid is created from another objective analysis process, that name can appear as the origin.

Navigation

Navigation is the process of determining the latitude and longitude location of data points on a grid. This information is needed when colocating data points from a grid with another data type. For example, you can use this information to contour the gridded field on a satellite image displayed on the McIDAS-X Image Window.

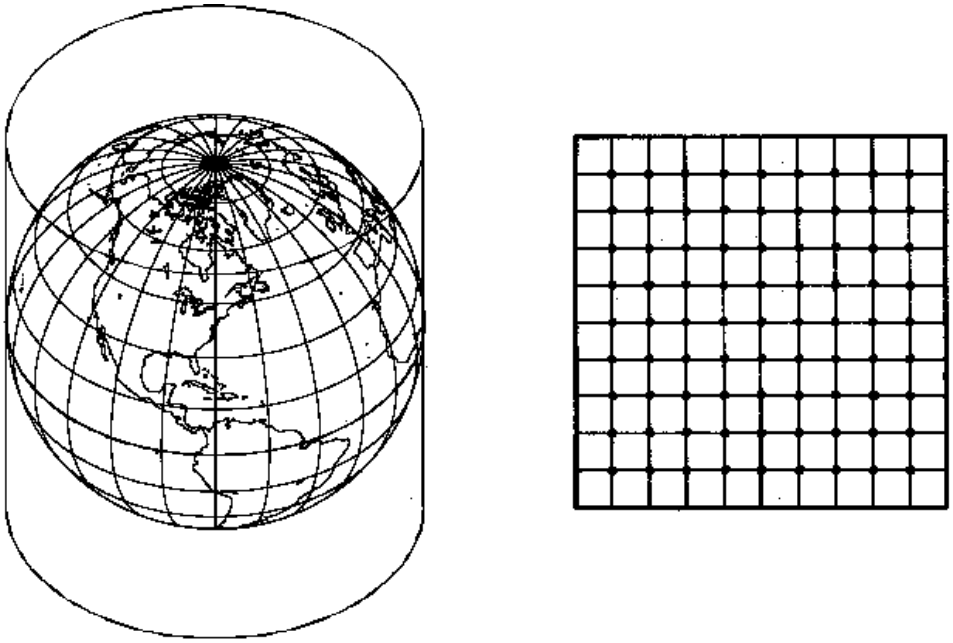
McIDAS-X currently recognizes these grid projection formats:

- Pseudo-Mercator
- Polar stereographic
- Lambert conformal secant
- Tangent cone
- Equidistant

The diagrams below show how a three-dimensional Earth is represented on a two-dimensional surface for the pseudo-Mercator, polar stereographic and Lambert conformal projection formats, and the orientation of the data points on these projections. A sixth grid projection format is also available, but since it has no navigation, the data points can't be converted to planetary coordinates.

For more information about grid projections and McIDAS-X navigation, see the section titled *McIDAS navigation* later in this chapter.

Figure 5-3. *Pseudo-Mercator projection.*



Figue 5-4. *Polar stereographic projection*.

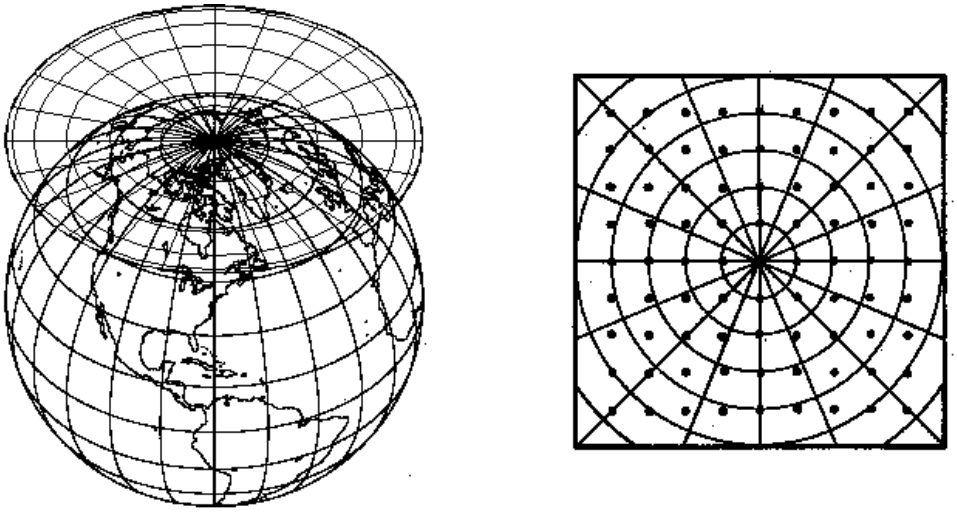
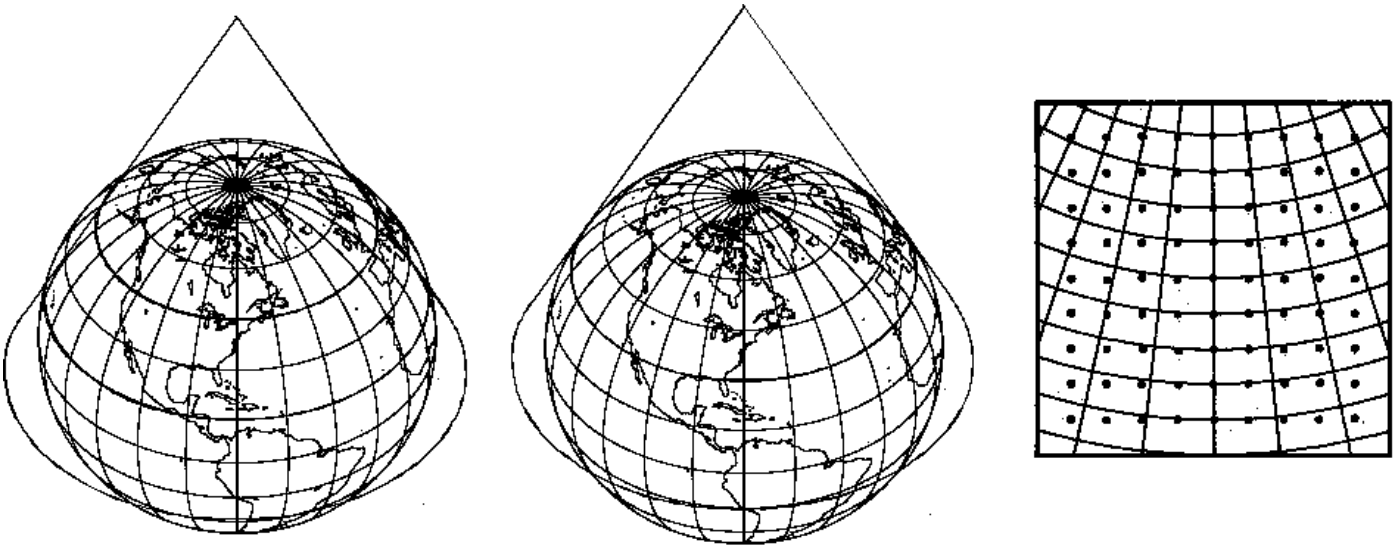


Figure 5-5. *Lambert conformal secant cone (left) and tangent cone (right) projections*



What is a grid object?

Gridded data is two-dimensional data representing an atmospheric or oceanic parameter along an evenly spaced matrix. For the matrix to be useful, ancillary information about the grid must also be known. This ancillary information, along with the gridded data, is collectively called a *grid object*. Grid objects in McIDAS-X contain two blocks of information.

- The *grid header* contains a list of ancillary information about the grid, such as the parameters and units of the data in the grid, the level in the atmosphere or ocean the data represents, the grid navigation information, and the time.
- The *data block* contains the matrix of gridded data values.

The API functions and procedures for reading and writing grid objects are described below.

Reading grid objects

Most applications for reading grid objects will do one of the following:

- Open a connection to read only the grid header of a grid object and then read the grid header
- Open a connection to read a grid object, including the directory and data blocks, and then read the grid data

The table below lists alphabetically the McIDAS-X library functions for performing these tasks.

Function	Description
mcgdrd	reads the grid header from a grid object
mcgget	opens a connection to read the data block of a grid object
mcgdir	opens a connection to read the grid header of a grid object
mcgfdrd	retrieves the grid file header
mcgridc	reads the grid object and returns the grid in row-major (C) format
mcgridf	reads the grid object and returns the grid in column-major (Fortran) format
m0gsort	gets the parameters from the command line and adds them to the selection array for future mcgget call

These functions are described below along with sample code, following an explanation of the selection conditions for requesting grid objects.

Defining selection conditions

Applications use selection clauses to restrict the information sent from the server to the client. For example, selection clauses can restrict the search based on grid time attributes, the parameter type and level, or the process that generated the grid. Below is a list of the valid grid selection clauses as sent to the server. Additional information for each follows.

Selection clause	**Description**
DAY *day1 .. dayn*	list of primary grid days
DRANGE *bday eday inc*	range of primary grid days with a day increment
FDAY *day*	forecast day
FHOUR *hour1 .. hourn*	list of forecast hours
FRANGE *bvt evt inc*	range of forecast hours with an hour increment
FTIME *time*	forecast time
GRIB *geographic parameter model level*	specific grib numbers to find
GRID *bgrid egrid*	specific range of grids in a grid file
LEV *lev1 .. levn*	list of data levels
NUM *numgrid*	number of grids to find
OUT *option*	header output format to return
PARM *p1 .. pn*	list of parameters
POS *offset*	relative position offset in a dataset
PRO *projection*	grid projection type
SRC *src1 .. srcn*	list of sources that generated the grid
TIME *time1 .. timen*	list of primary grid times
TRANGE *btime etime inc*	range of primary grid times with a time increment

DAY -- Use this clause to identify a list of primary days that the grids represent. For model forecast grids, these values will be the day the model was initialized. Otherwise, DAY is the Julian day the data represents.

DRANGE -- Use this clause to identify a range of primary days that the grid represents. Enter the beginning and ending day numbers and the increment between days in the range. The increment default is one day.

FDAY -- Use this clause to identify the secondary day attribute for requested forecast grids. For example, if you want only the forecast fields valid for day 1997017, specify FDAY=1997017.

FHOUR -- Use this clause to identify a list of secondary forecast hours that the grids represent. For example if you want only the 12-, 24- and 48-hour forecasts from a model run, specify FHOUR=12 24 48.

FRANGE -- Use this clause to identify a range of forecast hours that the grids represent. Enter the beginning and ending forecast hours and the increment between hours in the range. The increment default is one hour.

FTIME -- Use this clause to identify the secondary time attribute for the grids requested. Use this field with the FDAY clause to isolate grids that are valid at a particular time. The format for the arguments is *hhmmss*. For example, to request all grids valid at 12 UTC on day 96017, specify FDAY=96017 FTIME=120000.

GRIB -- Use this clause to select grids with the GRIB values specified for the geographic region, the parameter, the model, or the level. This is a useful way to find grids when other sort conditions provide insufficient information to differentiate between grids.

GRID -- Use this clause to access grids based on their position in specific grid files. Since this is an artifact of previous McIDAS-X API functions, avoid using this clause if a practical alternative exists.

LEV -- Use this clause to identify a list (not a range) of levels in the atmosphere or ocean that this data represents. This field is typically filled with height in millibars or words such as SFC, MSL or TRO. To retrieve data for several levels, enumerate them individually.

NUM -- Use this clause to specify the maximum number of grids returned from the server. The default is one grid. To receive all grids matching the selection conditions, use NUM=ALL.

OUT -- Use this clause to receive only the grid header. To get the entire grid header, specify ALL (default). To get only the grid file header, use FILE.

PARM -- Use this clause to specify a list of parameter types to retrieve from the server. To retrieve temperature and height fields, enter PARM=T Z.

POS -- Use this clause to specify a grid file in a dataset. This is a relative position based on the dataset description. For example, to request grid file 5010 from a dataset that contains grid files 5001 to 5100, specify POS 10.

PRO -- Use this clause to specify a projection type for a grid. The valid entries are LAMB, CONF and MERC.

SRC -- Use this clause to specify a list of grid source names to retrieve from the server. This is usually the name of the model or process that generated the grid, such as ETA, NGM or MDX.

TIME -- Use this clause to identify a list of primary times that the grids represent. For model forecast grids, these values will be the time of day that the model was initialized. The format for the arguments is *hhmmss*.

TRANGE -- Use this clause to identify a range of primary times that the grid represents. Enter the beginning and ending times and the increment between times in the range. The increment default is one hour. The format for the arguments is *hhmmss*.

You can use the **m0gsort** function with any application-level program to retrieve command line keyword parameters and translate them into equivalent selection clauses. The table below lists the keywords for accessing grid objects and their equivalent selection clauses. You can also set a flag in **m0gsort** to disable a request to contain multiple grid selection matches.

Command line keyword	**Equivalent selection clause**	**m0gsort restrictions**
DAY=*d1 .. dn*	DAY	cannot use with DRANGE
DRANGE=*bday eday inc*	DRANGE	cannot use with DAY
FDAY=*day*	FDAY	cannot use with FHOUR or FRANGE
FHOUR=*h1 .. hn*	FHOUR	cannot use with FDAY, FRANGE or FTIME
FRANGE=*bhr ehr inc*	FRANGE	cannot use with FDAY, FHOUR or FTIME
FTIME=*time*	FTIME	cannot use with FHOUR or FRANGE
GPRO=*g1 ..gn*	PRO	validate with projection: MERC, PS, LAMB, EQUI
GRIB=*geo param model level*	GRIB	
GRID=*bgrid egrid*	GRID	use LAST to get the last grid in the dataset and position; when specified, all other selection conditions are ignored
LEV=*l1 .. ln*	LEV	
PARAM=*p1 .. pn*	PARM	
SRC=*s1 .. sn*	SRC	
TIME=*t1 .. tn*	TIME	cannot use with TRANGE
TRANGE=*btim etim inc*	TRANGE	cannot use with TIME

Opening a connection to read the grid header

In ADDE, a client may request only grid headers from a server. This allows the client to sample information on a server without transferring large amounts of grid data that may not be necessary for an application. For example, the McIDAS-X GRDLIST command reads only grid headers. The ADDE interface to the grid directory is through **mcgdir**, which opens a connection based on a set of selection clauses for a given dataset name.

Reading the grid header

Once **mcgdir** opens the connection, the application makes repeated calls to **mcgfdrd** and **mcgdrd** until all grid and grid file headers are retrieved. The **mcgfdrd** function is called first to retrieve the grid file header, then **mcgdrd** is called until it can't find any more grids in the grid file. Then **mcgfdrd** is called again and the loop continues, as shown below.

```
character*32 selects(5) character*32 dataset integer grid_header(64) integer
file_header(64) integer nselects integer error_flag integer status c--- assign the dataset
name dataset = 'RTGRIDS/ALL' c--- assign the selection conditions to retrieve six grids
from c--- the dataset RTGRIDS/ALL that are from the ETA, NGM or c--- MRF model with a
primary day of either 96017 or 96019 selects(1) = 'DAY=96017 96019' selects(2) = 'SRC=ETA
NGM MRF' selects(3) = 'NUM=6' nselects = 3 c--- set an error flag to print a message if an
error occurs error_flag = 1 c--- open the connection to the server status =
mcgdir(dataset, nselect, selects, error_flag) if (status .lt. 0)then return endif c---
every time statement 100 is reached, try to read a new grid c--- file header 100 continue
hdread = mcgfdrd(file_header) c--- if you have successfully read the grid file header if
(hdread .eq. 0)then c--- every time statement 200 is reached, try to read a new grid
header 200 continue gread = mcgdrd(grid_header) c--- if you have successfully read the
grid header if (gread .eq. 0)then {process grid header here} c--- see if there are any
more grid headers to read goto 200 c--- if you have read the last grid from this file, go
see if there c--- are any more grid files to read from elseif (gread .eq. 1)then goto 100
c--- if there was a problem reading the grid header elseif (gread .lt. 0)then return endif
elseif (hdread .lt. 0)then call sdest(`Unable to read grid file header',0) endif c--- if
you make it to here, hread has returned the value 1, c--- which means the server has
finished sending data
```

The grid header contains a list of ancillary information about the grid. The entries in the grid header are described in the table below.

Header Word	**Description**
1	total size; rows * columns (not to exceed the value of MAXGRIDPT in gridparm.inc)
2	number of rows
3	number of columns
4	Julian date of the data, *ccyyddd*
5	time of the data, *hhmmss*
6	forecast time for the grid, if applicable
7	name of the gridded variable, four character ASCII
8	scale of the gridded variable, specified as a power of 10
9	units of the gridded variable, four character ASCII
10	value of the vertical level 1013 = 'MSL' 999 = ' ' 0 = 'TRO' 1001 = 'SFC' (Otherwise, it is displayed as entered.)
11	scale of the vertical level
12	unit of the vertical level
13	gridded variable type: 1 = time difference 2 = time average 4 = level difference 8 = level average (or any sum of 1, 2, 4 and 8)
14	used if the grid parameter is a time difference or time average, *hhmmss*
15	used if the grid parameter is a level difference or level average; values are the same as Word 9

16 - 32	reserved
33	grid origin; identifies the type of program that generated the grid data
34	grid projection type: 1 = pseudo-Mercator 2 = polar stereographic or Lambert conformal 3 = equidistant 4 = pseudo-Mercator (more general) 5 = no navigation 6 = tangent cone
35 - 40	varies, depending on the grid type; see the *GRIDnnnn* data file in Chapter 6 for more information
41 - 52	reserved; filled only if the grid was created by the McIDAS-XCD GRIB decoder
49	geographic grib number
50	parameter grib number
51	model grib number
52	level grib number
53 - 64	grid description

Opening a connection to read the grid object

In ADDE, a client can request an entire grid object, including the grid header and data block from the server. For example, the McIDAS-X GRDDISP and GRDCOPY commands request entire grid objects.

The **mcgget** function opens a connection to read the data block of a grid object. It passes an application's selection conditions for requesting grid objects from the client to the server. The return status from **mcgget** indicates if the application's request can be fulfilled.

The **mcgget** function also allows the application to separately specify the units and format of the data returned. These are not part of the selection conditions because they are required. Units may be any unit identifier valid for the data type being retrieved. The format parameter can be either I4 for integer or R4 for real number.

Reading the grid data

If a grid in the dataset satisfies the client request, a connection is established between the server and the client and the transaction proceeds. The requested grid objects are read with either the **mcgridf** or **mcgridc** function; **mcgridf** reads the column-major (Fortran) format, while **mcgridc** reads the row-major (C) format.

Unlike image objects, which require multiple calls to **mcalin** to get an entire object, **mcgridf** and **mcgridc** return an entire grid object with each call. Below is a sample code fragment demonstrating a **mcgget**/**mcgridf** calling pair. Note that the **mcgget** call must occur before **mcgridf**.

```
include 'gridparm.inc' character*24 dataset integer grid(maxgridpt) integer header(64)
character*24 selects(8) c--- set up a request to get the 24-hour 500 mb temperature c---
field forecast grids from the ETA and NGM model runs at c--- 12 UTC on day 96017 dataset =
'RTGRIDS/ALL' selects(1) = 'DAY=96017' selects(2) = 'TIME=12' selects(3) = 'SRC=ETA NGM'
selects(4) = 'PAR=T' selects(5) = 'LEV=500' selects(6) = 'FHOUR=24' selects(7) = 'NUM=2'
nselects = 7 c--- send the request to the server to return the data as c--- scaled
integers in Celsius status = mcgget(dataset, nselect, selects, 'C ', 'I4', & maxpts*4, 1,
numgrids, totbyts) c--- if there was an error finding the data requested if (status .ne.
0)then return endif c--- if you have made it to here, numgrids contains the number c--- of
grid objects the server wants to return to you, so call c--- mcgridf to retrieve the grid
objects do 100 I = 1, numgrids status = mcgridf (grid, header) if (status .lt. 0)then call
sdest('error retrieving grid',0) goto 100 endif c--- do some processing.. 100 continue
```

Writing grid objects to a dataset

Writing a grid object to a grid dataset has two restrictions:

- It must be performed on your local workstation.
- The application must specify a position within the dataset to write the grid object to.

Use the API functions below to write grid objects to a dataset.

Function	Description
mcgoutc	writes a grid object stored in row-major format to a file
mcgoutf	writes a grid object stored in column-major format to a file
mcgput	opens a connection to write a grid object

Opening a connection to write a grid object

The request to open a connection for writing a grid object is performed by the function **mcgput**, which requires the following:

- Valid dataset name and position number, which is entered as one of the selection conditions in the **mcgput** call
- Grid header and a data block

When writing a grid object, the application may create and initialize a file in the destination dataset using the selection conditions below.

Selection clause	Description
DEL=YES or DEL=NO	deletes the destination dataset file before recreating it to write the grid object (default=NO)
GRID=*num*	grid number in a dataset location to write the grid object to; the previous grid stored in this location is overwritten
LABEL=	label to attach to the dataset when the file is created
MAX=*num*	maximum number of grid objects that can be stored in the newly created file
NUM=	number of grid objects to write to the dataset
POS=*pos*	position number in the dataset to write the grid object to

Writing the data block

Once the connection is open, the server expects to transfer the number of bytes specified by the entries in the grid header. Transferring too few or too many bytes will result in a error. The **mcgoutf** and **mcgoutc** functions send the grid objects to the server. **mcgoutf** assumes the data block is stored in column-major order; **mcgoutc** assumes the data block is stored in row-major order. You must call **mcgput** before calling **mcgoutf** or **mcgoutc**, as shown in the sample code fragment below.

```
integer grid1(MAXGRIDPT), grid2(MAXGRIDPT) integer grid_h1(64) grid_h2(64) character*48
selects(10) character*48 dataset c--- write to the dataset LOCAL/GRIDS dataset =
'LOCAL/GRIDS' c--- set up the selection conditions. We will write 2 grid object c--- to
dataset position 3 on the server. selects(1) = 'POS=3' selects(2) = 'NUM=2' selects(3) =
'MAX=100' nselects = 3 c--- initialize the grid headers. Not all fields are shown c--- in
this example. The first grid object will contain 300 c--- mb height fields, the second
will contain 250 mb height c--- fields c--- assign the size of the first grid object
grid_h1(1) = 500 grid_h1(2) = 20 grid_h1(3) = 25 c--- assign 300 mb level grid_h1(10) =
300 : : c--- copy the first grid header to the second call movw(64, grid_h1, grid_h2) c---
change the level of the second grid header to 250 mb grid_h2(10) = 250 c--- assign the
total number of bytes that will be transmitted. c--- this number will be the total number
of data points in c--- each of the data blocks, plus the size of the grid header c--- of
each of the objects. We multiply the total by 4 because c--- the storage format of the
grid object is whole words and c--- there are 4 bytes per word. ngrids=2 total_bytes = 8+
(grid_h1(1)+grid_h2(1)+(ngrids*64)+ngrids)*4 c--- send the request to write data to the
server ok = mcgput(dataset, nselects, selects, 1, total_bytes) if (ok .lt. 0)then return
endif c--- write the first grid object ok = mcgoutf(grid1, grid_h1) if (ok .lt. 0)then
return endif c--- write the second grid object ok = mcgoutf(grid2, grid_h2) if (ok .lt.
0)then return endif
```

Calculating the number of bytes to transfer

In the sample code above, you will notice that the last parameter in the **mcgput** function is the total number of bytes to transfer from the client to the server. You must provide this number using the equation below.

total bytes = 8 + 4 * ((*number of grids* * 64) + (*number of data points*) + (*number of grids*))

For example, to transfer two grids to a server where the first grid has 200 data points and the second grid has 300 data points, the total number of bytes to transfer is 2528, as shown below.

8 + 4 * ((2 * 64) + (200 + 300) + 2) = 2528

Point data

McIDAS-X point data is typically composed of atmospheric and oceanographic data occurring at irregularly spaced locations on the Earth or vertically within the atmosphere or ocean. This type of data storage is most often used with station observations such as synoptic, RAOB or ship reports.

This section describes:

- Attributes unique to point data
- Blocks of information contained in a point object
- API functions available for reading both the point data and the file header
- Selection conditions you can use when requesting point data

Basic concepts

Point data has two unique attributes:

- Its structure
- Its ability to contain mixed data types

Each of these attributes is described below along with the limitations imposed on point data.

Point structure

Conceptually, you can think of point data as a spreadsheet with each cell containing a predefined number of data values. Each cell contains data for a specific location at a given instant in time. For example, one cell may contain all the mandatory level RAOB data from Green Bay, Wisconsin, at 12 UTC on 17 January 1996.

Mixed data types

Unlike grids, point objects may contain a combination of data types and units for different elements within a cell. For example, in a point cell of surface hourly information, you can store the following:

- Temperature and dew point in degrees Celsius
- Mean sea level pressure in millibars
- Character string representations of the station ID, state or province, and country

Limitations

When accessing point data, the MD (Meteorological Data) file structure has four limitations, which are explained below. These limitations pertain only to the MD file structure and are not limitations of the ADDE point object subsystem. Point servers for other formats, netCDF for example, use configuration files to map the parameter names in the file to McIDAS-compatible names.

Cells

Each cell is limited to 400 elements. For most observational data, this isn't problematic. The exception occurs with observations containing data at several levels of the atmosphere. For example, you can't store all the information for an upper air observation reporting values for eight different parameters at 50 levels of the atmosphere in one cell. Although the cell can accommodate the 400 values (8 x 50), it won't have enough space for the time and geographic location of the observation, which are also provided.

Element names and units

Element names and units are limited to four characters, which can be restricting when designating parameter units, especially derived parameters.

Character string elements

Character string elements are limited to four characters, which can be limiting for any type of alphanumeric parameter, such as station ID or country. You can bypass this restriction by using several parameters strung together to represent strings. McIDAS-XCD uses this method when representing five-character IDs associated with ship reports.

Numeric values

Numeric values can be stored only as scaled integers.

What is a point object?

McIDAS-X point data typically represents data occurring at irregularly spaced locations on the Earth. For this data to be useful, ancillary information about the data must also be known. This ancillary information, combined with the actual point data values, is collectively called the *point object*.

Each point object in McIDAS-X contains five blocks of information.

- The *parameter block* contains a list of the parameter names in the point object returned by the server.
- The *unit block* contains a list of units for the parameters returned by the server.
- The *scale block* contains a list of scaling factors for the floating- point values returned by the server.
- The *form block* contains a list of the return forms for each of the parameters.
- The *data block* contains the actual data values returned by the server.

The API functions and the procedures for reading point objects are described below.

Reading point objects

Most applications for reading point objects will request one of the following:

- A specific list of parameters
- All parameters for a given dataset

The McIDAS-X command PTLIST is an example of an application that reads point objects. The table below lists alphabetically the McIDAS-X library functions for acquiring point data.

Function	Description
m0ptget	opens a connection to read a point object
m0ptparm	extracts the parameter information from the command line
m0ptrd	reads the point data block from the server
m0psort	gets the selection parameters from the command line and adds them to a selection array used by **m0ptget**

These functions are described below along with an explanation of the selection conditions for requesting point objects.

Defining selection conditions

Applications use selection clauses to restrict the information sent from the server to the client. You can tell the server to return only fields that fall within certain thresholds. These selection limitations may include a list of stations, a time range, or a level in the atmosphere. Below is a list of the valid point selection clauses. Additional information for each follows.

Selection clause	Description
MAX	maximum number of point objects to find (default=1)
POS	position number in a dataset to retrieve data from (default=ALL if day is specified; default=0 if day is not specified)
SELECT *s1 .. sn*	list of selection conditions

MAX -- Use this clause to specify the maximum number of point objects returned from the server. To receive all point data matching the selection conditions, use MAX=ALL.

POS -- Use this clause to identify a specific point file in a dataset. This is a relative position based on the dataset description. For example, to request point file 5010 from a dataset containing point files 5001 to 5100, specify POS=10. To read from all the files in a dataset, specify POS=ALL.

SELECT -- Use this clause to identify the selection conditions for limiting the objects returned to the client. The syntax of this clause varies, depending on the request. To include multiple conditions, enclose each clause in single quotes. For example, to limit the list of stations in a surface hourly observation to include only Madison and Milwaukee, you can use: 'ID KMSN, KMKE'. To limit a selection to include only those stations with a temperature between 0° and 32° Fahrenheit, include: 'T 0 TO 32 F'.

Since piecing the selection conditions together can be a difficult task, the **m0psort** function will build the appropriate SELECT string for you. Use **m0psort** with any application-level program to retrieve command line keyword parameters and translate them into equivalent selection clauses. For example, if you enter a point command with the following arguments:

SELECT='T[F] 40 50; ST WI, MI; TIME 12 13'

m0psort will return the following information, which can then be passed to the server via **m0ptget**.

'T 40 TO 50 F'
'ST WI,MI'
'TIME 12 TO 13'

Opening a connection to read the point object

Once the selection conditions are made, the **m0ptget** function opens a connection to read point data from the server. The calling sequence for **m0ptget** allows the client to access the data in one of two modes.

In the first mode, the server returns all the parameters for a given data type matching the selection conditions, such as all the decoded information from a METAR report for a given station. If you use the PTLIST command without specifying the PARAM keyword in this mode, the client may not know the valid parameters and units for the data type until a successful return from **m0ptget**.

In the second mode, the client knows the list of parameters to request and the units they can be returned in, such as temperature in Celsius and wind speed in knots.

The API for **m0ptget** contains the field *asknparm* for input and output. If this field is zero, the first mode of data acquisition is assumed and the client retrieves all data associated with this data type. If a list of parameters is specified from the client, the second mode of data acquisition occurs and *asknparm* will contain the number of parameters to return.

When accessing point data in the second mode, you must supply **m0ptget** with a list of parameters and units to retrieve. If you adhere to the McIDAS-X command line syntax PARAM=*parameter*[*units*], you can use the **m0ptparm** function to build the parameter and unit list.

Upon successful return from **m0ptget**, all elements of the point object are returned by the server, except the actual data block. If the client requests all parameters, the parameter and unit block are returned. The data form is also returned, indicating the type of data each parameter is stored in. For character strings, the form is C*#;* for integer values, it is I*#;* and for floating point values, it is F#, where # is the number of bytes for this parameter. The scaling factor is also returned for floating point numbers.

If the request can be fulfilled by the server, the parameter, unit, scale and form blocks are filled accordingly, regardless of which mode the client uses for **m0ptget**. The example in the next section demonstrates the second mode of data retrieval.

Reading the point data

If the point request for a given dataset can be fulfilled, a connection is established between the server and the client and the transaction proceeds. The **m0ptrd** function reads the point data. It is called continuously until all the data is read. Because each call to **m0ptrd** may yield character strings, integers and floating point numbers, the McIDAS-X library contains a group of functions for extracting these mixed data type values. You can call the **m0ptbufinit** function after each **m0ptget** call to initialize the application environment to more easily extract data from the buffer.

The example below demonstrates point data acquisition, using the **m0ptbufc**, **m0ptbuff** and **m0ptbufi** functions to extract characters, floating point, and integer values from the buffer filled by **m0ptrd**.

```
subroutine main0 implicit none include 'ptparm.inc' integer MAXBYTE ! max bytes the buffer
may contain parameter (MAXBYTE = MAXNUMPARM * 4) character*24 dataset ! adde dataset name
character*80 select(2) ! list of selection conditions integer nselect ! number of
selection conditions integer nparams ! number of parameters to return character*1 quote !
single quote character*4 params(MAXNUMPARM) ! list of parameters character*4
units(MAXNUMPARM) ! list of units character*4 form(MAXNUMPARM) ! list of return forms
integer scales(MAXNUMPARM) ! list of scales integer buffer(MAXNUMPARM) ! data buffer
integer ok ! function return value double precision temperature ! temperature extracted
from the ! buffer double precision cloudheight ! cloud height extracted from the ! buffer
character*4 id ! station id extracted from the ! buffer character*12 ct ! temperature
character*12 czcl ! cloud height character*12 chms ! ob time integer obtime ! observation
time extracted ! from the buffer c--- external library routines integer m0ptbufinit
integer m0ptbufc integer m0ptbufi integer m0ptbuff integer m0ptget integer m0ptrd data
units/MAXNUMPARM * ' '/ quote = char (39) c--- assign the dataset dataset =
'RTPTSRC/SFCHOURLY' c--- set up the selection conditions to retrieve the station id, c---
the temperature in Celsius, the observation time and the c--- first non-ceiling cloud
height for five SFCHOURLY stations c--- in Wisconsin with a temperature between 60 and 70
Fahrenheit c--- for day 1996274 params(1) = 'ID' params(2) = 'T' units(2) = 'C' params(3)
= 'HMS' params(4) = 'ZCL1' nparams = 4 select(1) = 'MAX=5' c--- the assignment below is
```

```
usually more easily performed by the c--- function m0psort. it is demonstrated in this
example in a fully c--- expanded manner to show the most basic access method. select(2) =
'SELECT=' & //quote//'ST WI'//quote//' ' & //quote//'T 60 TO 70 F'//quote//' ' &
//quote//'DAY 1999274'//quote c--- select(2) will contain the following. note that the
single c--- quote characters are an important element of the string: c--- SELECT='ST WI'
'T 60 TO 70 F' 'DAY 1999274' nselect = 2 c--- make the request to the server ok = m0ptget
(dataset, nselect, select, nparams, params, & units, form, scales, MAXBYTE, 1) if (ok .lt.
0)then goto 999 endif c--- upon a successful return from m0ptget, the arrays will contain
c--- the following: c--- location params units form scales c--- 1 ID CHAR C4 0 c--- 2 T C
F4 2 c--- 3 HMS HMS I4 0 c--- 4 ZCL1 FT F4 -2 c--- initialize the system so m0ptbufc/f/i
can be used to c--- extract the buffer values ok = m0ptbufinit (nparams, form, scales) if
(ok .lt. 0)then goto 999 endif c--- continuously call m0ptrd until it indicates there are
no more c--- point data blocks to return. call sdest('ID T[C] Time ZCL1[FT]',0) 100
continue ok = m0ptrd (buffer) if (ok .lt. 0)then goto 999 endif c--- if data was found,
process the data if (ok .eq. 0)then ct = ' ' chms = ' ' czcl = ' ' c--- extract the
station id ok = m0ptbufc (1, buffer, id) c--- extract the temperature ok = m0ptbuff (2,
buffer, temperature) if (ok .eq. 0)then write(ct,FMT='(f9.1)')temperature endif c---
extract the observation time ok = m0ptbufi (3, buffer, obtime) if (ok .eq. 0)then
write(chms,FMT='(i6)')obtime endif c--- extract the cloud height ok = m0ptbuff (4, buffer,
cloudheight) if (ok .eq. 0)then write(czcl,FMT='(f7.1)')cloudheight endif write(cline, &
FMT='(a4,2x,a5,1x,a6,1x,a6)') & id,ct(6:),chms,czcl call sdest(cline,0) goto 100 endif 999
continue call edest('done',0) return end
```

Reading the point-data file header

Occasionally, an application may need to access the file header associated with point data without accessing the data itself. The McIDAS-X ADDE command PTLIST with the FORM=FILE option is an example of such an application.

The **m0pthdr** function opens a connection to read point data file headers from the server based on the selection conditions shown in the table below.

Selection clause	Description
BPOS	beginning file in the dataset or ALL
EPOS	ending file in the dataset

Upon a successful return from **m0pthdr**, the **m0ptrdhdr** function is repeatedly called until all the data is returned. Below is a sample code fragment demonstrating the use of **m0pthdr** and **m0ptrdhdr**.

```
include 'ptparm.inc' character*24 dataset integer header(HEADSIZE) character*24 selects(2)
integer nselects c--- request file header positions 1 through 5 from the dataset c---
RTPTSRC/SFCHOURLY dataset = 'RTPTSRC/SFCHOURLY' selects(1) = 'BPOS=1' selects(2) =
'EPOS=5' nselects = 2 ok = m0pthdr(dataset, nselects, selects) if (ok .lt. 0)then return
endif 100 continue ok = m0ptrdhdr(header) c--- if there was an error if (ok .lt. 0)then
return c--- if a header was successfully returned elseif (ok .eq. 0)then (process header)
goto 100 endif
```

The contents of a header returned from **m0ptrdhdr** are described in the table below.

Word	Description
0	schema name
1	schema version number
2	schema registration date, *ccyyddd*
3	default number of rows
4	default number of columns
5	total number of keys in the record
6	number of keys in the row header
7	number of keys in the column header
8	number of keys in the data portion

9	1-based position of the column header
10	1-based position of the data portion
11	number of repeat groups
12	size of the repeat group
13	starting position of the repeat group
14	missing data code
15	integer ID of the file
16 - 23	text ID of the file
24	creator's project number
25	creation date, *ccyyddd*
26	creator's ID
27	zero-based offset to the row header
28	zero-based offset to the column header
29	zero-based offset to the data portion
30	first unused word in the file
31	start of the user record
32	start of the key names
33	start of the scale factors
34	start of the units
35 - 38	reserved
39	beginning Julian day of the data, *ccyyddd*
40	beginning time of the data, *hhmmss*
41	ending Julian day of the data
42	ending time of the data, *hhmmss*
43 - 60	reserved
61 - 62	file name
63	MD file number
64 - 463	user record, MD coordinates (0,0); not described by the schema; use for storing arbitrary information
464 - 863	names of the file keys
864 - 1264	scale factors for the keys
1264 - 1663	units of the keys
1664 - 4095	reserved

Words 39-42 are filled in only during the production of real-time files. When real-time file data is copied to other MD files, words 39-42 in the destination files are set to null.

Text data

Text data stores a variety of information for the McIDAS-X user, such as:

- Administrative messages
- Forecasts, advisories and warnings
- Observational data, such as METAR or RAOB reports

McIDAS-X has two types of text data: flat-file text and general weather text. Both are described in this section along with:

- the attributes that distinguish text data from other types of McIDAS-X data
- the blocks of information contained in text data
- the API functions available for reading text data

Flat-file text

Flat-file text is the simplest text data available in McIDAS-X. It is used most often on the server machine to convey administrative or configuration information to the user. The McIDAS-X READ command accesses this text.

The data is stored on the server machine as a simple ASCII file that can be manipulated with any file editor. It is delivered to the client one line at a time. There is no practical limit to the length of an individual line.

The table below lists the McIDAS-X library functions for flat file text.

Function	Description
M0textgt	opens a connection to read a flat file
M0txtread	reads the text data one line at a time
M0PrintTextFromServer	opens a connection to read a flat file and prints the file contents to the McIDAS-X Text Window

Opening a connection to read a flat file

Because there is a one-to-one relationship between the ADDE dataset name created on the server and the file being served, no selection conditions are needed to access most flat files. Only the dataset name is required.

The ADDE interface to flat-file text is through the function **M0textgt**. It has one selection condition, FILE=, which can be used to access files that don't have specific dataset names in ADDE. The WXTLIST command with the DIR option is the only command that uses this selection condition.

Reading the text data

Once the connection is opened with **M0textgt**, the **M0txtread** function is called continuously until no more lines of text data are found. The code example below demonstrates the **M0textgt/M0txtread** function pair.

```
char *dataset; char **selects; int ok; . . . ok = M0textgt (dataset, n_select, selects, 1)
; if (ok < 0) { return (ok); } while (ok == 0) { char line[1000]; ok = M0txtread (line,
sizeof (line)) ; if (ok == 0) { Mcprintf ("%s\n", line); } }
```

General weather text

General weather text is composed of information compiled by weather agencies and distributed to the user community. It typically contains forecasts, public announcements, advisories and warnings. In McIDAS-X, this type of data is ingested and made available from a server running McIDAS-XCD and can be accessed by the user with the McIDAS-X command WXTLIST.

WMO format standards

Because weather agencies provide this data, certain WMO format standards must be used during transmission. The sample text below demonstrates the WMO formatting standards used for text data transmission.

```
FPUS1 KMKE 251641 SFPWI WIZALL-252200- STATE FORECAST FOR WISCONSIN 1040AM CST SAT JAN 25
1997 ...LAKE SNOW WARNING IRON COUNTY IN THE LAKE SUPERIOR SNOW AREA TODAY.. .TODAY...
WINDY AND COLD WITH AREAS OF LIGHT SHOW OR FLURRIES. HEAVIER SQUALLS IN THE LAKE SUPERIOR
SNOW BELT OF IRON COUNTY. HIGHS SINGLE DIGITS NORTHWEST TO THE TEENS EXTREME EAST.
```

WMO header line

The first line of text is the WMO header line. In the example above, the first two characters, FP, contain the WMO product header. Most WMO product headers begin with one of the following characters:

WMO product header	Description
A	advisory
C	climatic
F	forecast
N	notice
R	river
S	surface
T	satellite
U	upper air

The second character of the WMO product header will vary. The third and fourth characters are usually the country code (US) of the bulletin. The country code is followed by a product number (5), which further specifies the type of bulletin being transmitted. The product number is followed by the ID of the station initiating the bulletin (KMKE). The final six digits are the day of the month and time of the bulletin (251641).

AWIPS header line

The second line of the bulletin is an AWIPS header inserted by the U.S. National Weather Service; it is optional. The first three characters are the product code (SFP). The next two or three characters contain the state/province or the station the report is valid for (WI).

General weather text functions

The table below lists the McIDAS-X library functions for reading general weather text.

Function	Description
M0wtxget	opens a connection to read the text
M0wtxread	reads the text

Opening a connection to read general weather text

In ADDE, the **M0wtxget** function opens a connection to the server. This function takes a variety of selection conditions, allowing the user to limit the amount of data searched to fulfill a user request. The more information specified in the selection conditions, the faster the search will be. The valid selection conditions are described in the table below.

Selection clause	Description	Remarks
APRO=*p1 .. pn*	list of AWIPS product types to match	three characters; don't use with WMO=
ASTN=*s1 .. sn*	list of AWIPS station IDs to match	two or three characters
DAY=	most recent day to search	

DTIME=	maximum number of hours of reports to search	
MATCH=*match1 .. n*	list of strings within the body of the text to match	if more than one match string is requested, all match strings requested must be found in each text block for a successful return
NUM=	maximum number of text blocks to list	
PROD=	predefined product name	to list the available predefined products, use WXTLIST DIR
SOURCE=*s1 .. sn*	list of circuit sources	seldom used
WMO=*w1 .. wn*	list of WMO headers to match	minimum of two characters; wildcard characters are allowed; see the help for the WXTLIST command; can't be used with APRO=
WSTN=*s1 .. sn*	list of WMO station IDs to match	four characters

Reading the text

Once **M0wtxget** opens the connection, the application makes repeated calls to **M0wtxread** until all matching text products are received. Each call to **M0wtxread** returns the actual text along with a 64-byte header containing information about the text data. The components of the header are described in the table below. Note that all character strings are sent blank padded.

Bytes	**Description**
0 - 3	circuit source
4 - 7	number of bytes of data
12 - 15	time of day that the data was received, *hhmmss*
16 - 19	4-character WMO product ID, such as FPUS
20 - 23	product number
24 - 27	4-character WMO station ID
28 - 31	3-character AWIPS product ID
32 - 35	2- or 3-character AWIPS station ID
36 - 39	3-character product origin (optional)
40 - 43	Julian day of the data
44 - 47	number of bytes per line
60 - 63	FAA catalog number (optional)

The sample code below demonstrates an ADDE general weather text request.

```
char **selects; char *text; char t_string[64]; char header[64]; int n_selects; selects =
VecNew (); sprintf (t_string, "DAY=%d", current_day); /* * we are going to acquire the
State Weather Roundups (SWR) from * Wisconsin, Minnesota and Michigan. */ selects = VecAdd
(selects, t_string); selects = VecAdd (selects, "APRO=SWR"); selects = VecAdd (selects,
"ASTN=WI MN MI"); selects = VecAdd (selects, "NUM=3"); n_selects = VecLen (selects); ok =
M0wtxget ("RTWXTEXT", n_selects, selects, 1); if (ok < 0) { Mceprintf ("No data found\n");
```

```
return (2); } while ((ok = M0wtxread (header, &text)) == 0) { /* text found, do something
*/ free (text); }
```

McIDAS navigation

All geophysical data must have a location to be meaningful. McIDAS-X stores geophysical data in the form closest to its native spatial structure; for example, remotely sensed data is stored in the image coordinates of the scanner line and element, while NWP model output is stored as model grid points. Since earth coordinates (for example, latitude, longitude and height above mean sea level) are easiest to interpret, McIDAS-X uses a process called *navigation* to convert a dataset's native coordinate system to earth coordinates and back again.

Navigating datasets implies the ability to transform any dataset into the spatial coordinates of any other dataset. McIDAS-X' ability to integrate and display a variety of datasets is made possible by its navigation subsystems.

This section describes:

- Basic navigation concepts, such as navigation transforms, subsystems and terminology
- APIs you will use with the navigation subsystems
- How to design a navigation type and implement an image navigation module

For more information about McIDAS-X coordinate systems, see *Chapter 2, Learning the Basics*. For additional information about map projections, see *Map Projections -- A Working Manual*, USGS Professional Paper 1395, U.S. Government Printing Office, Washington, 1987.

Basic concepts

To select and use the best McIDAS-X navigation subsystem for a given task, you need to understand the nature of navigation transforms. Navigation systems and transforms are described below, along with some navigation-specific terminology used throughout this section.

Navigation transforms

A *navigation transform* is a set of equations for converting a dataset's image or native coordinates to and from earth coordinates. Each dataset has its own navigation transform. However, navigation transforms can be grouped into classes or *types*. Within a type, the differences between transforms are quantitative only. Each type has a set of variables, or parameters, whose values uniquely specify an *instance* of that type. For example, the parameters for a tangent cone projection include the standard latitude and longitude, the scale factor, and the location of the pole relative to the projection surface's origin.

All navigated McIDAS-X data has an associated type and a full set of parameters defining an instance.

Navigation subsystems

McIDAS-X uses three navigation subsystems.

- Grid navigation
- Image navigation
- Frame navigation

Grid navigation

Grid navigation converts earth coordinates to and from the row and column locations in the grid structure. This subsystem is separate from image and frame navigation, and supports fewer projections These projection types are implemented in a single set of functions. To add other types, you would have to modify the functions and recompile all grid applications. Since only a single instance can be in use at one time, you must repeatedly reinitialize the subsystem to use multiple projections within an application.

For example, to compare two grids, A and B, you initialize grid navigation for A, compute the earth location of a grid point *a* in A, reinitialize the subsystem using B's parameters, and transform the earth location of *a* to a grid in B. This process is then repeated for every grid point being compared. Most applications that process or use grids use grid navigation.

For more information about grid projection formats, see the *Grid data* section earlier in this chapter.

Image navigation

Image navigation converts image coordinates to and from earth coordinates. Each image navigation type is implemented in its own source file or *module*, and each navigable image dataset has an associated navigation block that identifies the type and contains a complete set of parameters needed to define an instance of that type.

Each module implements the same function names and calling sequence. Since McIDAS-X uses the type identifier to select the correct module, you can add new types without modifying existing modules. In addition, up to three instances of navigation, of the same or different types, can be active

simultaneously in a single application through different *slots*. This requires a one- or two-step initialization process before use and is typically used by applications that work directly with image datasets.

Frame navigation

Frame navigation converts frame coordinates to and from earth coordinates. Frame navigation uses the image navigation subsystem but through a simpler API. It is sufficient for most applications that just add information to an existing display.

Terminology

This section contains some navigation-specific terms that may be unfamiliar to you. They are defined below.

- A *conformal projection* is a projection in which angles are preserved; for example, parallels of latitude and meridians of longitude intersect at right angles. Conformal projections preserve area at the expense of shape. McIDAS-X supports Mercator, Lambert conformal, polar stereographic, and tangent-cone conformal projections. The pseudo-Mercator (latitude/longitude) projection is not conformal.
- An *equal-area projection* is a projection in which areas are preserved; two equal areas on the Earth are also equal on the projection, even though their shapes are different. McIDAS-X supports the sinusoidal equal-area projection.
- The *geocentric latitude* of a point is the angle between the equatorial plane and a ray through the point from the Earth's center.
- *Geodetic latitude* is the angle between a line perpendicular to the surface of the geoid through a point and the Earth's equatorial plane. Due to the Earth's oblateness, geodetic latitudes (the most common form of earth location) are slightly greater than geocentric latitudes except at the equator and poles where they are identical.
- A *geoid* is the spheroid (surface formed by rotating an ellipse about the polar or Z axis of the terrestrial coordinate system) that most closely approximates the Earth's surface.
- A *navigation block* is a McIDAS-X data structure containing the projection type and set of projection parameters needed to compute transformations between earth and image coordinates. A navigation block is sometimes also called a *navigation codicil*.
- A *projection* is a set of equations relating earth locations (three variables) to a location in Cartesian coordinates on the projection plane. Distortions and even breaks, or interruptions, are unavoidable and are especially prominent when most or all of the Earth's surface is represented. Some projections are conformal or equal-area projections. The relationships between satellite image coordinates and earth coordinates can also be thought of as a projection, albeit a complicated one.
- *Projection parameters* are one or more constants contained in projection equations. Specifying values for these constants defines an instance of the projection.
- A *pseudo-Mercator projection* is a projection in which latitude and longitude vary uniformly with line (or row) and element (or column). This projection is distinct from a true Mercator and is neither conformal nor equal-area.

Using the navigation APIs

The McIDAS-X library provides Application Program Interfaces for geographic calculations as well as grid, image and frame navigation. These APIs are described below.

Geographic calculation API

Many calculations just require a change from one form of earth or planetary coordinate to another; for example, changing from a terrestrial (earth-relative) to celestial (star-relative) coordinate system, or converting a position from a Cartesian to spherical representation. The table below lists the functions for performing these tasks.

Functions	Description
m0distnc	calculates the great-circle distance on a sphere
geolat	performs geodetic/geocentric latitude conversions
nllxyz, nxyzll	perform spherical/Cartesian conversions
raerac, racrae	perform terrestrial/celestial longitude conversions
solarp	calculates solar position
cartll, llcart, llobl, llopt	perform generic spherical/Cartesian conversions

Grid navigation API

The table below shows the grid navigation API. You will use the **grddef** function with the header to initialize the subsystem, then call **ijll** and **llij** to transform grid coordinates to and from earth coordinates, respectively.

Function	Description
grddef	initializes the grid transformation
ijll	converts grid coordinates to earth coordinates
llij	converts earth coordinates to grid coordinates

No API functions currently exist for creating grid headers for a projection. When creating McIDAS-X grid files, you must determine the projection to use and then insert the projection type and parameters into the grid header. The grid header format is described in the *GRIDnnnn* data structure in Chapter 6.

Frame navigation API

Use the two functions below to determine the earth coordinates of a pixel or the pixel location of a point in earth coordinates.

Function	Description
illtv	converts earth coordinates to image and frame coordinates
itvll	converts frame coordinates to image and earth coordinates

The only prerequisite for these functions is that the frame has navigation associated with it. The McIDAS-X IMGDISP command creates frame navigation when it displays an image dataset. The McIDAS-X MAP, PTDISP and GRDDISP commands, which can define a map base before generating a display, also establish frame navigation.

The two examples below assume that frame navigation information exists. The first code fragment uses the **illtv** function to convert an earth location (**zlat,zlon**) to a frame location (**itvlin,itvele**), as is done in the McIDAS-X PC command.

```
zlat = 43.13 zlon = 89.35
if(illtv(iframe,gnum,zlat,zlon,ilin,iele,itvlin,itvele).ne.0)then call edest('unable to
navigate point',0) return endif
```

The second code fragment uses **itvll** to determine the earth location of the cursor, as is done in the McIDAS-X E command.

```
if( mcmoubtn( 0, left, right, itvlin, itvele ) ) then C // handle cursor-positioning error
here end if frame = mcgetimageframenumber() rc =
itvll(frame,itvlin,itvele,ilin,iele,rlat,rlon,iscene)
```

Image navigation API

Use the appropriate image navigation API functions from the table below if your application must perform one of these tasks:

- Navigate image datasets that have not been displayed
- Use navigation information from an independent source, such as system navigation files
- Make repeated navigation calculations for more than one dataset
- Use navigation services other than earth/image coordinate changes, such as satellite subpoint

Function	Description
nvprep	dynamically loads a navigation module for a specified slot, given a set of parameters
nvset	initializes navigation for a specified slot from a specified source
nvxeas	converts earth coordinates to image coordinates; due to dynamic linking, this function is called nv1eas, nv2eas or nv3eas from applications, depending on the navigation slot used
nvxini	initializes the navigation module; due to dynamic linking, this function is called nv1ini, nv2ini or nv3ini from applications, depending on the navigation slot used
nvxopt	performs special navigation services and is module-dependent; due to dynamic linking, this function is called nv1opt, nv2opt or nv3opt from applications, depending on the navigation slot used
nvxsae	converts image coordinates to earth coordinates; due to dynamic linking, this function is called nv1sae, nv2sae or nv3sae from applications, depending on the navigation slot used

To use image navigation services, the application must do the following:

- Obtain a navigation block
- Dynamically link the proper module for the type
- Initialize an instance of that type in a slot with the parameters

The example below, from **remap.pgm**, illustrates the initialization and use of two instances (slots) of navigation at the same time.

The lines below assume that navigation blocks were already obtained from the source (**snav**) and destination (**dnav**) areas. To link the appropriate navigation module to a slot and initialize it, use **nvprep**. The lines below initialize slot 1 for the source and slot 2 for the destination. The **nv1ini** and **nv2ini** calls set the earth coordinate form to Cartesian for both slots.

```
c --- check for initialization if( init.eq.0 ) then c ------ initialize the source
navigation if( nvprep( 1, snav(1) ).ne.0 ) then navinit = -1 return endif c ------
initialize the destination navigation if( nvprep( 2, dnav(1) ).ne.0 ) then navinit = -1
return endif init = 1 endif c ------ initialize the navigations c ------ trans can be 'LL
' for latitude longitude c ------ or 'XYZ ' for cartesian itrans = lit( trans(1:4) ) call
nv1ini(2, itrans) call nv2ini(2, itrans)
```

The lines below map image coordinate (**dline,delement**) in the destination image to image coordinate (**sline,selement**) in the source image by converting from destination image coordinates to earth coordinates, then from earth coordinates to source image coordinates.

```
c --- convert destination line/element to latitude/longitude status = nv2sae( dline,
delement, 0., dlat, dlon, dz ) if( status.ne.0 ) then crossnav = -1 return endif c ---
convert latitude/longitude to source line/element status = nv1eas( dlat, dlon, dz, sline,
selement, dz) if( status.ne.0 ) then crossnav = -2 return endif
```

When the navigation block source is a local McIDAS-X area format image file or a frame whose number is known and you only intend to use slot 1, you can use **nvset**. It reads the navigation block out of an area or frame and calls **nvprep**.

```
nvset ('AREA', area_num)
      -or-
nvset ('FRAME', frame_num)
```

In addition to the commonly used conversions between earth and image coordinates available through **nvxsae** and **nvxeas**, other special services are available through the **nvxopt** function. Some of these services and the modules that support them are listed in the table below.

Special services	Modules that support them
oblate radius	**nvxmerc, nvxps, nvxrect**
orbital period	**nvxdmsp**
parallax correction	**nvxgoes, nvxgvar, nvxmsat**
satellite location/subpoint	**nvxdmsp, nvxgoes, nvxgraf, nvxgvar, nvxlamb, nvxmerc, nvxmsat, nvxps, nvxradr, nvxrect, nvxsin, nvxtiro**
view angles (satellite, solar, etc.)	**nvxgoes, nvxgvar**

The sample code below shows the use of **nvxopt**. When using nvxopt, you need to be aware that the input/output can be different for the different satellite navigations, and you'll have to check the nvx *sat*.dlm file to check what is appropriate for input/output.

```
real xin(6) real xout(6) ... C --- initialize slot 1 navigation (navblk is assumed to be C
--- filled with valid parameters) if( nvprep( 1, navblk ) .ne.0 ) then C (error handler)
end if C --- call nvxopt(), and extract the outputs if( nv1opt( lit('SPOS'), xin, xout )
.ne.0 ) then C (error handler) end if lat = xout(1) lon = xout(2)
```

Implementing an image navigation module

You can extend image navigation to include a new type and make it available to all McIDAS-X commands without changing any applications or existing navigation modules. This section describes the process for implementing a tangent-cone map projection. The complete source listing is in the next section titled *Example navigation module code*.

Requirements

The new module must adhere to strict guidelines governing the following:

- Contents of the navigation block
- Module name
- Names and calling sequences of the functions that it implements
- Retention of parameters between calls
- Response to invalid inputs

Navigation block contents

The navigation block must consist of an array of no more than 640 Fortran integers that are sufficient to uniquely specify a particular instance of the new navigation type. The first word of the block must be four ASCII characters that uniquely identify the type, stored in an integer variable. The block contents are obvious once the algorithm is fully specified and understood.

Module name

The navigation module name must be **`nvx`***type*`.dlm`, where *type* is the same four characters specified in the first word of the block. The **nvprep** function uses the type from the block to link the application with the correct module.

Functions

The navigation module must implement the functions **nvxini**, **nvxsae**, **nvxeas** and **nvxopt**, with calling sequences following the documentation blocks in the tangent cone example. In particular, **nvxini** must support modes 1 (initialize) and 2 (set earth coordinate mode `'LL'` (Latitude-longitude) or `'XYZ'` (Cartesian)). The **nvxopt** function must provide, at minimum, an `'SPOS'` (subpoint) service.

Parameters

Between calls, the navigation module must remember the navigation parameters uniquely defining an instance. This is typically accomplished by storing these parameters, and quantities derived from them, in a named common block that is shared among all four API functions in the module.

Invalid inputs

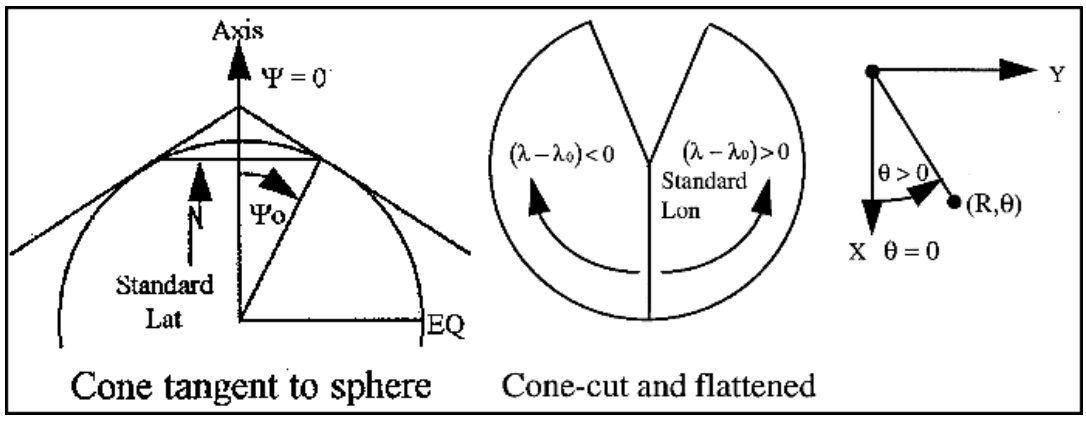
Satellites and map projections cannot physically represent all earth locations. The limits vary from type to type and even from instance to instance. Applications, therefore, cannot be expected to validate inputs to navigation modules. The new module must be able to recognize inputs it cannot handle and return an error status, typically a negative integer.

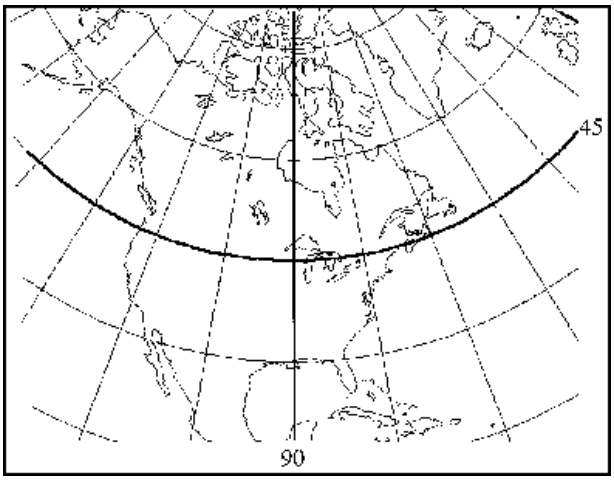
Algorithms

A navigation algorithm is a set of equations for mapping McIDAS-X image coordinates to earth coordinates and vice versa. For satellite navigation types, this typically involves a lengthy expression in vector notation. Map projections are simpler. Some manipulation and derivation of additional relationships is often necessary to extend a published relationship into a full algorithm.

The example presented here will show the development of a northern hemisphere tangent cone projection algorithm. The geometry of the tangent cone is shown in Figure 5-6 below. The cone is tangent to the planet at one parallel of latitude and the planetary axis pierces the apex of the cone. A line between a point on the Earth's surface and the pole opposite the cone apex maps that point onto the cone. The imaginary cone is then cut along a meridian opposite an arbitrary standard longitude and flattened. Each point on the Earth's surface in colatitude ψ and longitude λ corresponds to a point (R,θ) in polar coordinates on the flattened cone.

Figure 5-6. *Tangent cone projection geometry*.





A published form of the equations defining the location (R,θ) on the flattened cone in terms of an earth location (ψ,λ) is:

$$R = a\tan\psi_0\left[\left(\tan\frac{\psi}{2}\right)/\left(\tan\frac{\psi_0}{2}\right)\right]^{\cos\psi_0} \quad (1)$$

$$\theta = \cos\psi_0(\lambda-\lambda_0) \quad (2)$$

Also provided is an equation for the ratio of the distance between two closely spaced points on the cone to the distance between the corresponding points on the Earth's surface.

$$M(\psi) = m\left(\frac{\sin\psi_0}{\sin\psi}\right)\left[\frac{\tan\psi}{\tan\psi_0}\right]^{\cos\psi_0} \quad (3)$$

As you can see from (1) and (2), the planetary radius, a the standard colatitude ψ_0, and the standard longitude λ_0 define an instance of the tangent cone. In equation 3, m is the map scale expressed as a unitless ratio of projection plane distance to map distance.

This is not yet a complete algorithm ready for McIDAS-X implementation. The inverse relationship capable of converting earth location to projection coordinate (R,θ) must be derived, the earth location (λ,ψ) related to its equivalent in McIDAS-X (LAT, LON), and the projection coordinate (R,θ) related to McIDAS-X image coordinates. The inverse is:

$$\lambda = \lambda_0 + \frac{\theta}{\cos\psi_0} \quad (4)$$

and

$$\psi = 2\tan^{-1}\left[\tan\frac{\psi_0}{2}\left(\frac{R}{a\tan\psi_0}\right)^{\frac{1}{\cos\psi_0}}\right] \quad (5)$$

The relationships between McIDAS-X planetary coordinates and (λ,ψ) are:

$$\psi = \frac{\pi}{2} - \frac{\pi}{180}Lat \quad \lambda = -\frac{\pi}{180}Lon \quad (6)$$

and

$$Lat = 90 - \frac{180}{\pi}\psi \quad Lon = -\frac{180}{\pi}\lambda \quad (7)$$

Those between projection and image coordinates are:

$$L = L_0 + \frac{R\cos\theta}{m} \quad E = E_0 + \frac{R\sin\theta}{m} \quad (8)$$

and

$$R = m\sqrt{(L-L_0)^2 + (E-E_0)^2} \quad (9)$$

$$\theta = \tan^{-1}[(E-E_0)/(L-L_0)]$$

Adding (6) - (9) to make a complete McIDAS-X navigation algorithm adds additional parameters to make the complete set ψ_0, λ_0, m, L_0, and E_0, where the latter specify the scale in km per pixel and the location of the pole of the projection surface in image coordinates.

The last step before implementation is to examine the completed algorithm for limits and singularities; these will serve as the basis for input validation. The limits on the inputs are summarized below.

Parameters	
$0 < \psi_0 < \pi/2$	standard colatitude is confined to the northern hemisphere
$-\pi < \lambda_0 \leq \pi$	standard longitude must be a legal value
Earth coordinates	
$0 \leq \psi < \pi$	all colatitudes are navigable except the south pole; note that map scale $M(\psi)$ is undefined at the north pole,
$-\pi < \lambda \leq \pi$	however all longitudes are navigable
Image coordinates	
$R \geq 0$	no valid earth location maps to -R
$\pi \cos(\psi_0) < \theta \leq \pi \cos(\psi_0)$	area in the split region of the flattened cone isn't navigable; exclude it as input

Implementation

Implementing the navigation module consists of three steps:

1. Specifying the navigation block format
2. Creating or modifying the routines and applications that will create instances of the new type
3. creating the navigation module for the new type

Each of these steps is discussed below.

Defining the navigation block

The contents of the navigation block follow directly from the parameters provided in the previous *Algorithms* section. The five parameters are:

- Standard latitude
- Standard longitude
- Pole line
- Pole element
- Scale

A sixth parameter, planetary radius, could also be added but is not included in the examples. The parameters can be represented in different ways, such as standard colatitude in place of latitude. However, since the applications will create navigation blocks, you should use units and quantities convenient for McIDAS-X, as shown in the lines of code below. This code is taken from the documentation block at the beginning of **nvxini** and is shown in lines 0017 through 0022 in the complete source listing that follows.

```
*$ param( 1) = 'TANC' *$ param( 2) = image line of pole *10000 *$ param( 3) = image
element of pole*10000 *$ param( 4) = km per pixel *10000 *$ param( 6) = standard latitude
*10000 *$ param( 7) = standard longitude *10000
```

All instances of **'TANC'** navigation must be specified with these parameters.

Creating navigation blocks in applications

Applications that create navigation blocks must provide values for a complete set of parameters. Sometimes they must be derived from user inputs using the navigation transformation itself. Below is a code fragment from an application (not the navigation module itself) that creates a tangent-cone navigation block. It assumes that the center latitude and longitude (**clat** and **clon**), the standard latitude and longitude (**slat** and **slon**), and the scale of standard latitude (**sscale**), in km per pixel, are already provided. First, the earth locations are converted from McIDAS-X to projection form.

```
psi = PI/2.d0 - D2R*clat psi_0 = PI/2.d0 - D2R*slat lam = - D2R*clon lam_0 = - D2R*slon
```

Next, the radius of the specified image coordinate center point (relative to the projection center or the pole) is computed.

```
radius = A * tan(psi_0) * $ (tan(psi/2.d0)/tan(psi_0/2.d0))**cos(psi_0) theta =
cos(psi_0)*(lam-lam_0)
```

Then the pole's image coordinate is computed using simple trigonometry.

```
call status = mcfsize(mcgetimageframenumber, nlins, neles) lin_c = dble(nlins) / 2.d0
ele_c = dble(neles) / 2.d0 lin_0 = lin_c - radius*cos(theta)/sscale ele_0 = ele_c -
radius*sin(theta)/sscale
```

Once the parameters are computed, the navigation block can be filled and written either to the image dataset:

```
C // Create and insert the navigation block nvblk( 1) = lit('TANC') ! Tangent Cone nav
type nvblk( 2) = nint(10000*lin_0) ! image line of pole nvblk( 3) = nint(10000*ele_0) !
image element of pole nvblk( 4) = nint(10000*sscale) ! scale km/pixel at slat nvblk( 5) =
nint(10000*slat) ! standard latitude nvblk( 6) = nint(10000*slon) ! standard longitude
status = mcaput(dataset, nsort, sort, adir, nvblk, calblk)
```

or a McIDAS-X area format image file:

```
call araput(aranum,4*DIRSIZ,4*NVWDS,nvblk)
```

Creating the navigation module

If the algorithm and navigation block are complete and unambiguous, implementing a map projection as a McIDAS-X navigation module is generally straightforward. Your major design decision is what to store in the common block. At minimum, you should include the navigation parameters or quantities derived from them without loss of information. This is necessary for the module to remember the characteristics of the instance.

Because navigation modules are called many times, you should precompute additional quantities in the initialization routine **nvxini** and store them in common, especially if doing so eliminates trigonometric function calls in the other functions. Thus, the variables **`Coscl`**, **`Tancl`**, **`Tancl2`**, and **`Mxtheta`** are included in the common block below. This code is located in lines 0099 through 0120 of the source listing that follows.

```
C // Common block contents: precomputed intermediate values real Coscl ! cosine(Colat0)
real Tancl ! tangent(Colat0) real Tancl2 ! tangent(Colat0/2) real Mxtheta ! limit of angle
from std. lon C ! on projection surface C // Common block contents: constants real D2R !
degrees to radians factor real Pi real Badreal ! returned when navigation C ! 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
```

Also included in the block are physical constants and two flags; **`Init`** is set when **nvxini** runs successfully, and **`Latlon`** is set if the earth coordinate mode is latitude and longitude rather than Cartesian.

The overall structure of **nvxini** is a large if-block to handle the two options. Option 1 (lines 0144 - 0220 in the source listing) initializes the instance by range checking the navigation parameters and precomputing the quantities to retain in the common block. Option 2 (lines 0222 - 0233) sets the earth coordinate mode based on word **`param(1)`**.

The forward (image to Earth) navigation routine's one subtlety is input validation. Points within the split of the projection plane, which is shown in Figure 5-6, are not navigable.

Recognizing such points is possible only after the conversion from McIDAS-X image coordinates to projection coordinates is complete. The code below is from lines 0370 through 0384.

```
C // Compute radius and bearing from pole dx = Scale*(lin-Lin0) dy = Scale*(ele-Ele0)
radius = sqrt(dx*dx+dy*dy) theta = atan2(dy,dx) C // Apply range checking on theta to
determine if point is navigable if ( theta.le.-Mxtheta .or. theta.gt.Mxtheta ) then NVXSAE
= -1 return end if
```

When range checking is done, the projection is inverted and the colatitude and longitude (in radians, east positive) is converted to latitude and longitude (in degrees, west positive); see lines 0389 - 0401. A call to **nllxyz** to generate Cartesian coordinate output is made if the **`Latlon`** flag was cleared by an earlier call to **nvxini** (line 0404). Inverse (earth-to-image) navigation in **nvxeas** follows a similar pattern, except the earth-coordinate option must be handled at the beginning (line 0544).

The **nvxopt** function performs special services. Since the contents of the argument vector depend on the option selected, the routine consists of a large if-block (lines 0716 - 0742) with one branch per recognized option and the necessary input validation done within each branch. As in **nvxini**, the code must be able to recognize options that it doesn't know and return an appropriate error status. All core McIDAS-X navigation modules have an **`SPOS`** option that returns the subpoint at a given time (for satellites) or the latitude and longitude of the center (maps). The example that follows implements only an **`SCAL`** to return the map scale factor at any latitude.

---

1. Saucier, W. J. 1983: *Principles of meteorological analysis*. Dover Publications, Inc., New York. 438 pp.

Example navigation module code

```
0001 C THIS IS SSEC PROPRIETARY SOFTWARE - ITS USE IS RESTRICTED. 0002 0003 C *** McIDAS Revision History *** 0004 C *** McIDAS
Revision History *** 0005 0006 *$ Name: 0007 *$ nvxini - Initialize navigation for tangent cone projection 0008 *$ 0009 *$
Interface: 0010 *$ integer function 0011 *$ nvxint(integer option, integer param(*)) 0012 *$ 0013 *$ Input: 0014 *$ option - 1 to
set or change projection parameters 0015 *$ option - 2 set output option 0016 *$ param - For option 1: 0017 *$ param( 1) = 'TANC'
0018 *$ param( 2) = image line of pole*10000 0019 *$ param( 3) = image element of pole*10000 0020 *$ param( 4) = km per pixel *10000
0021 *$ param( 6) = standard latitude *10000 0022 *$ param( 7) = standard longitude *10000 0023 *$ for option 2: 0024 *$ param( 1) =
'LL' or 'XYZ' 0025 *$ 0026 *$ Input and Output: 0027 *$ none 0028 *$ 0029 *$ Output: 0030 *$ none 0031 *$ 0032 *$ Return values:
0033 *$ 0 - success 0034 *$ -3 - invalid or inconsistent navigation parameters 0035 *$ -4 - invalid navigation parameter type 0036
*$ -5 - invalid nvxini() option 0037 *$ 0038 *$ Remarks: 0039 *$ Latitudes and longitudes are in degrees, West positive. 0040 *$
Projection parameters must be in the following ranges: 0041 *$ 0. < standard latitude < 90. 0042 *$ -180. <= standard longitude <
180. 0043 *$ 0. < scale 0044 *$ Accuracy may suffer near the standard latitude limits. 0045 *$ 0046 *$ The projection algorithm is
adapted from that in 0047 *$ Saucier, W. J. 1989: Principles of meteorological analysis. 0048 *$ Dover Publications, Inc. 433 pp.
0049 *$ 0050 *$ Categories: 0051 *$ navigation 0052 0053 C // CODING CONVENTION note: function declarations and common 0054 C //
block declarations are all capitalized to be recognizable 0055 C // to script 'convdlm;' this is necessary for a correct build 0056
C // in MCIDAS-X. For the same reason, avoid referring to 0057 C // function or common block names in uppercase elsewhere 0058 0059
INTEGER FUNCTION NVXINI(option,param) 0060 0061 0062 implicit NONE 0063 0064 0065 C // Interface variables (formal arguments) 0066
0067 integer option ! initialization option 0068 integer param(*) ! navigation parameters or 0069 C ! output coordinate type 0070
0071 C // Local variable definitions 0072 0073 character*4 navtyp ! codicil type 0074 character*4 outcoord ! output coordinate type
0075 real lat0 ! standard latitude 0076 character*80 cbuf ! text output buffer 0077 0078 0079 0080 C
////////////////////////////////////////////////////// 0081 C Common block variables and declaration. 0082 0083 C ALL CODE
BETWEEN THE '//////' SEPARATORS MUST BE 0084 C DUPLICATED EXACTLY IN EACH NAVIGATION ROUTINE 0085 0086 C (A more maintenance-safe
version would use ENTRY points 0087 C rather than separate functions for the navigation API 0088 C but entry points cannot be
processed by 'convdlm.') 0089 0090 C // Common block contents: projection parameters 0091 0092 real Lin0 ! image line of pole 0093
real Ele0 ! image element of pole 0094 real Scale ! km per unit image coordinate 0095 C ! (pixel) 0096 real Lon0 ! standard
longitude 0097 real Colat0 ! standard colatitude 0098 0099 C // Common block contents: precomputed intermediate values 0100 0101
real Coscl ! cosine(Colat0) 0102 real Tancl ! tangent(Colat0) 0103 real Tancl2 ! tangent(Colat0/2) 0104 real Mxtheta ! limit of
angle from std. lon 0105 C ! on projection surface 0106 0107 C // Common block contents: constants 0108 0109 real D2R ! degrees to
radians factor 0110 real Pi 0111 real Badreal ! returned when navigation 0112 C ! cannot be done 0113 real Erad ! Earth radius 0114
logical Init ! initialized flag 0115 logical Latlon ! .TRUE. for lat/lon I/O 0116 0117 0118 COMMON/TANC/ Lin0, Ele0, Scale, Lon0,
Colat0, 0119 & Coscl, Tancl, Tancl2, Mxtheta, 0120 & D2R, Pi, Badreal, Erad, Init, Latlon 0121 0122 C End of common block variables
and declaration. 0123 C ////////////////////////////////////////////////////// 0124 0125 0126 0127 C // Begin initialization
process by setting values of constants. 0128 0129 Erad = 6370. ! This value of Erad is ok 0130 C ! for low precision nav 0131 C !
where a spherical Earth is 0132 C ! adequate (Saucier, p. 32) 0133 Pi = acos(-1.) 0134 D2R = Pi / 180. 0135 Badreal = -1.E10 !
obvious unreasonable value 0136 C ! for nav transform result 0137 0138 C // Process initialization options. Only one option,
initialize 0139 C // navigation parameters, is supported in this demo version, 0140 C // but a 'hook' is left for an additional
option to set the 0141 C // output coordinate to something other than lat/lon 0142 0143 if( option.eq.1 ) then 0144 0145 call
DDEST('nvxini(tanc) option=1',0) 0146 0147 call movwc(param(1),navtyp) 0148 if( navtyp.eq.'TANC') then 0149 0150 C // Unpack tangent
cone projection parameters 0151 0152 Lin0 = param(2) / 10000. 0153 Ele0 = param(3) / 10000. 0154 Scale = param(4) / 10000. 0155 lat0
= param(5) / 10000. 0156 Lon0 = param(6) / 10000. 0157 0158 write(cbuf,'('' nvxini: lat0, Lon0 '',2F12.4)') 0159 * lat0, Lon0 0160
call DDEST(cbuf,0) 0161 0162 C // apply range checking 0163 0164 if(Scale.le.0. ) then 0165 call DDEST('nvxini(tanc) scale is
negative',0) 0166 Init = .FALSE. 0167 NVXINI = -3 0168 return 0169 end if 0170 0171 if(lat0.le.0. .or. lat0.ge.90. ) then 0172 call
DDEST('nvxini(tanc) standard lat out of range',0) 0173 Init = .FALSE. 0174 NVXINI = -3 0175 return 0176 end if 0177 0178
if(Lon0.le.-180. .or. Lon0.gt.180. ) then 0179 call DDEST('nvxini(tanc) standard lon out of range',0) 0180 Init = .FALSE. 0181
NVXINI = -3 0182 return 0183 end if 0184 0185 C // convert degrees to radians and latitude to colatitude. 0186 C // Account for
McIDAS longitude convention 0187 0188 Lon0 = -Lon0 * D2R 0189 Colat0 = Pi/2. - D2R*lat0 0190 0191 write(cbuf,'('' nvxini: Colat0,
Lon_0 '',2F12.4)') 0192 * Colat0, Lon0 0193 call DDEST(cbuf,0) 0194 0195 C // Compute intermediate quantities 0196 0197 Coscl =
cos(Colat0) 0198 Tancl = tan(Colat0) 0199 Tancl2 = tan(Colat0/2.) 0200 Mxtheta = Pi*Coscl 0201 0202 write(cbuf,'('' nvxini: Coscl,
Tancl'', 2F7.4)') 0203 * Coscl, Tancl 0204 call DDEST(cbuf,0)

0205 write(cbuf,'('' nvxini: Tancl2, Mxtheta '', 2F7.4)') 0206 * tancl2, Mxtheta 0207 call DDEST(cbuf,0) 0208 0209 Latlon = .TRUE.
0210 0211 0212 else ! option=1 but type not 'TANC' 0213 0214 call DDEST('nvxini(tanc) parameter type bad',0) 0215 Init = .FALSE.
0216 NVXINI = -4 0217 return 0218 0219 end if 0220 0221 else if ( option .eq. 2) then 0222 0223 call movwc(param(1),outcoord) 0224
if( outcoord.eq.'LL' ) then 0225 Latlon = .TRUE. 0226 else if( outcoord.eq.'XYZ') then 0227 Latlon = .FALSE. 0228 else 0229 call
DDEST('option=2 coord '//outcoord//' not supported',0) 0230 Init = .FALSE. 0231 NVXINI = -5 0232 end if 0233 0234 else ! option not
1 or 2 0235 0236 call DDEST('nvxini(tanc) unrecognized output option ',option) 0237 NVXINI = -4 0238 return 0239 0240 end if 0241
0242 NVXINI = 0 0243 Init = .TRUE. 0244 0245 return 0246 end 0247 0248 0249 0250 0251 *$ Name: 0252 *$ nvxsae - Compute earth
coordinates from image coordinates 0253 *$ 0254 *$ Interface: 0255 *$ integer function 0256 *$ nvxsae( real lin, real ele, real
dummy, 0257 *$ real e1, real e2, real e3 ) 0258 *$ 0259 *$ Input: 0260 *$ lin - image line 0261 *$ ele - image element 0262 *$ dummy
- (unused) 0263 *$ 0264 *$ Input and Output: 0265 *$ none 0266 *$ 0267 *$ Output: 0268 *$ e1 - latitude or x 0269 *$ e2 - longitude
or y 0270 *$ e3 - height or z 0271 *$ 0272 *$ Return values: 0273 *$ 0 - success 0274 *$ -1 - input data physically valid but not
navigable 0275 *$ given the specified projection 0276 *$ -6 - module not initialized 0277 *$ 0278 *$ Remarks: 0279 *$ The navigation
module must first be initialized with 0280 *$ a call to nvxini(). The output form (lat,lon) or (x,y,z) 0281 *$ depends on the last
call to nvxini() with option 2. 0282 *$ 0283 *$ Categories: 0284 *$ navigation 0285 0286 INTEGER FUNCTION NVXSAE( lin, ele, dummy,
e1, e2, e3 ) 0287 0288 implicit NONE 0289 0290 C // Interface variables (formal arguments) 0291 0292 real lin ! image line to
navigate 0293 real ele ! image element to navigate 0294 real dummy ! (unused argument) 0295 real e1 ! Earth coordinate 1 0296 real
e2 ! Earth coordinate 2 0297 real e3 ! Earth coordinate 3 0298 0299 C // Local variables 0300 0301 real lat ! latitude (McIDAS
convention) 0302 real lon ! longitude (McIDAS convention) 0303 real hgt ! height 0304 real dx ! zonal displacement from pole 0305 C
! on projection surface 0306 real dy ! meridional displacement from pole 0307 real radius ! distance from pole on projection 0308
real theta ! angle from standard longitude on 0309 C ! projection surface 0310 real colat ! colatitude of navigated point 0311 0312
0313 C ////////////////////////////////////////////////////// 0314 C Common block variables and declaration. 0315 0316 C ALL CODE
BETWEEN THE '//////' SEPARATORS MUST BE 0317 C DUPLICATED EXACTLY IN EACH NAVIGATION ROUTINE 0318 0319 C (A more maintenance-safe
version would use ENTRY points 0320 C rather than separate functions for the navigation API 0321 C but entry points cannot be
processed by 'convdlm.') 0322 0323 C // Common block contents: projection parameters 0324 0325 real Lin0 ! image line of pole 0326
real Ele0 ! image element of pole 0327 real Scale ! km per unit image coordinate 0328 C ! (pixel) 0329 real Lon0 ! standard
longitude 0330 real Colat0 ! standard colatitude 0331 0332 C // Common block contents: pre-computed intermediate values 0333 0334
real Coscl ! cosine(Colat0) 0335 real Tancl ! tangent(Colat0) 0336 real Tancl2 ! tangent(Colat0/2) 0337 real Mxtheta ! limit of
angle from std. lon 0338 C ! on projection surface 0339 0340 C // Common block contents: constants 0341 0342 real D2R ! degrees to
radians factor 0343 real Pi 0344 real Badreal ! returned when navigation 0345 C ! cannot be done 0346 real Erad ! Earth radius 0347
logical Init ! initialized flag 0348 logical Latlon ! .TRUE. for lat/lon I/O 0349 0350 COMMON/TANC/ Lin0, Ele0, Scale, Lon0, Colat0,
0351 & Coscl, Tancl, Tancl2, Mxtheta, 0352 & D2R, Pi, Badreal, Erad, Init, Latlon 0353 0354 C End of common block variables and
declaration. 0355 C ////////////////////////////////////////////////// 0356 0357 e1 = Badreal 0358 e2 = Badreal 0359 e3 = Badreal
0360 0361 0362 C // verify initialized module 0363 0364 if(.not.Init) then 0365 NVXSAE = -6 0366 return 0367 end if 0368 0369 0370 C
```

// Compute radius and bearing from pole 0371 0372 dx = Scale*(lin-Lin0) 0373 dy = Scale*(ele-Ele0) 0374 0375 radius = sqrt(dx*dx+dy*dy) 0376 theta = atan2(dy,dx) 0377 0378 0379 C // Apply range checking on theta to determine if point is navigable 0380 0381 if (theta.le.-Mxtheta .or. theta.gt.Mxtheta) then 0382 NVXSAE = -1 0383 return 0384 end if 0385 0386 C // Forward navigation: compute longitude and colatitude 0387 C // from radius and theta 0388 0389 lon = Lon0 + theta/Coscl 0390 if(lon.le.-Pi) lon = lon + 2.d0*Pi 0391 if(lon.gt. Pi) lon = lon - 2.d0*Pi 0392 0393 colat = 2.*atan(Tancl2 * (radius/(Erad*Tancl))**(1./Coscl)) 0394 0395 C // Rescale to McIDAS convention (degrees, West positive). 0396 C // Apply conversion to Cartesian coordinates if 'XYZ' set 0397 C // as output form. Set return code for success. 0398 0399 lon = -lon/D2R 0400 lat = 90. - colat/D2R 0401 hgt = 0. 0402 0403 if(.not.Latlon) then 0404 call nllxyz(lat,lon,e1,e2,e3) 0405 else 0406 e1 = lat 0407 e2 = lon 0408 e3 = 0. 0409 end if 0410 0411 NVXSAE = 0 0412 0413 return 0414 end

0415 *$ Name: 0416 *$ nvxeas - Compute image coordinates from earth coordinates 0417 *$ 0418 *$ Interface: 0419 *$ integer function 0420 *$ nvxeas(real e1, real e2, real e3, 0421 *$ real lin, real ele, real dummy) 0422 *$ 0423 *$ Input: 0424 *$ e1 - latitude or x 0425 *$ e2 - longitude or y 0426 *$ e3 - height or z 0427 *$ 0428 *$ Input and Output: 0429 *$ none 0430 *$ 0431 *$ Output: 0432 *$ lin - image line 0433 *$ ele - image element 0434 *$ dummy - (unused) 0435 *$ 0436 *$ Return values: 0437 *$ 0 - success 0438 *$ -1 - input data physically valid but not navigable 0439 *$ given the specified projection 0440 *$ -2 - input data exceed physical limits 0441 *$ -6 - module not initialized 0442 *$ 0443 *$ Remarks: 0444 *$ The navigation module must first be initialized with 0445 *$ a call to nvxini(). The input form (lat,lon) or (x,y,z) 0446 *$ depends on the last call to nvxini() with option 2. 0447 *$ Input longitude may be in the range -360 to +360; 0448 *$ values outside this range will not be de-navigated. 0449 *$ Height (hgt) is ignored. 0450 *$ 0451 *$ Categories: 0452 *$ navigation 0453 0454 INTEGER FUNCTION NVXEAS(e1, e2, e3, lin, ele, dummy) 0455 0456 implicit NONE 0457 0458 C // Interface variables (formal arguments) 0459 0460 real e1 ! Earth coordinate 1 0461 real e2 ! Earth coordinate 2 0462 real e3 ! Earth coordinate 3 0463 real lin ! image line to navigate 0464 real ele ! image element to navigate 0465 real dummy ! (unused argument) 0466 0467 C // Local variables 0468 0469 real lat ! latitude (McIDAS convention) 0470 real lon ! longitude (McIDAS convention) 0471 real hgt ! height 0472 real in_lon ! input longitude (radians, 0473 C ! East positive) 0474 real colat ! colatitude 0475 real radius ! distance from pole on projection 0476 real theta ! angle from standard longitude on 0477 C ! projection surface 0478 0479 C // 0480 C Common block variables and declaration. 0481 0482 C ALL CODE BETWEEN THE '//////' SEPARATORS MUST BE 0483 C DUPLICATED EXACTLY IN EACH NAVIGATION ROUTINE 0484 0485 C (A more maintenance-safe version would use ENTRY points 0486 C rather than separate functions for the navigation API 0487 C but entry points cannot be processed by 'convdlm.') 0488 0489 C // Common block contents: projection parameters 0490 0491 real Lin0 ! image line of pole 0492 real Ele0 ! image element of pole 0493 real Scale ! km per unit image coordinate 0494 C ! (pixel) 0495 real Lon0 ! standard longitude 0496 real Colat0 ! standard colatitude 0497 0498 C // Common block contents: precomputed intermediate values 0499 0500 real Coscl ! cosine(Colat0) 0501 real Tancl ! tangent(Colat0) 0502 real Tancl2 ! tangent(Colat0/2) 0503 real Mxtheta ! limit of angle from std. lon 0504 C ! on projection surface 0505 0506 C // Common block contents: constants 0507 0508 real D2R ! degrees to radians factor 0509 real Pi 0510 real Badreal ! returned when navigation 0511 C ! cannot be done 0512 real Erad ! Earth radius 0513 logical Init ! initialized flag 0514 logical Latlon ! .TRUE. for lat/lon I/O 0515 0516 0517 COMMON/TANC/ Lin0, Ele0, Scale, Lon0, Colat0, 0518 & Coscl, Tancl, Tancl2, Mxtheta, 0519 & D2R, Pi, Badreal, Erad, Init, Latlon 0520 0521 C End of common block variables and declaration. 0522 C // 0523 0524 lin = Badreal 0525 ele = Badreal 0526 dummy = Badreal 0527 0528 C // verify that module is initialized 0529 0530 if(.not.init) then 0531 NVXEAS = -6 0532 return 0533 end if 0534 0535 C // Preprocess input values. If mode is 'XYZ' first convert 0536 C // from Cartesian to lat/lon. If mode is 'LL' just transcribe 0537 C // from arguments. 0538 0539 if(Latlon) then 0540 lat = e1 0541 lon = e2 0542 hgt = e3 0543 else 0544 call nxyzll(e1, e2, e3, lat, lon) 0545 hgt = 0. 0546 end if 0547 0548 C // check that input values are physically possible and 0549 C // then convert to radians and East positive 0550 0551 if (lat.lt.-90. .or. lat.gt.90.) then 0552 NVXEAS = -2 0553 return 0554 end if

0555 if(lon.le.-360..or.lon.gt.360.) then 0556 NVXEAS = -2 0557 return 0558 end if 0559 0560 if(lat.eq.-90. .or. lat.eq.90.) then 0561 NVXEAS = -1 0562 return 0563 end if 0564 0565 colat = Pi/2. - D2R*lat 0566 in_lon = -D2R*lon 0567 0568 C // map longitude into range -Pi to Pi 0569 0570 if(in_lon.le.-Pi) in_lon = in_lon + 2.*Pi 0571 if(in_lon.gt. Pi) in_lon = in_lon - 2.*Pi 0572 0573 0574 C // Now trap South Pole. Though a physically possible latitude, 0575 C // tan(colat/2) -> infinity there so it is not navigable 0576 0577 if (colat.eq.Pi) then 0578 NVXEAS = -1 0579 return 0580 end if 0581 0582 0583 C // Compute radius and theta of point on projection surface. 0584 C // Theta is tricky; you have to compute offset relative 0585 C // to standard longitude, force that into -pi to +pi range, 0586 C // and THEN scale by cos(Colat0) 0587 0588 radius = Erad * Tancl *(tan(colat/2.)/Tancl2) ** Coscl 0589 theta = in_lon-Lon0 0590 0591 if(theta.le.-Pi) theta = theta + 2.*Pi 0592 if(theta.gt. Pi) theta = theta - 2.*Pi 0593 0594 theta = Coscl * theta 0595 0596 0597 C // Compute line and element 0598 0599 lin = Lin0 + radius*cos(theta)/Scale 0600 ele = Ele0 + radius*sin(theta)/Scale 0601 dummy = 0. 0602 0603 NVXEAS = 0 0604 0605 return 0606 end 0607 0608 0609 *$ Name: 0610 *$ nvxopt - Perform supplemental navigation operations 0611 *$ 0612 *$ Interface: 0613 *$ integer function 0614 *$ nvxopt(integer option, real xin(*), 0615 *$ real xout(*)) 0616 *$ Input: 0617 *$ option - 'SCAL' compute projection scale 0618 *$ xin(1) - latitude 0619 *$ 0620 *$ Input and Output: 0621 *$ none 0622 *$ 0623 *$ Output: 0624 *$ xout(1) - km per pixel at given latitude

0625 *$ Return values: 0626 *$ 0 - success 0627 *$ -1 - input latitude physically valid, but projection 0628 *$ undefined or scale infinite there 0629 *$ -2 - input latitude exceeds physical limits 0630 *$ -5 - unrecognized option 0631 *$ -6 - module not initialized 0632 *$ 0633 *$ Remarks: 0634 *$ The navigation module must first be initialized by 0635 *$ a call to nvxini(). Latitude is in degrees, north positive, 0636 *$ and must lie between -90. and +90. 0637 *$ 0638 *$ Categories: 0639 *$ navigation 0640 0641 INTEGER FUNCTION NVXOPT(option, xin, xout) 0642 0643 implicit NONE 0644 0645 C // Interface variables (formal arguments) 0646 0647 integer option ! special service name (character 0648 C ! stored as integer) 0649 real xin(*) ! input vector 0650 real xout(*) ! output vector 0651 0652 C // Local variables 0653 0654 character*4 copt ! special service (character form) 0655 real colat ! input colatitude 0656 0657 C // 0658 C Common block variables and declaration. 0659 0660 C ALL CODE BETWEEN THE '//////' SEPARATORS MUST BE 0661 C DUPLICATED EXACTLY IN EACH NAVIGATION ROUTINE 0662 0663 C (A more maintenance-safe version would use ENTRY points 0664 C rather than separate functions for the navigation API 0665 C but entry points cannot be processed by 'convdlm.') 0666 0667 C // Common block contents: projection parameters 0668 0669 real Lin0 ! image line of pole 0670 real Ele0 ! image element of pole 0671 real Scale ! km per unit image coordinate 0672 C ! (pixel) 0673 real Lon0 ! standard longitude 0674 real Colat0 ! standard colatitude 0675 0676 C // Common block contents: precomputed intermediate values 0677 0678 real Coscl ! cosine(Colat0) 0679 real Tancl ! tangent(Colat0) 0680 real Tancl2 ! tangent(Colat0/2) 0681 real Mxtheta ! limit of angle from std. lon 0682 C ! on projection surface 0683 0684 C // Common block contents: constants 0685 0686 real D2R ! degrees to radians factor 0687 real Pi 0688 real Badreal ! returned when navigation 0689 C ! cannot be done 0690 real Erad ! Earth radius 0691 logical Init ! initialized flag 0692 logical Latlon ! .TRUE. for lat/lon I/O 0693 0694 0695 COMMON/TANC/ Lin0, Ele0, Scale, Lon0, Colat0, 0696 & Coscl, Tancl, Tancl2, Mxtheta, 0697 & D2R, Pi, Badreal, Erad, Init, Latlon 0698 0699 C End of common block variables and declaration. 0700 C // 0701 0702 0703 xout(1) = Badreal 0704 0705 C // verify initialized module 0706 0707 if(.not.init) then 0708 NVXOPT = -6 0709 return 0710 end if 0711 0712 C // Extract and interpret the option 0713 0714 call movwc(option,copt) 0715 0716 if(copt.eq.'SCAL') then 0717 0718 C // Compute colatitude and make sure it is 0719 C // physically possible and navigable 0720 0721 if (xin(1).gt.90. .or. xin(1).lt.-90.) then 0722 NVXOPT = -2 0723 return 0724 else if (xin(1).eq.90. .or. xin(1).eq.-90.) then 0725 NVXOPT = -1 0726 return 0727 end if 0728 0729 colat = Pi/2. - D2R*xin(1) 0730 0731 C // Now compute actual scale for this colatitude 0732 0733 xout(1) = scale 0734 * *(sin(Colat0)* (tan(colat/2.)/Tancl2)**Coscl)/sin(colat) 0735 0736 C else if(copt.eq.'????') 0737 C // Add code for additional options here 0738 0739 else 0740 NVXOPT = -5 0741 return 0742 end if 0743 0744 NVXOPT = 0 0745 0746 return 0747 end

McIDAS calibration

Calibration is the process of converting data values sensed by an instrument to useful, physical quantities such as temperature, radiance and albedo. The McIDAS-X calibration subsystem is designed to:

- Allow the addition of new datasets from a variety of platforms, such as satellite, aircraft, and radar, with no changes to existing software or file format
- Be extensible, so that new data or calibration techniques for existing data types can be included

Thus, you can define your own calibration modules that will allow McIDAS-X applications to view the data that you prescribe.

This section describes:

- Calibration API
- Design and structure of a calibration module
- How to write a calibration module and integrate it into McIDAS-X
- Additional guidelines to use when writing a calibration module

Basic concepts

McIDAS-X has a defined Application Program Interface, including subroutine names, calling sequence and functionality, that all calibration modules must adhere to. This section provides a brief history of how the calibration API was developed and an overview of the applications interface.

Historical perspective

In the mid-1980s, the McIDAS-X area-format image file was redesigned to accommodate the ever increasing number of remotely sensed data types. The redesign allowed programmers to add new datasets from a variety of platforms to McIDAS-X without changing the existing software or file format. The new area file format accommodated multibanded, multibyte data along with a variety of ancillary data. The redesign process also provided a general method of storing, navigating and calibrating these data. It defined an API that all navigation and calibration modules would adhere to, and a mechanism for accessing the appropriate module at application runtime. The previous section in this chapter described the navigation system; this section describes the calibration system.

For information about the McIDAS-X area format image file, see *Chapter 6, Format of the Data Files*.

Calibration API

The calibration API is defined at several levels. For ADDE applications, the call to **mcaget** has a parameter to request data to be returned in a specific physical quantity. The calibration process occurs on the data server, and although calibration information is returned, it is usually not needed by the ADDE client application.

The next level down in the API is used by non-ADDE applications and by the ADDE data servers. A call to the **araopt** function, which sets the area options, will use a calibration module if the physical quantity specified in its UNIT option is different than that stored in the image file (word 53 of the area directory). Subsequent calls to **redara**, which reads the data, will use the calibration module if needed. This provides the application with calibrated data in a rather transparent and data-independent way.

Below **araopt** in the API is the subroutine **kbprep**, which provides the interface to the dynamic calibration modules. Most applications don't call the lower level API functions directly. However, you must understand this lowest API definition to incorporate new calibration modules.

The function names used to access the calibration are listed below:

- **kb1ini, kb1cal, kb1opt**
- **kb2ini, kb2cal, kb2opt**
- **kb3ini, kb3cal, kb3opt**

These are not the names of the functions within the calibration module itself. A mapping is done in **kbprep** that allows applications to use multiple instances of different calibrations. **kbprep** builds the name of the module to load, based on the slot number and data type.

The numeric (1, 2 or 3) is the *slot number*, which allows simultaneous use of up to three different calibration modules. The data type, which is stored in word 52 of the area directory, is needed to construct the name of the module to call. For example, the data type for GOES-8 is GVAR. Thus, the name constructed, using slot number 1, is KB1GVAR, which uses the functions KB1INIGVAR, KB1CALGVAR, and KB1OPTGVAR.

Designing your calibration module

All calibration modules have the same framework. They must conform to the McIDAS-X 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 it's working with; the calibration interface provides a means to acquire certain aspects that are common to all.

Naming your calibration module

When naming your calibration module, use this form: **`kbx`***type*`.dlm`, where *type* is the same four characters in word 52 of the area directory. For example, *type* could be GVAR, TIRO or AAA.

Conforming to 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 these three functions:

Function	Description
kbxini	initializes and verifies the requested calibration
kbxcal	calibrates the data
kbxopt	provides additional operations, which are usually queries from the application

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. Current McIDAS-X calibration modules requiring the calibration block read the block from disk from within **kbxcal**. A few modules have the capability to receive the calibration block through **kbxopt**, **kbxgvar** for example.

All the functions should return either the requested data or an error status. They should never terminate, but rather rely on the exception handling of the application.

Storing parameters

Most McIDAS-X calibration modules retrieve necessary parameters from one of these locations:

- Directory block
- Calibration block
- Line prefix
- McIDAS-X disk file

Some calibration algorithms, such as VISR (GOES 1-byte data) don't require additional parameters, 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 through a lookup table coded in the module.

The calibration of these images are handled in the code without an external data structure.

The calibration parameters stored in the area directory, calibration block, line prefix and disk files are described below.

Directory block

The MSAT (Meteosat) calibration parameters are stored in the area directory of an area format image file. Although the entire image only has three constant parameters, they change twice per day. SSEC does not recommend that you store parameters in the area directory, since it lacks available words. A calibration block is the preferred location.

Calibration block

The calibration block contains the calibration parameters for:

- AAA (GOES-7 Mode AAA)
- QTIR (Quick AVHRR)
- PRD (Product)
- GVAR (GOES-8, etc.)

This block consists of 128 words and is the most common location for storing calibration parameters. The values in this block are used for the entire image file. Sections in the calibration block are defined for AAA and GVAR.

Normally, integers or scaled integers are stored so that moving the data to different platforms is not a problem. However, potential problems exist when storing character data. For example, when accessing files that are not native to the platform, byte flipping may be needed for integers but not 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.

For more information about the calibration blocks for AAA and GVAR, see the section titled *Image-specific characteristics* in the area file description in *Chapter 6, Format of the Data Files*.

Line prefix

The prefix part of the data line contains the calibration parameters for:

- VAS (GOES-5 Sounder)
- AAA (GOES Mode AAA)
- TIRO (AVHRR)

This is the preferred location for image data, where 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. Although you can define the documentation and calibration sections for specific data, use only the calibration section to store calibration parameters. The documentation section is not guaranteed to move with some copy commands.

For more information about the line prefix, see the section titled *Data block* in the area file description in *Chapter 6, Format of the Data Files* .

McIDAS-X disk files

Disk files, which are described earlier in this chapter, are used to store calibration parameters for:

- GMS (Japanese satellite)
- VAS

The use of these files is decreasing as the definition of the calibration block is now less restricted. Previously, there was a 128-word limit in the calibration block; thus, storing large lookup tables in a disk file was preferable to including them as DATA statements in the code.

For VAS calibration, the line-to-line variability in the calibration coefficients required that the lookup tables be pregenerated for better performance. The file VASTBLS, which accounts for all possible lookup tables, is over six 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 was restricted to 128 words, a disk file was chosen to store the calibration information. For GMS-5, the tables are provided in the satellite data stream and are stored in a now-expanded calibration block. The disk file is not needed in this case, and will not be used for any future GMS satellites.

Generating lookup tables

Most calibration modules in McIDAS-X generate lookup 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-X image frame is about 300,000 pixels. Since most input data is eight or 10 bits per value, performing 256 or 1024 computations to create a lookup table is much more efficient than doing 300,000. These are many byte and word manipulation routing in McIDAS that can help to simplify the code in calibration modules. These can be found in the Conversion Utilities section of Chapter 4.

The table below describes some of the data-specific functions you will use in your calibration modules. These functions pass values through a lookup table for performance and memory considerations.

Function	Description
maaatb	AAA-specific mpixtb
mavhtb	AVHRR-specific mpixtb
mvastb	VAS-specific mpixtb
mgvatb	GVAR-specific mpixtb

For example, AAA data is sent by the satellite as 10-bit data, but is stored on disk as 15 bits. Rather than generating a lookup 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 lookup table.

For more information on conversion utilities, see the section titled *Conversion utilities* in *Chapter 4, McIDAS-X Utilities*.

Writing your calibration module

The sample calibration program named **KBXSIN.DLM**, which follows, illustrates the structure of a calibration module. It accepts input data, from 0 to 255, and returns values modified by a sine curve. To use this module with existing 1-byte data, run the following McIDAS-X command:

Type: **IMGCHA** *dataset* **STYPE** = **SIN CTYPE** = **RAW**

After compiling **KBXSIN.DLM** and any appropriate applications, McIDAS-X applications can be run against the data.

The three required functions below are present in the sample code. An additional subroutine, **maktab**, generates the lookup table.

- **kbxini**, lines 1-72
- **kbxcal**, lines 74-129
- **kbxopt**, lines 132-205

kbxini takes its input, usually from **araopt**, and copies it to a local buffer, as shown in line 53. It then verifies that the calibration requested is valid; see lines 59 and 60.

kbxcal is usually not called directly from the application, but rather when required by **redara**. It takes as input:

- Line prefix
- Area directory
- Number of values to calibrate
- Band, if required
- Buffer containing the data

The calibrated data is returned through the same buffer. **kbxcal** checks if the lookup table was generated (line 118). If not, a call is made to **maktab (** line 119). **mpixtb** completes the calibration (line 125) by taking the data, passing it through the lookup table, and expanding or packing the bytes.

kbxopt contains additional operations for querying information about the calibration. The KEYS option, shown in lines 182-187, passes a frame directory block to the calibration module; the number and list of physical quantities are returned.

This option was written for the McIDAS-X IMGPROBE command, which lists the stored data values converted to appropriate quantities. The ADDE image directory servers actually use this option to pass the information block to the client application. Because the information returned by KEYS was incomplete, the option INFO (lines 191-202) was added to provide scale factors and units. The input for INFO is:

- band number
- sensor source number
- calibration type

Most calibration modules contain code to handle stretch tables generated by the SU command. By calling **kbxopt** with BRKP as the option and the name of the stretch table, the calibration module computes a modified brightness value based on the table.

To identify the sections of code where this is done, find the CALTYP variable, which is held in COMMON/BRKPNT. In ADDE, this function is done in the client application instead of the calibration module. This code will be removed from the calibration modules in the future.

Sample program

The sample calibration module, **KBXSIN.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: *$
0 - success 26: *$ -1 - unit conversion not possible 27: *$ 28: *$ Remarks: 29: *$ This calibration module will only accept values
from 0 to 30: *$ 255 and will return values modified by a sine curve. There 31: *$ is no 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 47: C--- Store
information needed in other functions 48: C 49: COMMON/M0SIN/JTYPE,ISOU,IDES,JOPT 50: C 51: C--- Copy what araopt sent in 52: C 53:
CALL MOVW(NUMAREAOPTIONS,IOPT,JOPT) 54: 55: JTYPE=0 56: ISOU=IOPT(1) ! length in bytes of input data 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 62: C-
-- If not one of the 2 cases above is true, error 63: C 64: IF(JTYPE.EQ.0) GO TO 900 65: 66: KBXINI=0 67: RETURN 68: 69: 900
CONTINUE 70: KBXINI = -1 71: RETURN 72: END 73: 74: INTEGER FUNCTION KBXCAL(PREFIX,IDIR,NVAL,IBAND,IBUF) 75: *$ Name: 76: *$ kbxcal
- Calibrate data 77: *$ 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: 104: INCLUDE
'areaparm.inc' 105: 106: INTEGER JTYPE 107: INTEGER ISOU 108: INTEGER IDES 109: INTEGER JOPT(NUMAREAOPTIONS) 110: 111: INTEGER
ITAB(256) 112: 113: COMMON/M0SIN/JTYPE,ISOU,IDES,JOPT 114: DATA IFLAG/0/ 115: C 116: C--- If the calibration type changes, remake
the lookup table 117: C 118: IF( JTYPE .NE. IFLAG) THEN 119: CALL MAKTAB(JTYPE, ITAB) 120: IFLAG = JTYPE 121: ENDIF 122: C 123: C---
Pass the data IBUF through the lookup table ITAB 124: C 125: CALL MPIXTB(NVAL,ISOU,IDES,IBUF,ITAB) 126: 127: KBXCAL=0 128: RETURN
129: END 130: 131: 132: INTEGER FUNCTION KBXOPT(CFUNC,IIN,IOUT) 133: *$ Name: 134: *$ kbxopt - Additional operations 135: *$ 136: *$
Interface: 137: *$ integer function 138: *$ kbxopt( character*4 cfunc, integer iin(*), integer iout(*)) 139: *$ 140: *$ Input: 141:
*$ cfunc - function ('INFO', 'KEYS') 142: *$ iin - for cfunc 'KEYS', iin contains frame directory block 143: *$ for cfunc 'INFO'
144: *$ iin(1) - band number 145: *$ iin(2) - sensor source number 146: *$ iin(3) - calibration type ('GVAR', for example) 147: *$
148: *$ Input and Output: 149: *$ none 150: *$ 151: *$ Output: 152: *$ iout - for cfunc 'KEYS' 153: *$ iout(1) - number of physical
quantities,'TEMP' etc. 154: *$ iout(2-n) - list of physical quantities 155: *$ iout - for cfunc 'INFO' 156: *$ iout(1) - number of
physical quantities,'TEMP' etc. 157: *$ iout(2-n) - list of physical quantities, units, 158: *$ and scale factors 159: *$ 160: *$
Return values: 161: *$ 0 - success 162: *$ 1 - invalid function 163: *$ 164: *$ Categories: 165: *$ calibration 166: 167:
CHARACTER*4 CFUNC 168: 169: INTEGER IIN(*) 170: INTEGER IOUT(*) 171: 172: INCLUDE 'areaparm.inc' 173: 174: INTEGER JTYPE 175:
INTEGER ISOU 176: INTEGER IDES 177: INTEGER JOPT(NUMAREAOPTIONS) 178: COMMON/M0SIN/JTYPE,ISOU,IDES,JOPT 179: C 180: C--- KEYS option
181: C 182: IF( CFUNC .EQ. 'KEYS') THEN 183: IOUT(1) = 3 ! Number of types 184: IOUT(2) = LIT('RAW ') ! Physical quantities 185:
IOUT(3) = LIT('SIN ') 186: IOUT(4) = LIT('BRIT') 187: ENDIF 188: C 189: C--- INFO option 190: C 191: IF( CFUNC .EQ. 'INFO') THEN
192: IOUT(1) = 3 ! Number of types 193: IOUT(2) = LIT('RAW ') ! Physical quantities 194: IOUT(3) = LIT('SIN ') 195: IOUT(4) =
LIT('BRIT') 196: IOUT(5) = LIT(' ') ! Units 197: IOUT(6) = LIT('none') 198: IOUT(7) = LIT(' ') 199: IOUT(8) = 1 ! Scale factors 200:
IOUT(9) = 1000 201: IOUT(10)= 1 202: ENDIF 203: 204: RETURN 205: END 206: 207: 208: SUBROUTINE MAKTAB(JTYPE, ITAB) 209: *$ Name:
210: *$ maktab - Make lookup table for sine modified calibration 211: *$ 212: *$ Interface: 213: *$ subroutine 214: *$ maktab(
integer jtype, integer itab(*)) 215: *$ 216: *$ Input: 217: *$ jtype - calibration type 218: *$ 1 - sine 219: *$ 2 - grayscale 220:
*$ 221: *$ Input and Output: 222: *$ none 223: *$ 224: *$ Output: 225: *$ itab - lookup table of 256 values 226: *$ 227: *$ Remarks:
228: *$ This routine makes a lookup table by computing the sine 229: *$ for all possible values from 0 to 255 (the range of the 230:
*$ input data). Rather than computing the sine directly on 231: *$ the values 0 to 255, it is initially scaled to 0 to 10 232: *$
which is approximately 3 sine waves (3 * PI = 10) 233: *$ 234: *$ Categories: 235: *$ calibration 236: 237: INTEGER ITAB(*) 238:
239: REAL SINVAL 240: REAL X 241: C 242: C--- 243: 244: DO 100 I = 1, 256 245: 246: X = I - 1 ! X goes from 0 to 255 247: X = 10. *
(X / 255.) ! Normalize and scale to 3 sine waves 248: SINVAL = SIN(X) 249: C 250: C--- Output sine value 251: C 252: IF (JTYPE .EQ.
1) THEN 253: ITAB(I) = NINT(SINVAL * 1000.) ! Scale sine by 1000 254: C 255: C--- Output grayscale value 256: C 257: ELSE IF (JTYPE
.EQ. 2) THEN 258: ITAB(I) = NINT( 127. + 128 * SINVAL) ! Scale to 0 to 255 259: ENDIF 260: 261: 100 CONTINUE 262: 263: 264: RETURN
265: END
```

Chapter 6

Format of the Data Files

This chapter describes the formats of the data files developed for applications running under McIDAS-X. The data files are presented alphabetically with the following information:

- Name of the data file and a description of its contents
- Relevant facts about the file, such as how it's organized and if it's an ASCII or binary file
- Word allocation for each part of the file
- API library functions used with the file

This chapter describes these file formats:

File	Description
AREAnnnn	Area (image) files
DC*	MD schema definition files
*.ET	Enhancement save files
FRAMED	Panel configuration file
FRAMENH.001	Frame enhancement table
FRAMEn.p	Image frame files
GMSCAL	GMS calibration file
GRIDnnnn	Grid files
*.GRX	Graphics save tables for McIDAS-X
HIRSCRPF	HIRS calibration reference parameters
HIRSTAUL	HIRS transmittance coefficients
MDXXnnnn	MD files
MSUSCRPF	MSU calibration reference parameters
OUTL*	Base map files
SATANNOT	Image annotation description file
SATBAND	Image annotation description file
SKEDFILE	Scheduled McIDAS commands
UC	McIDAS User Common
VASTBLS	VAS calibration tables
VIRTnnnn	Virtual graphics files

AREA*nnnn*

Area (image) files, where nnnn is a user-defined number.

In McIDAS-X, images are stored in binary files called *areas*. Each area file is a collection of information that defines the image and its associated ancillary data.

Area files are usually named AREA*nnnn*, where *nnnn* is a four-digit number between 0000 and 9999. This number is called the *area file number*. For example, AREA0013 is the name of the file containing area 13.

Areas do not have to follow this standard naming convention. The file masking option of DSSERVE may be used to access a data file of any name through the ADDE.

Area files consist of these six blocks:

- Directory block
- Navigation block (NAV block)
- Calibration block (CAL block)
- Supplemental block
- Data block
- Comment block (AUDIT block)

Some blocks also contain satellite-specific information, which is located in the *Image-specific characteristics* section. For area file Application Program Interfaces (APIs), refer to the *API functions* list.

For more information about reading, writing and deleting image data, see the section titled *Image data* in *Chapter 5, Accessing Data*.

Directory block

The first 64 words of an area file contain the directory block for the image. The directory lists ancillary information about the image, such as the number of lines and data points per line, the satellite ID and the number of spectral bands. The data in the directory is stored as 32-bit (4-byte) twos-complement binary integers or as ASCII characters.

Each of the directory's 64 words is described below. Since some of the words are satellite-specific, see the section titled *Image-specific characteristics* that follows. All byte offsets and pointers are zero-based. Note that all data shown as *yyyddd* are the year and day-of-year. The *yyy* values are the actual year modulo 1900.

Word	Description
1	an integer value of zero
2	image type (currently=4)
3	SSEC sensor source number; see the Appendices
4	nominal year and Julian day of the image, *yyyddd*
5	nominal time of the image, *hhmmss*
6	upper-left image line coordinate
7	upper-left image element coordinate
8	reserved
9	number of lines in the image
10	number of data points per line
11	number of bytes per data point
12	line resolution
13	element resolution

14	number of spectral bands
15	length of the line prefix
16	SSEC project number used when creating the file
17	year and Julian day the file was created, *yyyddd*
18	file creation time, *hhmmss*
19	spectral band map for bands 1-32
20	spectral band map for bands 33-64 if word 14 is greater than 32; otherwise this value is satellite-specific
21 - 24	reserved for sensor-specific data
25 - 32	memo field; 32 ASCII characters
33	reserved
34	byte offset to the start of the data block
35	byte offset to the start of the navigation block
36	validity code
37 - 44	PDL (Program Data Load); used for pre-GOES-8 satellites
45	source of band 8; used for GOES AA processing
46	actual image start year and Julian day, *yyyddd*
47	actual image start time, *hhmmss*; in milliseconds for POES data
48	actual image start scan
49	length of the prefix documentation
50	length of the prefix calibration
51	length of the prefix band list
52	source type; satellite-specific (ASCII)
53	calibration type; satellite-specific (ASCII)
54 - 56	reserved
57	original source type
58	units
59	scaling
60	byte offset to the supplemental block
61	number of entries in the supplemental block
62	reserved

63	byte offset to the start of the calibration block
64	number of comment cards

For more information about image data, see the section titled *Image data* in *Chapter 5, Accessing Data*. For more information about coordinate systems, see the section titled *Coordinate systems* in *Chapter 2, Learning the Basics*.

Navigation block

The navigation (NAV) block contains the information for determining the location of data points in physical space. Word 35 of the directory block contains the byte offset to the start of the navigation block. If an image isn't navigated, word 35 is zero. The NAV block format varies with each satellite; see the section titled *Image-specific characteristics* that follows.

Calibration block

The calibration (CAL) block contains information for converting image data from its stored (internal) units to more meaningful units such as radiance or albedo. The presence of this block depends on the implementation of the satellite-specific calibration. Word 63 of the directory block contains the byte offset to the start of the calibration block. If there is no CAL block, word 63 is zero. The calibration block format varies with each satellite; see the section titled *Image-specific characteristics* that follows.

Supplemental (or auxiliary) block

The supplemental (or auxiliary) block contains additional information that is specific to a data type. For example, information specific to radar data is stored in this block. Also, the latitude/longitude grid for the LALO navigation is stored in this block. Word 60 of the directory block contains the byte offset to the start of the supplemental block. Word 61 contains the total number of entries in the supplemental block. If there is no supplemental block, words 60 and 61 are zero.

Data block

The data block contains the actual image data values. Any data point in an image or image sector can be located with image and file coordinates.

An area file may be produced from an image by sampling or averaging the data. In the case of multiband images, the file may include only a portion of the measured spectral bands, so that each element contains fewer data values than are contained in the original image. To map an area back to the original image, these two formulas are used:

*Image Line = UpperLeftLine + (File Line * LineResolution)*
*Image Element = UpperLeftEle + (File Element * ElementResolution)*

UpperLeftLine is the line coordinate of the first image line and *UpperLeftEle* is the element coordinate of the first image element. *File Line* and *File Element* are zero-based.

When *LineResolution* and *ElementResolution* are both 1, the image resolution is 1, or full resolution. If the image resolution is 4, every fourth line and element of an image originally at resolution 1 are included in the image. Each sensor has its own scan resolution, so an image resolution of one will mean different geographic resolutions from one satellite to another.

Each line is divided into two parts: the *line prefix* and the actual *data values* as shown below. The line prefix contains information about the image and the particular line.

```
line prefix 1 line data 1 line prefix 2 line data 2 etc.
|_____________|___________|_____________|___________|___...
0 byte numbers increase >>
```

Although the size and content of the line prefix depend on the image source defined in word 52 of the directory block, each line in an image has the same prefix length. Word 15 of the directory block contains the length of the line prefix, in bytes. If no line prefix exists, word 15 is zero.

The line prefix may contain any region shown in the diagram below and described in the following table; the regions' lengths are multiples of four bytes.

```
validity code documentation calibration band list
|_____________|_____________|_____________|___________|
0 byte numbers increase >>
```

Region	Description
validity code	Verifies the existence of the data portion of the image line. It is a constant value within each image and is stored in word 36 of the directory block. Comparing the value in the directory with the value of the validity code determines if data exists for an image line. If a line of data is missing, the corresponding place in the data file is either filled with zeros or flagged using non-matching validity codes.
documentation	Holds the documentation specific to each satellite. Word 49 of the directory block defines the length of the prefix documentation.
calibration	Holds the calibration coefficients for the data; needed when coefficients vary between image lines. Word 50 of the directory block defines the length of the prefix calibration.
band list	Contains an ordered list of the spectral bands comprising the data portion of the image line. Word 51 of the directory block defines the length of the prefix band list. Each band number is stored in a byte; thus, the range is 1 to 255.

Word 34 of the directory block contains the byte offset to the start of the data block. Each line in an image is the same length and a multiple of four bytes. To calculate the length of a line prefix, the line data, or the entire data block, use the formulas below.

line prefix length = *doc* + *cal* + *band* + 4 (if *valcode* is present)
line data length = *nbands*nele*nbytes*
line length = *line prefix length* + *line data length*
data block length = *nlines* * *line length*

The parameters used in these formulas are defined in the directory block and provided in the table below.

Parameter	Directory block word	Definition
valcode	36	length of the prefix validity code; if nonzero, the length is four bytes; otherwise it is zero
doc	49	length of the prefix documentation
cal	50	length of the prefix calibration
band	51	length of the prefix band list
nbands	14	number of bands per line
nele	10	number of data points per line
nbytes	11	number of bytes per band
nlines	9	number of lines in the image

Comment block

An area file may contain a comment (AUDIT) block containing a variety of textual information such as a list of commands run on the image object to date. Each comment record is 80 ASCII characters. Word 64 of the directory block contains the number of comment records, or cards.

Image-specific characteristics

Some aspects of McIDAS-X area files are specific to their image type. This section describes characteristics specific to the following types:

Although the descriptions for each image type vary, most include information about the directory, data, navigation and calibration blocks.

Meteosat PDUS

Meteosat PDUS images are remapped and calibrated at the ground station before the stretched signal is disseminated. This simplifies the navigation and calibration data sections. All data is eight bits; each band is stored in a separate file.

Meteosat PDUS directory block

Word	Value	Description
14	1	each band is stored separately
19	0 128 512	for visible image band map (eighth bit from right) for IR band map (tenth bit from right) for WV band map
22	.xxxxx	MIEC absolute calibration band value (IR or WV) from the calibration section of the Meteosat header; stored as scaled integer xxxxx
23	xx.x	space count corresponding to the calibration value from the calibration section of the Meteosat header; stored as scaled integer xxx
24	1 or 2	physical sensor number from the Meteosat header
37		line offset of the southeast corner of the area in image coordinates; 16-bit value from the Meteosat header, right justified plus 1
38		element offset of the southeast corner of the area in image coordinates; 16-bit value from the Meteosat header, right justified plus 1
39		satellite center longitude of rectification; 16-bit value from the Meteosat header and right justified
44	0	
49	24	length of the data block line prefix documentation, in bytes
50	0	length of the data block line prefix calibration, in bytes
51	0	length of the data block line prefix band list, in bytes
52	MSAT	image source type; 4 bytes ASCII
53	RAW	calibration type; 4 bytes ASCII
54	0 1	data was ingested as sent (full resolution) data was sampled down (every other pixel); VIS is sent as resolution 1 in some images and resolution 2 in others
55		bitmap indicating types of data in the original image; bits are numbered right to left (least to most significant bit): bit 0: 1 if VIS is included in transmission; 0 if not bit 1: 1 if IR is included in transmission; 0 if not bit 2: 1 if WV is included in transmission; 0 if not all other bits = 0

Meteosat PDUS data block

The line prefix for the data block contains the information below. For the line data, each value is transmitted as eight bits and is stored west to east and north to south in the area, the opposite of how it is transmitted.

Region	Description
validity code	this is present for the real-time data

documentation	24 bytes; this is a copy of the label that arrives with each subframe
calibration	0 bytes (not used)
band list	0 bytes (not used) since each band is stored in a separate area

Meteosat PDUS navigation block

A PDUS navigation block is divided into 256 words.

Word	Value	Description
1	MSAT	navigation type
2		Julian day of the navigation, *yyddd*
3		time of the navigation, *hhmmss*
4	0	reference position for the telescope
5	0	line number of the telescope reference position
6	1250	center scan line
7		center longitude of rectification (west positive), *ddmmss*
8 - 9	0	reserved
10		Julian day of the navigation, *yyddd*
11 - 256	0	reserved

Meteosat PDUS calibration block

No calibration block is needed for PDUS; the calibration information is stored in the directory block.

Meteosat Second Generation (MSG) Level 1.5

Meteosat Second Generation (MSG) images are remapped and calibrated at the ground station before the stretched signal is disseminated. This simplifies the navigation and calibration data sections. All data is sixteen bits; each band is stored in a separate file.

MSG directory block

<table>
<tr><th>Word</th><th>Value</th><th>Description</th></tr>
<tr><td>3</td><td>51</td><td>MSG-1</td></tr>
<tr><td>14</td><td>11
1</td><td>Number of Bands
bands 1-11 are present
band 12 is present</td></tr>
<tr><td>19</td><td colspan="2">Band bit map

<table>
<tr><th>Bit</th><th>Band</th><th>Wavelength (μm)</th><th>Description</th></tr>
<tr><td>0</td><td>1</td><td>0.6</td><td>Visible cloud and surface features</td></tr>
<tr><td>1</td><td>2</td><td>0.8</td><td>Visible aerosols over water, vegetation</td></tr>
<tr><td>2</td><td>3</td><td>1.6</td><td>Near-Infrared Surface, cloud phase</td></tr>
<tr><td>3</td><td>4</td><td>3.9</td><td>Infrared low-level cloud/fog, fire detection</td></tr>
<tr><td>4</td><td>5</td><td>6.2</td><td>Infrared upper-level water vapor</td></tr>
<tr><td>5</td><td>6</td><td>7.3</td><td>Infrared mid-level water vapor</td></tr>
<tr><td>6</td><td>7</td><td>8.7</td><td>Infrared total water, cloud phase, dust</td></tr>
<tr><td>7</td><td>8</td><td>9.7</td><td>Ozone</td></tr>
<tr><td>8</td><td>9</td><td>10.8</td><td>Infrared surface/cloud-top temperature</td></tr>
<tr><td>9</td><td>10</td><td>12.0</td><td>Infrared surface/cloud temperature, low-level water vapor</td></tr>
<tr><td>10</td><td>11</td><td>13.4</td><td>Infrared CO_2; cloud heights</td></tr>
<tr><td>11</td><td>12</td><td>0.7</td><td>Broadband visible (0.4 - 1.1 μm)</td></tr>
</table>

Bands 1-11 are low resolution images (3 km) with a center latitude/longitude point of (0,0).

Band 12 is a high resolution visible image (1 km) which has a southern and northern data block offset from each other. The center latitude/longitude point is (0,2).</td></tr>
<tr><td>52</td><td>MSG</td><td>image source type; 4 bytes ASCII</td></tr>
<tr><td>53</td><td>RAW</td><td>calibration type; 4 bytes ASCII</td></tr>
</table>

MSG data block

For the line data, each value is transmitted as sixteen bits and is stored east to west and south to north in the original files.

MSG navigation block

An MSG navigation block is divided into 128 words.

Word	Value	Description
1	MSG	navigation type
2 - 128	0	reserved

MSG calibration block

An MSG calibration block is divided into 128 words.

Word	Value	Description

1	*	central wave number #1 (IR channels 3-11)
2	*	central wave number #2 (IR channels 3-11)
3	*	gain (channels 1-3,12)
4	*	offset (channels 1-3,12)
5-128	0	reserved

* values stored as 4 ASCII F10.6 characters separated by a space character.

GVAR Imager

The GVAR Imager provides two types of data:

- Supplemental data
- Sensor data

The supplemental data is provided in Block 0, which is the GVAR Imager documentation block. Block 0 has its own directory block, validity code, documentation region and data. The GVAR Imager sensor data likewise has its own directory, data, navigation and calibration blocks. Both types of data are described below.

The OGE tables referenced in this section are from *Operations Ground Equipment, Internal Specification, DRL 504-02-1 Part 1, Specification No E007020*, released February 9, 1994, Space Systems/Loral, 3825 Fabian Way, Palo Alto, California 94303-4604. That document describes data which is formatted by the ground station and then retransmitted.

The band information for the GVAR Imager is provided in *Appendix B, Satellite Information*.

Block 0

The GVAR Imager documentation, Block 0, contains additional control information about an image. Some of this information is also contained in the imager sensor data. For each line of GVAR Imager sensor data transmitted, one line of Block 0 documentation is transmitted and stored in a separate area. Below are the values specific to the Block 0 directory.

Word	Value	Description
12	8	line resolution
13	1	element resolution
14	1	number of bands
19	1	band map
25 - 32	RT IMGR DOC	normal entry; ASCII
49	44	length of the data block line prefix documentation, in bytes
50	0	length of the data block line prefix calibration, in bytes
51	0	length of the data block line prefix band list, in bytes
52	GVAR	image source type; 4 bytes ASCII
53	RAW	calibration type; 4 bytes ASCII

The line prefix's validity code for Block 0 is four bytes. Its documentation region is 44 bytes, consisting of the following:

Documentation region	Bytes	OGE table
block header CRC	2	3-5
scan status	4	3-6
year, day and time from Block 0	8	3-6
block header	30	3-5

The rest of the Block 0 line contains 8040 bytes of 8-bit data. See OGE Table 3-6.

Bytes 17-24 contain the time the block was sent from the ground station.

GVAR Imager directory block

The GOES-8 through GOES-12 satellites have two instruments: an imager and a sounder. Areas with SSEC-assigned, even-numbered sensor sources provide imager data, while odd-numbered sensor sources provide sounder data. Word 52 of the directory block contains the image source type (GVAR) for 2-byte GVAR data as it is ingested. Word 53 contains the units that data is stored in; RAW for 2-byte raw GVAR data.

Word 14 of the directory block contains the number of spectral bands present in an image. The filter band map in word 19 of the directory block describes the bands in an area. A bit is set for each band appearing in the area. The number of bands must match the value in word 14. The values specific to the sensor data's directory block are shown below.

Word	Value	Description
11	2	number of bytes per band
19	1 for VIS 2, 4, 8 or 16 for IR	band map; one bit should be set for each band in the area
25 - 32	RT IMGR IR RT IMGR VIS RT IMGR	normal entry; ASCII visible band multibanded
49	228	length, in bytes, of the data block line prefix documentation if single band (last two bytes are zero); otherwise a higher value
50	0	length, in bytes, of the data block line prefix calibration
51	0	length, in bytes, of the data block line prefix band list
52	GVAR	image source type; 4 bytes ASCII
53	RAW	calibration type; 4 bytes ASCII
55	1	

GVAR Imager data block

The GVAR Imager produces observational data for a spatial location in five spectral bands: one visible (VIS) and four infrared (IR). An image contains only one of these five bands. Word 19 in the directory block contains a band filter map indicating the area file's band.

The highest resolution for a visible image is one. It is four for an IR image, since longer wavelengths have less resolution. For a GVAR satellite, resolution one means approximately 1 km resolution at the satellite subpoint.

Each element in a GOES-8 image contains one 10-bit pixel representing raw data from the instrument. Each pixel is stored as two bytes in the McIDAS-X area file. The hardware shifts the data so the 10 bits are formatted as shown below. The x's are the data bits; the rest is 0-filled after shifting.

| 0 | x | x | x | x | x | x | x | x | x | x | 0 | 0 | 0 | 0 | 0 |

The sensor data's line prefix contains a 4-byte validity code and a 76-byte documentation region consisting of the following:

Documentation region	Bytes	OGE table
stray light correction status; see OGE Table 3-6, bytes 259 - 260	2	3-6
scan status	4	3-6
year, day and time from Block 0	8	
block header	30	3-5
line documentation consisting of sixteen pairs of 10-bit fields, right justified; each 10-bit field can be obtained with a LOGICAL AND against 03FF	32	3-7
block zero record, first 150 words	150	3-6

The block header and line documentation blocks are included for every band in the image line.

The rest of the line consists of up to 41920 bytes of data. Since it is 2-byte data, half that many pixels are represented.

Bytes 17-24 contain the time the block was sent from the ground station.

GVAR Imager navigation block

The GVAR Imager navigation block contains 640 words. Unless otherwise noted, words are twos-complement binary integers. This navigation information comes from Block 0 records. Bytes designated R*4 in OGE Tables are in Gould format. They must be scaled and converted to integers or converted to Real on the machine doing the decoding, scaled as designated below, and then converted to integer.

Word	Value	Description
1	GVAR	navigation type; 4 bytes ASCII
2		ASCII string, usually a letter followed by three integers
3		imager scan status; bits 0-15 are right justified, with 15 the least significant; IMC active flag is bit 8, counting from the least significant bit; 1=active; see OGE Table 3-6, bytes 3-6
4		imager scan status; bits 16-31 are right justified, with 31 the least significant; yaw-flip processing enabled flag is bit 16, counting from the least significant bit; 1=enabled; see OGE Table 3-6, bytes 3-6
5	0	reserved
6 - 62 6 7 8 9 10 11 12 13 - 14 15 16 17 18 19 - 31 32 - 42 43 - 51 52 - 60 61 62		see OGE Table 3-6, bytes 295 - 522 reference longitude, rad*10000000 reference distance from nominal, km*10000000 reference latitude, rad*10000000 reference yaw, rad*10000000 reference attitude roll, rad*10000000 reference attitude pitch, rad*10000000 reference attitude yaw, rad*10000000 epoch date/time, BCD format delta from epoch time, minutes*100 image motion compensation roll, rad*10000000 image motion compensation pitch, rad*10000000 image motion compensation yaw, rad*10000000 longitude delta from reference values, rad*10000000 radial distance delta from reference values, km*10000000 sine of the geocentric latitude delta values, units*10000000 sine of the orbit yaw delta values, units*10000000 daily solar rate, rad/min*10000000 exponential start time from epoch, minutes*100
63 - 117 63 64 65 66 67 68 ... 95 96 97 98 99 100 101 102		roll attitude angle (OGE Table 3-6, bytes 523-742) exponential magnitude, rad*10000000 exponential time constant, minutes*100 mean attitude angle, rad*10000000 number of sinusoids/angles, no units magnitude of first order sinusoid, rad*10000000 phase angle of first order sinusoid, rad*10000000 ... magnitude of fifteenth sinusoid, rad*10000000 phase angle of fifteenth sinusoid, rad*10000000 number of monomial sinusoid, no units order of applicable sinusoid, no units order of first monomial sinusoid, no units magnitude of monomial sinusoid, rad*10000000 phase angle of monomial sinusoid, rad*10000000 angle from epoch at daily solar rate, rad*10000000 repeat of words 98-102 for second monomial

103-107		repeat of words 98-102 for third monomial repeat of words 98-102 for fourth monomial
108-112	MORE	reserved 4 bytes ASCII
113-117	GVAR	4 bytes ASCII
118-127		
128		
129		
130-239		attitude angles repeat of Words 63-117 for pitch attitude angle; see OGE Table 3-6, bytes 743-962
130-184		repeat of Words 63-117 for yaw attitude angle; see OGE Table 3-6, bytes 963-1182
185-239	MORE	reserved 4 bytes ASCII
240-255	GVAR	4 bytes ASCII
256		
257		
258-367		misalignment angles repeat of Words 63-117 for roll misalignment angle; see OGE Table 3-6, bytes 1183-1402
258-312		repeat of Words 63-117 for pitch misalignment angle; see OGE Table 3-6, bytes 1403-1622
313-367		
368		year and Julian day, *yyyddd*
369		nominal start time of the image; comes from Block 0 when navigation data is taken from Block 0, HHMMSSmmm
370	1	imager/sounder instrument flag; 1=imager, 2=sounder
371-379		reserved
380		instrument nadir, north/south cycles; see OGE Table 3-6, byte 6305
381		instrument nadir, east/west cycles; see OGE Table 3-6, byte 6306
382		instrument nadir, north/south increments; see OGE Table 3-6, byte 6307-6308
383		instrument nadir, east/west increments; see OGE Table 3-6, byte 6309-6310
384	MORE	4 bytes ASCII
385	GVAR	4 bytes ASCII
386-511		reserved
512	MORE	4 bytes ASCII
513	GVAR	4 bytes ASCII
514-640		reserved

GVAR Imager calibration block

The imager calibration block is made up of 128 words (512 bytes), as shown in the table below. The data is in the Gould format.

Word	Value	Description
1 - 8		visible bias coefficients; one per detector (OGE Table 3-6, bytes 6399-6430)
9 - 16		visible first order gain coefficients; one per detector (OGE Table 3-6, bytes 6431-6462)
17 - 24		visible second order gain coefficients; one per detector (OGE Table 3-6, bytes 6463-6494)
25		visible radiance to albedo conversion factor (OGE Table 3-6, bytes 6495-6498)
26 - 29		det side 1 IR bias scaling factors; one per IR channel (OGE Table 3-6: bytes 6667-6670 Ch 4, Side 1; bytes 6675-6679 Ch 5, Side 1; bytes 6683-6686 Ch 2, Side 1; bytes 6691-6694 Ch 3, Side 1)
30 - 33		det side 2 IR bias scaling factors; one per IR channel (OGE Table 3-6: bytes 6695-6698 Ch 4, Side 2; bytes 6703-6706 Ch 5, Side 2; bytes 6711-6714 Ch 2, Side 2; bytes 6719-6722 Ch 3, Side 2)
34 - 37		det side 1 IR gain scaling factors; one per IR channel (OGE Table 3-6: bytes 6723-6726 Ch 4, Side 1; bytes 6731-6734 Ch 5, Side 1; bytes 6739-6742 Ch 2, Side 1; bytes 6747-6750 Ch 3, Side 1)
38 - 41		det side 2 IR gain scaling factors; one per IR channel (OGE Table 3-6: bytes 6751-6753 Ch 4, Side 2; bytes 6759-6762 Ch 5, Side 2;bytes 6767-6770 Ch 2, Side 2; bytes 6775-6778 Ch 3, Side 2)
42-128	0	

GVAR Block 11

The GVAR Block 11 holding areas contain data for sounder images. This data is not easily accessed. A decoder must reformat the raw Block 11 data and place it in the sounder image, where it is available for analysis and display.

GVAR Block 11 directory block

Word	Value	Description
11	1 or 2	number of bytes per element, depending on the element size of the band; a holding area cannot contain both 1- and 2-byte data
14	1	number of bands in the image
19		block type filter; positions of set bits correspond to the block types requested; the least significant bit is the rightmost bit; the value 787968 translates to 0c0600 hex with bits set in positions 20, 19, 11 and 10
25-32	RT BK11 BYT1	normal entry; ASCII
49	40	length of the data block line prefix documentation, in bytes
50	0	length of the data block line prefix calibration, in bytes
51	0	length of the data block line prefix band list, in bytes
52	BK11	image source type; 4 bytes ASCII
53	RAW	calibration type; 4 bytes ASCII

GVAR Block 11 data block

GVAR transmits 22 types of Block 11 data. This can be 6-, 8- or 10-bit data. The user can specify any type to be stored in a single holding area. Control fields in the line prefix or the first portion of the data (called the SAD ID) are used by postprocesses, such as the sounder decoder, to determine the block type. Each data block line consists of a single Block 11 sector, or *block*. All blocks are 8040 bytes. Refer to the OGE, sections 3.3.7 - 3.3.7.14 for a description of the contents of each block type.

The 10-bit data is formatted as follows, with x representing a data bit and the rest being zero-filled after shifting.

| 0 | x | x | x | x | x | x | x | x | x | 0 | 0 | 0 | 0 | 0 |

The 8-bit data is formatted as follows:

| x | x | x | x | x | x | x | x |

The 6-bit data is formatted as follows:

| 0 | 0 | x | x | x | x | x | x |

The line prefix for Block 11 contains a 4-byte validity code. Its documentation region is 40 bytes, consisting of the following:

Documentation region	Bytes	OGE table
stray light correction status; see OGE Table 3-6, bytes 259 - 260	2	3-6
year, day and time from Block 0	8	3-6
block header	30	3-5

The rest of the Block 11 line consists of up to 8040 bytes of data, depending on block type.

Bytes 17-24 contain the time the block was sent from the ground station.

GVAR Sounder

GVAR Sounder areas are decoded from Block 11 data. The GVAR Sounder decoder reads the Block 11 holding areas, which contain blocks of type 32 (20 hex), type 35 (23 hex) and others. These blocks are documented in OGE, sections 3.3.7.2 and 3.3.7.3.

Navigation and calibration data is read from type 32 blocks, which are sounder documentation blocks. Sensor data is read from type 35 blocks, which are sounder scan data blocks. After the sensor data is read, it is reformatted and placed in the sounder image area.

The band information for the GVAR Sounder is provided in *Appendix B, Satellite Information*.

GVAR Sounder directory block

Word	Value	Description
12	10	line resolution; if lines are sampled or averaged, the resolution is a multiple of 10
13	10	element resolution; if pixels are sampled or averaged, the resolution is in multiples of 10
14	19	number of bands
19	524287	band filter map; translates to 0007ffff hex; a bit is set for each of bands 1 - 19
25-32	Priority Completed	normal entry; ASCII
49	36 or 216	length of the data block line prefix documentation, in bytes; if 216, the first 180 words of Sounder Auxiliary Data block have been have been added
50	0	length of the data block line prefix calibration, in bytes
51	up to 24 (in multiples of four)	length of the data block line prefix band list, in bytes; see the section below for more information
52	GVAR	image source type; 4 bytes ASCII
53	RAW	calibration type; 4 bytes ASCII

GVAR Sounder data block

The GVAR Sounder produces data for a given spatial location in 18 IR spectral bands and one visible band. The number of bands in the sounder data must match the value in word 14 of the directory block. Word 19 designates the band map. All sounder data fields are 13 bits placed in 2-byte (16-bit) fields. Each data point in a scan data block has 23 bands of data. The data points correspond to a geographic area 11 pixels west-east and 4 pixels north-south. For each image line, the decoder produces 11 sets of 23 interleaved fields of data.

Bands 20-23 of this data are not displayable; they hold the latitude and longitude of the first 19 bands. The latitude and longitudes are 32-bit values. Since the actual sounder data is 16 bits, the latitude and longitude values must be split in half to store them in the area structure.

Band 20 holds the two most significant bytes of the latitude; band 21 holds the two least significant bytes. Band 22 holds the two most significant bytes of the longitude; band 23 holds the two least significant bytes.

If the latitude and longitude values are not requested, the band list section will contain 20 bytes total with 19 bytes used.

These latitude and longitude values are in the Gould floating point format. See OGE, section 3.5.4. For example, if the latitude of a data point is 100.1640625, the hex representation is 42642A00; band 20 holds 4264, and band 21 holds 2A00.

This package does not provide any code for using these latitudes and longitudes; they are included only for reference purposes.

The four sounder detectors are numbered 1, 2, 3, and 4. Each data block line contains information from only one detector. The documentation block contains a field indicating which detector was used for the current line.

The line prefix contains a 4-byte validity code and a documentation region, which is described in the table below. The remainder of the line consists of the interleaved sounder data.

Documentation region	Bytes	OGE table
first nine words of the Sounder Auxiliary Data Block ID	9	3-8
year, day and time of the scan line start	8	3-11
scan status	2	3-11
number of blocks in the scan	2	3-11
O&A location (not used)	2	
detector status	10	3-11
number of the detector used for this line	1	
padding	2	
If the value in word 49 of the area directory is 216, the first 180 words of the Sounder Auxiliary Data block have been inserted here.	180	3-11

GVAR Sounder navigation block

Navigation blocks contain 640 words. Unless otherwise noted, words are twos complement binary integers. Navigation information comes from Block 11 records, type 32. Bytes designated R*4 in the OGE Tables are in Gould format in the holding areas. They must be scaled and then converted to integer, or converted to Real on the machine doing the decoding, scaled as designated below, and then converted to integer.

If the latitude and longitude values are not requested, the band list section will contain 20 bytes total with 19 bytes used.

Because the word allocation information for the sounder navigation block is nearly identical to that for the imager, it is not repeated here. Only the words with a different description are shown below. That difference is usually the OGE table number and/or byte numbers.

Word	Value	Description
3		scan status; bits 0-15 are right justified, bit 15 is the least significant bit; the IMC active flag is bit 8, counting from the least significant bit; 1=active (OGE Table 3-11, bytes 31-32)
4		yaw-flip flag; bits 0-15 are right justified, bit 15 is the least significant bit; the yaw-flip processing enabled flag is bit 16, counting from the least significant bit; 1=enabled (OGE Table 3-10, bytes 57-58)
6 - 62		(OGE Table 3-11, bytes 323 - 550)
63 - 117		roll attitude angle (OGE Table 3-11, bytes 551-770)
130-184		repeat of Words 63-117 for pitch attitude angle (OGE Table 3-11, bytes 771-990)
185-239		repeat of Words 63-117 for yaw attitude angle (OGE Table 3-11, bytes 991-1210)
258-312		repeat of Words 63-117 for roll misalignment angle (OGE Table 3-11, bytes 1211-1430)
313-367		repeat of Words 63-117 for pitch misalignment angle (OGE Table 3-11, bytes 1431-1650)
370		imager/sounder instrument flag; 1=imager, 2=sounder
380		instrument nadir, north/south cycles (OGE Table 3-6, byte 3005)
381		instrument nadir, east/west cycles (OGE Table 3-6, byte 3006)

382		instrument nadir, north/south increments (OGE Table 3-6, bytes 3007-3008)
383		instrument nadir, east/west increments (OGE Table 3-6, bytes 3009-3010)

GVAR Sounder calibration block

Word	Value	Description
1 - 4		visible bias coefficients; one per detector (OGE Table 3-11, bytes 3075-3090)
5 - 8		visible first order gain coefficients; one per detector (OGE Table 3-11, bytes 3091-3106)
9 - 12		visible second order gain coefficients; one per detector (OGE Table 3-11, bytes 3107-3122)
13		visible radiance to albedo conversion factor (OGE Table 3-11, bytes 3123-3126)
14 - 31		IR bias scaling factors; one per IR channel (OGE Table 3-11, bytes 3991-4278); all channels contain the same values for each detector
32 - 49		IR gain scaling factors; one per IR channel (OGE Table 3-11, bytes 4279-4566); all channels contain the same values for each detector
50-128	0	

GOES VISSR

The image source type VISR is historic in origin, going back to the early GOES satellites. At that time, data was 1-byte for both the visible and IR. Later satellites, such as GOES-6 and -7, were equipped with IR sensors that returned 10-bit values stored as two bytes. For data storage and transfer reasons, commands such as IMGCOPY can convert the 2-byte data to 1-byte. This conversion to 1-byte data preserves the temperature information for the IR channels. It is valid for GVAR, POES, Meteosat and GMS satellite data.

The image source type VISR is from word 52 of the directory block. If the area contains IR data, the temperature may be calculated from the pixel value using the formulas below, where *T* is the brightness temperature (degrees K) and *B* is the pixel value (0 to 255). For IR data, the highest pixel values correspond to the coldest temperatures.

$T = 418 - B$ where $B > 176$ or $B = 176$
$T = 330 - (B / 2)$ where $B < 176$ or $B = 176$

The line prefix in a VISR area may be absent or it may contain only the 4-byte validity code.

The band information for GOES VISSR is provided in *Appendix B, Satellite Information*.

GOES-7

GOES-7 produced data in two different modes:

- Mode AA
- Mode AAA

Most GOES-7 data after 24 March 1987 (Julian day 87083) is AAA. This section documents the mode AAA for the IR and VAS instruments. Word 52 of the area directory contains the source type AAA.

The VAS senses the atmosphere for a given spatial location in up to 12 different IR spectral bands and one visible band. All or some of the IR bands may be included in a single VAS type area. The visible, however, may be contained in a separate area of VISR type. As a result, it may require two areas to contain the total information transmitted by the satellite during a given time period.

GOES-7 AAA directory block

Word	Value	Description
14		number of spectral bands
19		band filter map; a bit is set to one for each band appearing in the area
52	AAA	image source type; 4 bytes ASCII
53	RAW	calibration type; 4 bytes ASCII

GOES-7 AAA data block

The line prefix consists of a 4-byte validity code, 512 bytes of IR common documentation, and 116 bytes of VAS calibration information organized as follows.

Bytes	Description
4	day, *yyddd*
4	time of the scan, *hhmmss*
4	scan number (satellite coordinate line number)
104	13 eight-byte groups (1 per possible band) each containing: channel number - 2-byte binary integer (see the table on the next page) number of spins - 2-byte binary integer unused - 4 bytes

The line prefix also contains 4, 8, 12 or 16 bytes of band list information; one byte for each band plus up to three bytes to round to the nearest whole word.

The structure of a VAS area is complicated by two facts:

- Every line may not contain all the spectral bands indicated in the band map (word 19 in the directory block).
- The order of the bands may not be the same on every line.

What does appear on a given line is indicated in the band list section, which acts as an index for the line. Only the leftmost *n* bytes of the band list contain nonzero data, with *n* being the actual number of bands contained in each element of the line. The *I* th byte of the band list corresponds to the *I* th 16-bit pixel in each element of the line. Unused band list bytes are filled with binary zeros; the data in the unused pixel locations may not be zero, but in any case should be ignored.

The channel numbers range from 1 to 38; channel 39 exists but has never been put into service. Each is described in the table below.

Channel	Detector	Size	Location	Spectral band
1	HGCDTE	large	upper	1
2	HGCDTE	large	upper	2
3	HGCDTE	large	upper	3

4	HGCDTE	large	upper	4
5	HGCDTE	large	upper	5
6	INSB	large	upper	6
7	HGCDTE	large	upper	7
8	HGCDTE	large	upper	8
9	HGCDTE	large	upper	9
10	HGCDTE	large	upper	10
11	INSB	large	upper	11
12	INSB	large	upper	12
13	HGCDTE	large	lower	1
14	HGCDTE	large	lower	2
15	HGCDTE	large	lower	3
16	HGCDTE	large	lower	4
17	HGCDTE	large	lower	5
18	INSB	large	lower	6
19	HGCDTE	large	lower	7
20	HGCDTE	large	lower	8
21	HGCDTE	large	lower	9
22	HGCDTE	large	lower	10
23	INSB	large	lower	11
24	INSB	large	lower	12
25	HGCDTE	small	upper	3
26	HGCDTE	small	upper	4
27	HGCDTE	small	upper	5
28	HGCDTE	small	upper	7
29	HGCDTE	small	upper	8
30	HGCDTE	small	upper	9
31	HGCDTE	small	upper	10
32	HGCDTE	small	lower	3
33	HGCDTE	small	lower	4
34	HGCDTE	small	lower	5
35	HGCDTE	small	lower	7

36	HGCDTE	small	lower	8
37	HGCDTE	small	lower	9
38	HGCDTE	small	lower	10

For a given spectral band, only one detector size is used in an area. However, two channels representing different positions of the detector for a particular band may appear in a single area although they may not appear on the same line. For example, channels 8 and 20 may appear in the same area, but not channels 8 and 36.

GOES-7 AAA navigation block

Unless otherwise noted, the words in the GOES-7 navigation block are twos-complement binary integers.

Word	Value	Description
1	GOES	navigation type (ASCII characters)
2		satellite ID, year, and Julian day, *ssyyddd*
3		nominal start time of the image, *hhmmss*
4 - 12 4 5 6 7 8 9 10 11 12	 1	orbit parameters orbit type epoch date (ETIMY), *yymmdd* epoch time (ETIMH), *hhmmss* semimajor axis (SEMIMA), km * 100 orbital eccentricity (ECCEN) * 1000000 (unitless) orbital inclination (ORBINC), deg * 1000 mean anomaly (MEANA), deg * 1000 argument of perigee (PERIGEE), deg * 1000 right ascension of ascending node (ASNODE), deg * 1000
13 - 15 13 14 15		attitude parameters declination of satellite axis (DECLIN), *dddmmss* (+=north) right ascension of satellite axis (RASCEN), *dddmmss* picture center line number (PICLIN)
16		spin period (SPINP); the satellite period, in microseconds, or the spin rate in revolutions/minute
17 - 20 17 18 19 20		frame geometry total sweep angle, line direction (DEGLIN), *dddmmss* number of scan lines (LINTOT), nnllllll where nn is the number of sensors and llllll is the number of scans; total number of lines is nn * llllll total sweep angle, element direction (DEGELE), *dddmmss* number of elements in a scan line (ELETOT)
21 - 30 21 22 23 24 25 26 27 28 29 30	 0	camera geometry forward-leaning (PITCH), *dddmmss* sideways-leaning (YAW), *dddmmss* rotation (ROLL), *dddmmss* reserved east/west adjustment (IAJUST) in visible elements (+ or -) time computed by IAJUST from the first valid landmark of the day (IAJTIM), *hhmmss* reserved angle between VISSR and sun sensor (ISEANG), *dddmmss* reserved for later implementation of *SKEW* reserved
31 - 38 31		betas for this area scan line of the first beta time of the first beta scan line (beginning), *hhmmss*

32 33 34 35 36 37 38		time of the first beta scan line (continued), milliseconds*10 beta count 1 scan line of the second beta time of the second beta scan line (beginning), *hhmmss* time of the second beta scan line (cont.), milliseconds*10 beta count 2
39- 128 39 40 41- 120 121- 128		gammas for this area gamma, element offset * 100; this is the nominal offset at time zero of this day gamma-dot, element drift per hour * 100 reserved memo; up to 32 ASCII characters of comments

GOES-7 AAA calibration block

The calibration block is composed of the following data.

Word	Value	Description
1		sensor source number
2		date, *yyddd*
3		time, *hhmmss*
4 - 9		radiance equation coefficients, array IAB(2,38)
80 - 117		radiance equation coefficients scale factors, IFAB(38)
118-128	0	

Transforming the VAS raw IR values into brightness temperatures is accomplished via the intermediate computation of calibrated VAS radiances. The array IAB contains two coefficients for each of the 38 channels; IFAB contains one scale factor for each channel.

If the channel is ICHAN, compute the radiance for the raw value *P* using:

AB1 = *IAB* (1,ICHAN)
AB2 = *IAB* (2,ICHAN)
FAB = 2.**(15-*IFAB* (ICHAN))
R = (*AB2* * *P* /32. - *AB1*) / *FAB*

The raw value *P* is divided by 32 because the data is stored as 15-bit numbers, but the coefficients expect 10-bit numbers.

AVHRR

The AVHRR (Advanced Very High Resolution Radiometer) instrument is a 5-channel scanning radiometer. It generates data in HRPT, LAC and GAC modes.

- HRPT (High Resolution Picture Transmission) is real-time, 1 km resolution, direct readout data. It is confined to areas where the satellite is in range of a ground receiving station.
- LAC (Local Area Coverage) is 1 km resolution data recorded onboard the satellite and transferred to the ground station at a later time.
- GAC (Global Area Coverage) is 4 km resolution data derived from 1 km data. An on-board processor averages four of five data points along every third scan line and stores the data for transmission.

The band information for the AVHRR sensor is provided in *Appendix B, Satellite Information*.

AVHRR directory block

Word	Value	Description
14	5	all five bands are normally stored in one file
15	220 or 244 or 364	line prefix length in bytes (AVH3=220, TIRO=244, AVHR=364)
19		band map for the image, which has valid bits 1 through 6 for the new AVHRR/3 instruments on NOAA-15 (Note that if bit 3 is turned on, bit 6 must be turned on as well, and vice-versa, since both sensors share a single data band identified as either 3 or 6); determining which line belongs to which sensor can be made only from reading the line prefixes, except when an entire area contains only band 3 or only band 6
26		POES signal type (HRPT, LAC, GAC)
36		line prefix validity (VAL) code (length=4)
47		time in milliseconds
49	0 or 192	length of the line prefix documentation (AVH3=0, AVHR and TIRO=192)
50	40 or 160 or 212	line prefix calibration length (TIRO=40, AVHR=160, AVH3=212)
51	8	length of the level (LEV) map section in the line prefix
52	TIRO/AVHR	image source type
53	RAW	pixel type stored in the area file
54	0, 1	sampling/averaging indicator (0=average, 1=sample)
55	1, 2, 3	signal type indicator (1=LAC, 2=GAC, 3=HRPT)
56		orbit position (ascending node, decending node, equatorial pass)

AVHRR navigation block

The AVHRR navigation block is divided into 128 words.

Word	Value	Description
1	TIRO	navigation type; 4 bytes ASCII
2		sensor source, year and Julian day of the navigation, *ssyyddd*
3		time of the navigation, *hhmmss*

4	1	orbit type
5		epoch date, *yymmdd*
6		epoch time, *hhmmss*
7		semi-major axis, km * 100
8		orbital eccentricity, *1000000
9		orbital inclination, degrees * 1000
10		mean anomaly, degrees * 1000
11	0	argument of perigee, degrees * 1000
12		right ascension of the ascending node, degrees * 1000
13	2048	number of samples per line
14		angular increment between samples, degrees * 1000
15		fraction of a second in epoch time
16 - 45		reserved
46	-1 1	satellite is in a descending pass satellite is in an ascending pass
47		image coordinates of the first line to navigate
48		time at the start of the first line, milliseconds from start-of-day
49		time interval between lines, in milliseconds
50	0 1	image is displayed normally image is inverted
51		number of lines in the inverted image
52		number of elements in the inverted image
53		time interval between lines, in microseconds (preferred over word 49)
54		time interval between individual data points * 100000000
55-120		reserved
121-128		comments; up to 32 characters

AVHRR Calibration

The calibration block is not used. Instead, calibration is done dynamically on a line-by-line basis to accommodate changing orbital conditions in the AVHRR/3 instrument. Therefore, the calibration data is contained in the data line prefixes. Refer to the*TIRO and AVHR line prefix* descriptions below, and *Appendix D*, *POES AVHRR Calibration Information* for additional information.

Because NOAA-12 and -14 AVHRR use the older TIRO calibration while the NOAA-15 AVHRR uses the newer AVHR calibration, changes have been made in the McIDAS-X area structure between the NOAA-14 areas and the NOAA-15 areas.

AVHRR data block

The AVHRR data block consists of data lines, each consisting of a line prefix and line data. The TIRO and AVHR line prefixes are different.

TIRO line prefix

The TIRO line prefix for the data block contains the information below.

Region	Description
validity code	this is present for the realtime data
documentation	192-byte DOC section from the signal transmission
calibration	40 bytes of zeros; filled during post-processing in McIDAS
level	8 bytes; values 1 through 5 (left to right), with three pad zeros in successive bytes, indicate the order the bands will appear in each pixel in the subsequent data section

The TIRO line prefix is defined as follows:

Byte Address	Section	Description
0	VAL	Validity code
4	DOC	Words 7-102 of HRPT minor frame (all left shifted 5 bits into a 2-byte sample)
196	CAL	Band 1 Slope/Gain 1 (all slopes and intercepts in the CAL block are scaled by 1000)
200		Band 1 Intercept/Offset 1
204		Band 1 Slope/Gain 2
208		Band 1 Intercept/Offset 2
212		Band 2 Slope/Gain 1
216		Band 2 Intercept/Offset 1
220		Band 2 Slope/Gain 2
224		Band 2 Intercept/Offset 2
228		Band 3 Slope/Gain 1
232		Band 3 Intercept/Offset 1
236		Band 3 Space scan five sample average (rounded and left shifted 5 bits)
240		Band 3 Internal target temperature (scaled by 100)
244		Band 4 Slope/Gain 1
248		Band 4 Intercept/Offset 1
252		Band 4 Space scan five sample average (rounded and left shifted 5 bits)
256		Band 4 Internal target temperature (scaled by 100)
260		Band 5 Slope/Gain 1
264		Band 5 Intercept/Offset 1
268		Band 5 Space scan five sample average (rounded and left shifted 5 bits)
272		Band 5 Internal target temperature (scaled by 100)

276	LEV	Band number: 01
277		Band number: 02
278		Band number: 03 or 06
279		Band number: 04
280		Band number: 05
281		Unused Band numbers: 00 00 00 (three bytes)

AVHR line prefix

The AVHR line prefix for the data block contains the information below.

Region	Description
validity code	this is present for the realtime data
documentation	192-byte DOC section from the signal transmission
calibration	160 bytes from the signal transmission
level	8 bytes; values 1 through 5 (left to right), with three pad zeros in successive bytes, indicate the order the bands will appear in each pixel in the subsequent data section

The AVHR line prefix for the data block is defined as folllows.

Section	Byte Address	Size (bytes)	Contents
VAL	0	4	Validity code
DOC	4	192	Words 7-102 of HRPT minor frame (all left shifted 5 bits into a 2-byte sample)
CAL	196	4	Band 1 Slope/Gain 1 (all slopes and intercepts in the CAL section are scaled by 100,000 except for the band 3 slope)
	200	4	Band 1 Intercept/Offset 1
	204	4	Band 1 Slope/Gain 2
	208	4	Band 1 Intercept/Offset 2
	212	16	Unused
	228	4	Band 2 Slope/Gain 1
	232	4	Band 2 Intercept/Offset 1
	236	4	Band 2 Slope/Gain 2
	240	4	Band 2 Intercept/Offset 2
	244	16	Unused
	260	4	Band 6 Slope/Gain 1
	264	4	Band 6 Intercept/Offset 1
	268	4	Band 6 Slope/Gain 2

	272	4	Band 6 Intercept/Offset 2
	276	16	Unused
...or...			
	260	4	Band 3 Slope/Gain (scaled by 10,000,000)
	264	4	Band 3 Intercept/Offset (scaled by 100,000)
	268	8	Unused
	276	4	Band 3 Space scan five sample average (rounded and left shifted 5 bits)
	280	4	Band 3 Target scan five sample average (rounded and left shifted 5 bits)
	284	4	Band 3 Internal target temperature (scaled by 100)
	288	4	Unused
	292	4	Band 4 Slope/Gain (scaled by 100,000)
	296	4	Band 4 Intercept/Offset
	300	8	Unused
	308	4	Band 4 Space scan five sample average (rounded and left shifted 5 bits)
	312	4	Band 4 Target scan five sample average (rounded and left shifted 5 bits)
	316	4	Band 4 Internal target temperature (scaled by 100)
	320	4	Unused
	324	4	Band 5 Slope/Gain (scaled by 100,000)
	328	4	Band 5 Intercept/Offset
	332	8	Unused
	340	4	Band 5 Space scan five sample average (rounded and left shifted 5 bits)
	344	4	Band 5 Target scan five sample average (rounded and left shifted 5 bits)
	348	4	Band 5 Internal target temperature (scaled by 100)
	352	4	Unused
LEV	356	1	Band number: 01
	357	1	Band number: 02
	358	1	Band number: 03 or 06
	359	1	Band number: 04
	360	1	Band number: 05
	361	3	Unused Band numbers: 00

The AVH3 calibration block is defined as follows.

Section	Byte Address	Description (pre-KLM data)	Description (KLM data)

CAL	0	* Slope, Band 1	** Slope #1, Band 1
	4	Intercept, Band 1	Intercept #1, Band 1
	8	0	Slope #2, Band 1
	12	0	Intercept #2, Band 1
	16	Slope, Band 2	Slope #1, Band 2
	20	Intercept, Band 2	Intercept #1, Band 2
	24	0	Slope #2, Band 2
	28	0	Intercept #2, Band 2
	32	0	Slope #1, Band 3a
	36	0	Intercept #1, Band 3a
	40	0	Slope #2, Band 3a
	44	0	Intercept #2, Band 3a
	48	Slope, Band 3	Slope, Band 3b
	52	Intercept, Band 3	Intercept, Band 3b
	56	0	Nonlinear Correction, Band 3b
	60	Slope, Band 4	Slope, Band 4
	64	Intercept, Band 4	Intercept, Band 4
	68	0	Nonlinear Correction, Band 4
	72	Slope, Band 5 1	Slope, Band 5
	76	Intercept, Band 5	Intercept, Band 5
	80	0	Nonlinear Correction, Band 5

* - For pre-KLM data, slope is scaled by 2^30; intercept is scaled by 2^22.

** - For KLM data, slope is scaled by 1.E+07; intercept is scaled by 1.E+06 for vis/near IR; Slope/Intercept/Non-linear terms are scaled by 1.E+06 for IR.

For a complete description of the AVHRR instruments on the satellites and their calibration, see the *NOAA-KLM User's Guide* at *http://www2.ncdc.noaa.gov/docs/klm* or *http://www2.ncdc.noaa.gov/docs/intro.htm*. For instruments prior to 1988, see *NOAA Technical Memorandum NESS 107*, 1988 *(now obsolete)* or http://noaasis.noaa.gov/NOAASIS/ml/calibration.html.

AVHRR line data

AVHRR data is transmitted as 10 bits and stored as 16 bits. The 10-bit data is formatted as follows, with x representing a data bit and the rest being zero-filled after shifting.

| 0 | x | x | x | x | x | x | x | x | x | 0 | 0 | 0 | 0 | 0 |

The 16-bit values from each of the five channels covering the same geographic area are stored interleaved in one image. The lines of data are stored in a time-ordered sequence and as the satellite scans right to left.

TIP data

TIP (TIROS-N Information Processor) data is extracted from the AVHRR, GAC and LAC data described in the previous section. TIP data also contains information received from the MSU (Microwave Sounding Unit), the HIRS (High Resolution Infrared Sounder), and the SSU (Stratospheric Sounding Unit). These three sources measure incoming radiation in the infrared and microwave spectrum.

The navigation block for TIP data is filled with zeros; no calibration information is needed.

TIP directory block

Word	Value	Description
14	1	
19	1	band map value
49	196	length of the line prefix documentation
50	0	length of the line prefix calibration
51	0	line prefix band list
52	TIRO	image source type; 4 bytes ASCII
53	RAW	calibration type; 4 bytes ASCII

TIP data block

TIP data is transmitted as 10 bits and stored as 16 bits. The 10-bit data is formatted as follows, with x representing a data bit and the rest being zero-filled after shifting:

| 0 | x | x | x | x | x | x | x | x | x | 0 | 0 | 0 | 0 | 0 |

The line prefix for the data block contains the information below.

Region	Description
validity code	optional, but recommended to flag missing data, which requires zeros as placeholder data or a validity code that does not match the value in the area directory
documentation	196-byte DOC section from the signal transmission
calibration	0 bytes; not needed for TIP data
band list	0 bytes; not needed for TIP data

For a complete description of the line documentation fields, see *NOAA Technical Memorandum NESS 107*, 1988. The band information for TIP data is provided in this manual in *Appendix B, Satellite Information*.

GMS

The following section describes the navigation and calibration for the visible and infrared channels of the Geostationary Meteorological Satellites, GMS-4 and GMS-5. GMS-4 has one channel of visible and one channel of infrared data. GMS-5 has two channels for visible and two channels for infrared data; only one channel of each is used. The remaining two channels are reserved for backup.

- The tables referenced in this section are from the document *Revision of GMS Stretched-VISSR Data Format*, Japan Meteorological Agency, October 1993.

GMS navigation block

The following three tables list the words used in the GMS navigation block. These are 1-byte words. The first table lists all the words in the GMS navigation block and the following two tables list the words used in the attitude prediction data sub-blocks and the orbit prediction data sub-blocks.

The Type column in the tables below shows scaled integers in the format R*M.N. The R indicates real numbers, M is the number of bytes and N is the exponent.

- The GMS data is also formatted using the navigation block format of GOES-7 data. The section *GOES-7 AAA navigation block* describes the GOES-7 navigation block.

Word	Type	Description
1-6	R*6.8	observation start time (MJD)
7-10	R*4.8	VIS channel stepping angle along line (rad)
11-14	R*4.8	IR channel stepping angle along line (rad)
15-18	R*4.10	VIS channel sampling angle along pixel (rad)
19-22	R*4.10	IR channel sampling angle along pixel (rad)
23-26	R*4.4	VIS channel center line number of VISSR frame
27-30	R*4.4	IR1 channel center line number of VISSR frame
31-34	R*4.4	VIS channel center pixel number of VISSR frame
35-38	R*4.4	IR1 channel center pixel number of VISSR frame
39-42	R*4.0	number of sensors of VIS channel
43-46	R*4.0	number of sensors of IR channel
47-50	R*4.0	VIS total line number of VISSR frame
51-54	R*4.0	IR total line number of VISSR frame
55-58	R*4.0	VIS pixel number of one line
59-	R*4.0	IR pixel number of one line

62		
63-66	R*4.10	VISSR misalignment angle around x-axis (rad)
67-70	R*4.10	VISSR misalignment angle around y-axis (rad)
71-74	R*4.10	VISSR misalignment angle around z-axis (rad)
75-78 79-82 83-86 87-90 91-94 95-98 99-102 103-106 107-110	 R*4.7 R*4.10 R*4.10 R*4.10 R*4.7 R*4.10 R*4.10 R*4.10 R*4.7	These words are modified as a function of the horizon points. See words 11-14 in Table A.1 in the document *Revision of GMS Stretched-VISSR Data Format* Element of VISSR misalignment matrix: row 1 and column 1 row 2 and column 1 row 3 and column 1 row 1 and column 2 row 2 and column 2 row 3 and column 2 row 1 and column 3 row 2 and column 3 row 3 and column 3
111-114	R*4.4	IR2 channel center line of VISSR frame
115-118	R*4.4	IR3 channel center line number of VISSR frame
119-122	R*4.4	IR2 channel center pixel number of VISSR frame
123-126	R*4.4	IR3 channel center pixel number of VISSR frame
127-240		not used
241-246	R*6.8	daily mean of Satellite Spin Rate (rpm)
247-256		not used
257-617		attitude prediction data sub-blocks 1 through 10 (10 similar attitude prediction data sub-blocks are repeated). See the table below.
618-2352		orbit prediction data sub-blocks 1 through 8 (eight similar orbit prediction data sub-blocks are repeated); see the table below.

The table below describes an attitude data prediction sub-block. The GMS navigation block contains ten attitude data prediction sub-blocks that occupy words 257-617. Each block occupies 36 words.

Word	**Type**	**Description**
1-6	R*6.8	prediction time (UTC represented in MJD)
7-12	BCD*6	anno domini represented by BCD (YYMMDDHHmmSS: year, month, day, hour, minute, second)

13-18	R*6.8	angle between z-axis and satellite spin axis projected on yz-plane in mean of 1950.0 coordinates (rad)
19-24	R*6.11	angle between satellite spin axis and yz-plane in mean of 1950.0 coordinates (rad)
25-30	R*6.8	dihedral Angle between the Sun and the Earth measured clockwise seeing from North (rad)
31-36	R*6.8	spin Rate: spin speed of satellite (rpm)

The table describes the orbit prediction data sub-block. The GMS navigation block contains eight orbit prediction sub-blocks that occupy words 618-2352. Each block occupies 182 words.

Word	Type	Description
1-6	R*6.8	prediction time (UTC represented in MJD)
7-48		not used
49-54	R*6.6	x component of satellite position in Earth-fixed coordinates (m)
55-60	R*6.6	y component of satellite position in Earth-fixed coordinates (m)
61-66		not used
67-84		not used
85-90	R*6.8	Greenwich sidereal time in true of data coordinates (deg)
91-102		not used
103-108	R*6.8	right ascension from the satellite to the sun in the Earth-fixed coordinates (deg)
109-114	R*6.8	declination from the satellite to the sun in the Earth-fixed coordinates (deg)
115-128		not used
		Element of nutation and precession matrix:
129-134	R*6.12	row 1 and column 1
135-140	R*6.14	row 2 and column 1
141-146	R*6.14	row 3 and column 1
147-152	R*6.14	row 1 and column 2
153-158	R*6.12	row 2 and column 2
159-164	R*6.16	row 3 and column 2
165-170	R*6.12	row 1 and column 3
171-176	R*6.16	row 2 and column 3
177-182	R*6.12	row 3 and column 3

GMS calibration block

The GMS calibration block contains both directory and data conversion tables located in sub-blocks that follow the directory. The 128-word directory block, shown below, indicates the locations of the six sub-blocks. The starting byte offset for each sub-block varies with the data; therefore, it is shown as a variable in the directory below.

Word	Value	Description
0	GMS5	4-byte ASCII identifier
1	0x58	directory block length in bytes
2	COEF	4-byte ASCII identifier for sub-block 1
3		starting byte offset from word 0

4	4VIS or 5VIS	identifies sub-block 2
5		starting byte offset from word 0
6	4IR or 5IR1	identifies sub-block 3
7		starting byte offset from word 0
8	5IR2	identifies sub-block 4
9		starting byte offset from word 0
10	5IR3	identifies sub-block 5
11		starting byte offset from word 0
12	SPAR	identifies spares block, sub block 6
13		starting byte offset from word 0
14-21		not used
22-85		Words 1 to 256 of GMS 4-or -5 calibration block; see the table below
86-127		reserved for future tables (GMS-6, etc.)

The table below describes the calibration data sub-blocks, not all of which may be filled. The calibration data, which follows the directory, has a length of 6400 bytes.

See table A-7 in the *Revision of GMS stretched-VISSR data format* document for a complete description of the data block.

Byte	**Descriptions**
1-4	calibration information ID
5-10	data generated date (YYYYMMDDHHmm)
11	sensor selection
12-256	sub-block 1; infrared radiance calculations
257-1280	sub-block 2; visible level-to-albedo conversion tables; four 64-level tables for VIS1 through VIS4 detectors
1281-2304	sub-block 3; IR1 level-to-temperature conversion table
2305-3328	sub-block 4; IR2 level-to-temperature conversion table
3329-4352	sub-block 5; IR3 level-to-temperature conversion table
4353-6400	sub-block 6; spares

The prefixes for each scan line of visible data must contain a code indicating the detector used. The first four bytes of the documentation section should contain the following information.

Code	**Description**
00000000	not a visible band
6C6C0000	detector 1 of the visible band
B4B40000	detector 2 of the visible band

D8D80000	detector 3 of the visible band
FCFC0000	detector 4 of the visible band

DMSP

The Defense Meteorological Satellite Program (DMSP) satellites are polar orbiting satellites. DMSP has two sensors: the Operational Linescan System (OLS) and the Special Sensor for Microwave Imagery (SSM/I) data. The following sections describe the navigation and calibration blocks for each sensor.

DMSP navigation block

The table below lists the contents of the DMSP navigation block. The block contains 128 words and is used for all DMSP signal types.

Word	Value	Description
1	DMSP	navigation type; indicates the signal type being navigated
2		sensor source, year and date of the signal, *ssyyddd*
3		nominal time of image, *hhmmss*
4	1	orbit type; set to one
5		epoch date from the ASCII record below, *yyddd*
6		epoch time from the ASCII record below; days*1.e9
7		mean motion from the ASCII record below; deriv*1.e9
8		mean motion from ASCII record below; accel and mantissa
9		mean motion from the ASCII record below; accel and expon
10		bstart and mantissa; usually not used
11		b-start and expon; usually not used
12		inclination from the ASCII record below
13		right ascension of ascending node from the ASCII record below; deg*1.e6
14		eccentricity from the ASCII record below; ecc*1.e6
15		argument of perigee from the ASCII record below; per,deg*1e7
16		mean anomaly from the ASCII record below; anomaly*deg* 1.e6
17		mean motion; revs/day*1.e7
18-44		unused
45		data type; OLS, MT, MT2 or MI
46		ascending LT; 0=morning, 1=evening
47		number of first scan to navigate
48		time of first scan; sec*1.e3
49		scan flipped flag; 0=no 1=yes
50		element flipped flag; 0=no 1=yes
51		unused
52		number of elements per scan

53-120		unused
121-128		32-character ASCII record

The DMSP navigation block includes a 0-based, type 8 ASCII record, in words 120-127, that contains the scaled integer values used by the navigation block. The table below describes the ASCII record.

Character numbers	Description
3-4	spacecraft ID
6-17	epoch date; a decimal point indicates a fraction of a day
18-25	mean motion derivative
26-28	mean motion acc mantissa
28-30	mean motion acc exponent
32-39	inclination
41-48	RA of ascending mode
50-56	eccentricity
58-65	perigee
67-74	mean anomaly
76-85	mean motion

DMSP calibration

The OLS and microwave sensors use different calibration. The OLS calibration uses only one pair of of gain and offset parameters to transform infrared pixel values to radiance. The offset is set to 190.0 and the gain is set to .4706. The calibration module **`kbxols.dlm`** contains the offset pair. The visible pixels are not calibrated.

For the microwave sensors, each data line is calibrated and the information is stored in the documentation section. A postprocessor, DMSPCAL, calculates the gains and offsets from the information in the documentation section. The table below describes the calibration documentation for each line of data.

Bytes	Description
0-1	validity code
2-4	hot load temperatures; two measurements in byte 2, one measurement in byte 3
5-6	radiometer temperatures; forward radiometer temperature
7-13	instantaneous gain; one for each channel; these are calculated from the hot load temperatures and cold counts hotloadavg = average of 3 hotload temperatures hotcountavg = average of band hot counts coldcountbandavg = avg of band cold counts instantaneousbandgain = (hotloadavg-2.7k)/(hotcountbandavg-coldbandavg)
14-15	reference voltages; reference voltage and reference return both from block 6
16-20	cold voltages band 1
21-25	hot voltage band 1

26-30	cold voltages band 2
31-35	hot voltages band 2
36-40	cold voltages band 3
41-45	hot voltages band 3
46-50	cold voltages band 4
51-55	hot voltages band 4
56-60	cold voltages band 5
61-65	hot voltages band 5
66-70	cold voltages band 6
71-75	hot voltages band 6
76-80	cold voltages band 7
81-85	hot voltages band 7

When DMSPCAL is run, a gain and offset for each channel is placed in the calibration section of each scan line prefix. The table below describes the byte location of each gain and offset pair.

Bytes	**Description**
0-1	gain, band 1
2-3	offset, band 1
4-5	gain, band 2
6-7	offset, band 2
8-9	gain, band 3
10-11	offset, band 3
12-13	gain, band 4
14-15	offset, band 4
16-17	gain, band 5
18-19	offset, band 6
20-21	offset, band 6
22-23	gain, band 7
24-25	offset, band 7

Geographic Projections

The structure of the succeeding words of the following geographic projections are listed here:

- LAMB - Lambert Conformal
- MERC - Mercator
- PS - Polar stereographic
- RADR - Radar
- RECT - Rectilinear
- LALO - Latitude/Longitude

Lambert Conformal

If the type is LAMB, this is the structure of the succeeding words:

Word	Description
2	image line of the North Pole
3	image element of the North Pole
4	standard latitude 1, *DDDMMSS*
5	standard latitude 2, *DDDMMSS*
6	spacing at standard latitude, *meters*
7	normal longitude, *DDDMMSS*
8	radius of the planet, *meters*
9	eccentricity of the planet, x1000000
10	coordinate type, ≥ 0 planetodetic, < 0 planetocentric
11	longitude convention, ≥ 0 west positive, < 0 west negative
12-120	reserved
121-128	memo entry; up to 32 characters of comments

Mercator

If the type is MERC, this is the structure of the succeeding words:

Word	Description
2	image line of the equator
3	image element of the normal longitude
4	standard latitude, *DDDMMSS*; usually zero
5	spacing at standard latitude, *meters*
6	normal longitude, *DDDMMSS*
7	radius of the planet, *meters*
8	eccentricity of the planet, x1000000
9	coordinate type, ≥ 0 planetodetic, < 0 planetocentric
10	longitude convention, ≥ 0 west positive, < 0 west negative
11-120	reserved
121-128	memo entry; up to 32 characters of comments

Polar Stereographic

If the type is 'PS ' (four characters), this is the structure of the succeeding words:

Word	Description
2	image line of the North Pole
3	image element of the North Pole
4	standard latitude, *DDDMMSS*
5	spacing at standard latitude, *meters*
6	normal longitude, *DDDMMSS*
7	radius of the planet, *meters*

8	eccentricity of the planet, x1000000
9	coordinate type, ≥ 0 planetodetic, < 0 planetocentric
10	longitude convention, ≥ 0 west positive, < 0 west negative
11-120	reserved
121-128	memo entry; up to 32 characters of comments

Radar

If the type is RADR, this is the structure of the succeeding words:

Word	Description
2	row (image coordinates) of the radar site
3	column (image coordinates) of the radar site
4	latitude of the radar site, *DDDMMSS*
5	longitude of the radar site, *DDDMMSS*
6	pixel resolution, *meters*
7	rotation of north from vertical, degrees x1000
8	if present, same as 6, but only for longitude direction

Rectilinear

If the type is RECT, this is the structure of the succeeding words:

Word	Description
2	a particular image row number
3	latitude corresponding to word 2, *degrees x10000*
4	a particular image column number
5	longitude corresponding to word 4, *degrees x10000*
6	latitude degrees/image line, *degrees x10000*
7	longitude degrees/image line, *degrees x10000*
8	radius of the planet, *meters*
9	eccentricity of the planet, x1000000
10	coordinate type, ≥ 0 planetodetic, < 0 planetocentric
11	longitude convention, ≥ 0 west positive, < 0 west negative

LALO (Latitude/Longitude)

If the type is LALO, this is the structure of the succeeding words:

Word	Description
2	**SERV** (indicates that the nav is coming from a server)
3-65	reserved
66	number of rows in LALO array
67	number of elements in LALO array
68	minimum latitude as a floating point number (in real*4 format) moved to the integer space
69	minimum longitude as a floating point number (in real*4 format) moved to the integer space
70	maximum latitude as a floating point number (in real*4 format) moved to the integer space
71	maximum longitude as a floating point number (in real*4 format) moved to the integer space
72	resolution of line lalo, 1 means there is latitude for every image line
73	resolution of element lalo, 1 means there is longitude for every image element
74	**512** (size of NAV array)
75	word 74 + ((word 66)(word 67)(4))
76	line number in image coordinates of top line
77	element number in image coordinates of leftmost element
78	total size of LALO block

79	**24** (size of latitude header)
80	offset in AUX to longitudes: word 79 + ((word 66)(word 67)(4)) + 28
81	offset in directory block to latitude block: 768 + word 79 + cal_length
82	offset in directory block to longitude block: word 81 + ((word 66)(word 67)(4)) + 28
83-128	reserved

API functions

Function	Description
mcadir	opens a connection to read the directory block from an image object
mcadrd	reads the directory block from an image object
mcaget	opens a connection to read the data block from an image object
mcalin	reads the data portion of the current image line
mcapfx	reads the prefix portion of the current image line
mcanav	reads the navigation block of an image object
mcacal	reads the calibration block of an image object
mcacrd	reads the comment block of an image object
mcafree	frees the handle and memory of a connection opened by mcaget
mcaput	opens a connection to write the directory, navigation and calibration blocks of an image object
mcaout	writes the line prefix and data portions of an image line
mcacou	writes the comment block to an image object
mcadel	deletes image objects from a dataset
mcasort	gets the parameters from the command line and adds them to the selection array for a future mcaget call

DC*

MD schema definition files.

Schema definition files are ASCII text files that contain lines of text defining the structure of MD files for point-source data types. To create a schema definition file, use a standard editor to enter a set of text lines. The file serves as input to the schema registration program, SCHE, that reads the text lines and forms a blueprint of the MD file's structure. SSEC distributes the schema definition files below with every McIDAS upgrade.

Schema file	Contents
DCFO14	NGM MOS forecasts
DCIRAB	international radiosonde observations
DCISEN	isentropic data
DCISFC	international surface hourly observations
DCISHP	international ship and buoy observations
DCPIRP	PIREP, AIREP and ACARS data
DCSYN	surface synoptic observations

Because schema definition files form the input to an application, you must use the formats provided below for each text line of a schema file.

File header

A header is required for each schema definition file. Its format is shown below, followed by a table of parameter definitions.

SCHEMA *name version date id "description*

Parameter	Definition
name	four-character schema name
version	schema version number
date	Julian day the schema definition file was created
id	schema identification number
"description	comments

Row and column headers

Either a row header or a column header is required for each schema definition file. The row and column header formats are shown below, followed by a table of parameter definitions.

ROWS *nrows "description*
rkeyname1 rscale1 runits1 "description
...
rkeynameN rscaleN runitsN "description

Parameter	Definition
nrows	default number of rows to make for this MD file
rkeyname1 ... N	name of the first and Nth row header keys
rscale1 ... N	scale factor of the first and Nth row header keys

runits1 ... N	physical units of the first and Nth row header keys
"description	comments

COLUMNS *ncols "description*
ckeyname1 cscale1 cunits1 "description
...
ckeynameN cscaleN cunitsN "description

Parameter	Definition
ncols	default number of columns to make for this MD file
ckeyname1 ... N	name of the first and Nth column header keys
cscale1 ... N	scale factor of the first and Nth column header keys
cunits1 ... N	physical units of the first and Nth column header keys
"description	comments

Data lines

Use the format below for the data lines.

DATA
dkeyname1 dscale1 dunits1 "description
...
dkeynameN dscaleN dunitsN "description

Parameter	Definition
dkeyname1 ... N	name of the first and Nth data keys
dscale1 ... N	scale factor of the first and Nth data keys
dunits1 ... N	physical units of the first and Nth data keys
"description	comments

Repeat groups

Repeat groups are optional. They are useful when data of the same type is repeated within the data portion of a record. The format is shown below.

REPEAT *nrepeats*
rkeynames rscales runits "description

Parameter	Definition
nrepeats	number of repeat groups in the data portion of the schema definition
rkeynames	names of the repeated data keys
rscales	scale factors of the repeated data keys
runits	physical units of the repeated data keys
"description	comments

End-of-schema and comment formats

To end the schema definition file, use **ENDSCHEMA**. To enter comments, use the format *"comments.*

API functions

No API functions exist for reading and writing MD schema definition files.

For more information about MD file schemas and associated terminology, see the *MDXXnnnn* data structure in this chapter.

*.ET

*Enhancement save files, where * is a user-defined file name.*

Enhancement save files are binary files, each containing an 817-word (3268-byte) table. Words 1 through 768 contain the red, green and blue color intensities for each of the possible 256 brightness values. The individual intensities have a physical range of 0 to 255, where 0 is the minimum and 255 is the maximum intensity.

Word allocation

Word	Description
0	reserved for system use
1	red color intensity for an image brightness of 0
2	red color intensity for an image brightness of 1
3	red color intensity for an image brightness of 2
...	...
256	red color intensity for an image brightness of 255
257	green color intensity for an image brightness of 0
258	green color intensity for an image brightness of 1
259	green color intensity for an image brightness of 2
...	...
512	green color intensity for an image brightness of 255
513	blue color intensity for an image brightness of 0
514	blue color intensity for an image brightness of 1
515	blue color intensity for an image brightness of 2
...	...
768	blue color intensity for an image brightness of 255
769-816	reserved

API functions

Currently, no API library functions exist for reading or writing enhancement save files.

FRAMED

Panel configuration file.

The panel configuration file is a binary file that describes the layout of the panels for each image frame. If it is not present, all frames are single-paneled (i.e. full screen). There are two bytes per frame, one for the number of panels in the x-direction and one for the number of panels in the y-direction. Up to 127 panels in each axis can be defined for a single frame.

If the frame has never been paneled with the PANEL command, the two bytes contain HEX 8080. If the frame was explicitly set to 1x1 geometry, the bytes contain HEX 0101. The software considers them both to represent a single panel frame.

Byte allocation

Byte	Description
0	number of panels in the x-direction for frame 1
1	number of panels in the y-direction for frame 1
2	number of panels in the x-direction for frame 2
3	number of panels in the y-direction for frame 2
...	...

FRAMENH.001

Frame enhancement table.

The frame enhancement file is a binary file that contains an 816-word (3264-byte) table for each frame allocated for the session. To calculate the word position of a particular frame's enhancement table, use the formula below.

position = (*frame number* * 816) + 1

In each table, the first 768 words contain the red, green and blue color intensities for each of the possible 256 brightness values. The individual intensities have a physical range of 0 to 255, where 0 is the minimum and 255 is the maximum intensity.

Word allocation for each file

Word	Description
0 - 815	reserved for system use
816 - 1631	image frame 1 enhancement table
1632 - 2447	image frame 2 enhancement table
2448 - 3263	image frame 3 enhancement table
...	...

Word allocation for each table

Word	Description
0	red color intensity for an image brightness of 0
1	red color intensity for an image brightness of 1
2	red color intensity for an image brightness of 2
...	...
255	red color intensity for an image brightness of 255
256	green color intensity for an image brightness of 0
257	green color intensity for an image brightness of 1
258	green color intensity for an image brightness of 2
...	...
511	green color intensity for an image brightness of 255
512	blue color intensity for an image brightness of 0
513	blue color intensity for an image brightness of 1
514	blue color intensity for an image brightness of 2
...	...
767	blue color intensity for an image brightness of 255
768-815	reserved

API functions

Currently, no API library functions exist for reading and writing frame enhancement files.

FRAME*n.p*

Image frame files, where n is the frame number and p is the panel number.

The image frame file is a binary file that describes the contents of an image frame. For single panel (i.e. full screen) frames, *y* is zero. A single panel frame 1 uses file FRAME1.0; a 4-panel frame 2 uses files FRAME2.1 through FRAME2.4.

The file has three components:

- 64-word frame directory block
- User-defined extension block
- 640-word navigation block

The default size of the user extension is zero. The size can be modified by editing the the file **`m0panel.h`** and rebuilding McIDAS-X. No SSEC programs use the extension.

The navigation block contains the information for determining the location of the data points in physical space. The navigation block's format varies with each satellite. See the section titled *Image-specific characteristics* for image files (AREA*nnnn*).

For image frame Application Program Interfaces (APIs), refer to the *API functions* listing.

Frame directory

Word	Description
0	sensor source number; -1 means no image is currently loaded on the frame and the remaining words can be ignored; see the SATANNOT file in this chapter for a description of the sensor source numbers
1	year and Julian day of the image, *ccyyddd*
2	time of the image, *hhmmss*
3	spectral band
4	upper-left image line in satellite coordinates
5	upper-left image element in satellite coordinates
6	reserved for system use
7	image frame line containing the upper-left image line
8	image frame element containing the upper-left image element
9	value specified for lmag in IMGDISP's MAG keyword, if it's positive; 1 if lmag is negative
10	displayed line resolution factor (where 1=base resolution of the instrument) if lmag in IMGDISP's MAG kewyord is negative; line resolution factor of source image file if lmag is positive
11	displayed element resolution if emag in IMGDISP's MAG keyword is negative; element resolution factor of source image file if emag is positive
12	year and Julian day the image was created, *ccyyddd*
13	time the image was created, *hhmmss*
14	year and Julian day the image frame was created, *ccyyddd*
15	time the image frame was created, *hhmmss*
16-18	reserved
19	value specified for lmag in IMGDISP's MAG keyword

20	value specified for emag in IMGDISP's MAG keyword
21-31	reserved
32	if positive, element blowup; if negative, same as word 11
33-35	reserved
36	original source type if changed by IMGCOPY, IMGREMAP, etc.; CHARACTER
37-38	file name used by the SU application; CHARACTER
39-52	reserved
53	displayed geographic line resolution x100 if GEORES=YES is specified in IMGDISP; 0 if GEORES=YES is not specified
54	displayed geographic element resolution x100 if GEORES=YES is specified in IMGDISP; 0 if GEORES=YES is not specified
55-60	ASCII ADDE dataset name from IMGDISP
61	position number within the ADDE dataset
62-63	NEXRAD station ID (NEXRAD image only)

API functions

Fortran Function	Description
erafrm	flags a frame directory or navigation block as unused (erased)
frtonv	reads a navigation block
getfrm	reads a frame directory block
m0getusr	reads a user extension block
m0putusr	writes a user extension block
nvtofr	writes a navigation block
putfrm	writes a frame directory block

GMSCAL

GMS calibration file.

GMSCAL is a binary file that contains calibration data for GMS VIS (visible) and IR (infrared) sensor data transmitted in the GMS Stretched-VISSR real-time signal.

VIS data is 6-bit. Calibration is achieved with a 64-value VIS level-albedo lookup table. This calibration table is interpolated to make a 256-value table. IR data is 8-bit. Calibration is achieved with a 256-value IR level- temperature lookup table, which may change with the spacecraft.

The GMS calibration file is supplied with McIDAS-X.

Word allocation

Word	Description
0	number of IR calibration tables in the file
1	identification number of the first IR table in the file
2	identification number of the second IR table in the file
511	identification number of the 511th IR table, if present
512-767	interpolated VIS calibration table; albedos are multiplied by 10**6 and stored as integers
1024-1279	first IR calibration table in the file; it corresponds to the identifier stored in word 1; temperatures are multiplied by 10**3 and stored as integers
1280-1535	second IR calibration table in the file; it corresponds to the identifier stored in word 2 and is only present if the calibration table changes with the spacecraft

API functions

Currently, no API library functions exist for reading and writing GMSCAL.

GRID*nnnn*

Grid files, where nnnn is a user-defined number.

A grid has two components:

- 64-word header describing the grid projection, grid directory, gridded variable, level parameter, time of the data, etc.
- Actual grid data, which is an *n*- by *m*-word matrix

A grid file is a binary file, which may contain a user-defined maximum number of grids. By default, a grid file is created with the ability to store 159 grids unless otherwise specified.

Grid file numbers can be between 1 and 999999. If a grid file number is five or six digits, the file name begins with only GRI or GR. For example, grid file number 12345 has the file name GRI12345, but grid file number 123456 has the file name GR123456.

Grid files do not have to follow this standard naming convention. The file masking option of DSSERVE may be used to access a data file of any name through the ADDE.

The word allocation for grid files is divided into several sections below. The grid file directory is described first, followed by the grid header. The first 33 words and words 40 through 64 of the grid header are the same for all grid types; however, words 34 to 39 are specific to a particular grid type.

For grid file Applications Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

Grid file directory

Word	Description
0 - 7	32 characters of label information
8	reserved
9	date the file was created, *yyyddd*
10	maximum number of grids (n) in the grid file
10+1	word offset, from the beginning of the grid file, where grid 1 starts; if the offset is -1, no grids exist
...	...
10+n	word offset for grid n
10+ n+1	next available address to start writing the next grid

Grid header

Each grid header contains 64 words. The offset of the first word in the header is defined by the word offset in Words 10 +1 through 10 + *n* in the grid file, where *n* is the grid number.

Header Word	Description
1	total size; rows * columns (not to exceed the value of MAXGRIDPT in gridparm.inc)
2	number of rows
3	number of columns
4	Julian date of the data, *yyyddd*
5	time of the data, *hhmmss*
6	forecast time for the grid, if applicable

7	name of the gridded variable, four character ASCII
8	scale of the gridded variable, specified as a power of 10
9	units of the gridded variable, four character ASCII
10	value of the vertical level 1013 = 'MSL' 999 = ' ' 0 = 'TRO' 1001 = 'SFC' (Otherwise, it is displayed as entered.)
11	scale of the vertical level
12	unit of the vertical level
13	gridded variable type: 1 = time difference 2 = time average 4 = level difference 8 = level average (or any sum of 1, 2, 4 and 8)
14	used if the grid parameter is a time difference or time average, *hhmmss*
15	used if the grid parameter is a level difference or level average; values are the same as Word 9
16 - 32	reserved
33	grid origin; identifies the type of program that generated the grid data
34	grid projection type: 1 = pseudo-Mercator 2 = polar stereographic or Lambert conformal 3 = equidistant 4 = pseudo-Mercator (more general) 5 = no navigation 6 = tangent cone
35 - 40	varies, depending on the grid type; see the GRID*nnnn* data file in Chapter 6 for more information
41 - 52	reserved; filled only if the grid was created by the McIDAS-XCD GRIB decoder
49	geographic grib number
50	parameter grib number
51	model grib number
52	level grib number
53 - 64	grid description

Remaining header words if grid type = 1 or 4 (pseudo-Mercator)

Header Word	Description
35	maximum latitude of the grid, *degrees*10000*
36	maximum longitude of the grid, *degrees*10000*
37	minimum latitude of the grid, *degrees*10000*

38	minimum longitude of the grid, *degrees*10000*
If TYPE=1:	
39	increment between grid points; same in x/y directions
40	reserved
If TYPE=4:	
39	increment between the grid points (latitude)
40	increment between the grid points (longitude)

Remaining header words if grid type = 2 (polar stereographic or Lambert conformal)

Header Word	Description
35	row number of the North Pole*10000
36	column number of the North Pole*10000
37	column spacing at standard latitude, meters
38	longitude parallel to columns, *degrees*10000*
39 - 40	standard latitudes, *degrees*10000*; set these two equal for polar stereographic

Remaining header words if grid type = 3 (equidistant)

Header Word	Description
35	latitude of (1,1), *degrees*10000*
36	longitude of (1,1), *degrees*10000*
37	clockwise rotation of column 1 relative to north, *degrees*10000*
38	column spacing, in meters
39	row spacing, in meters

Remaining header words if grid type = 5 (no navigation)

Header Word	Description
35 - 40	reserved

Remaining header words if grid type = 6 (tangent cone)

Header Word	Description
35	row number of the North Pole*10000
36	column number of the North Pole*10000
37	column spacing at standard latitude, meters
38	longitude parallel to columns, *degrees*10000*
39	standard latitude, *degrees*10000*
40	reserved

Reserved header words if grid was created by the McIDAS-XCD GRIB decoder

For more information, refer to the National Centers for Environmental Prediction Office Note 388: GRIB, Edition 1.

Header Word	Description
47	date grid was filed, *yyyddd*
48	time grid was filed, *hhmmss*
49	original geographic ID; GRIB projection number (PDS octet 7)
50	original parameter ID; GRIB parameter number (PDS octet 9)
51	original process ID or model number (PDS octet 6)
52	original level type (PDS octet 10)

Grid data

Word	Description
offset + 64	beginning of the grid data
...	...
offset + 64 + (rows*columns) - 1	end of the grid data

API functions

Function	Description
mcgdir	opens a connection to read a grid file directory
mcgdrd	reads a grid file directory
mcgfdrd	reads a grid file header
mcgget	opens a connection to read a grid
mcgridf	reads a grid in Fortran (column-major) order
mcgridc	reads a grid in C (row-major) order
mcgput	opens a connection to write a grid
mcgoutf	writes a grid in Fortran (column-major) order
mcgoutc	writes a grid in C (row-major) order
igquit	deletes a grid file
m0gsort	gets grid selection parameters from the command line

*.GRX

*Graphics save tables for McIDAS-X, where * is a user-defined file name.*

These binary files contain a table of the red, green and blue color intensities (0 to 255) for the graphics levels on a McIDAS-X workstation. The table is of variable length, depending on the number of graphics levels allocated by the McIDAS-X session that generated the file. Use the GU application to create a McIDAS-X graphics save table.

For graphics save tables Application Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

Word allocation

Word	Description
0 - 3	reserved
4	red color intensity for graphics level 1
5	green color intensity for graphics level 1
6	blue color intensity for graphics level 1
...	...
3 * (n-1) + 1	red color intensity for graphics level n
3 * (n-1) + 2	green color intensity for graphics level n
3 * (n-1) + 3	blue color intensity for graphics level n
...	...
3 * (max-1) + 1	red color intensity for the maximum graphics level
3 * (max-1) + 2	green color intensity for the maximum graphics level
3 * (max-1) + 3	blue color intensity for the maximum graphics level

API functions

Fortran Function	Description
delgra	deletes a saved graphics table
resgra	restores a saved graphics table
savgra	writes a graphics save table

HIRSCRPF

HIRS calibration reference parameters.

HIRS calibration reference parameters are used to compute HIRS brightness temperatures. This binary file is organized chronologically by satellite with the last record being the most recent. Each record contains 48 words.

The first record is a header record. The following records contain data. Words 0 through 39 in the data records are the coefficients of fourth degree polynomials used to convert platinum resistance thermistor count values to temperatures. The first four words in each group of eight are for the warm blackbody target, and the fifth through eighth words in each group are for the cool blackbody, which is not routinely used in the calibration process.

This file is supplied with McIDAS-X.

For HIRS calibration reference parameters Application Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

Word allocation for record 1

Word	Description
0	total number of records in the file
1 - 47	unused

Word allocation for records 2 through n

Word	Description
0 - 7	array of thermistor counts-to-temperature coefficients, zeroth order
8 - 15	array of thermistor counts-to-temperature coefficients, first order
16 - 23	array of thermistor counts-to-temperature coefficients, second order
24 - 31	array of thermistor counts-to-temperature coefficients, third order
32 - 39	array of thermistor counts-to-temperature coefficients, fourth order
40 - 43	array of weights for blackbody target temperature
44	coefficient for calibrating the HIRS visible channel
45	reserved
46	NOAA satellite number
47	date, *yyyddd*

API functions

Function	Description
hircon	initializes the HIRS constants

HIRSTAUL

HIRS transmittance coefficients.

HIRSTAUL is a binary file containing 506-word records provided by the sensor source. Each sensor source has 20 records:

- Records 1 through 19 contain transmittance coefficients for channels 1 through 19 of the HIRS instrument.
- Record 20 contains other parameters used for radiative transfer calculations.

This file is supplied with McIDAS-X.

For HIRS transmittance coefficients Application Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

Word allocation for records 1-19

Word	Description
0 - 199	carbon dioxide transmittance coefficients
200 - 319	carbon dioxide angular correction coefficients
320 - 325	continuum transmittance coefficients
326 - 339	water vapor transmittance coefficients
340 - 419	ozone transmittance coefficients
420 - 499	ozone angular correction coefficients
500 - 505	flags indicating whether to use coefficients

Word allocation for record 20

Word	Description
0 - 19	central wave numbers of the channels
20 - 59	coefficients for the Planck function
60 - 99	band correction coefficients
100 - 139	reserved
140 - 144	solar coefficients
145 - 157	flux coefficients
158 - 179	reserved
180 - 187	synthetic coefficients for the surface temperature
188 - 189	reserved
190 - 197	empirical coefficients for the surface temperature
198 - 505	reserved

API functions

Function	Description
pfcoef	retrieves HIRS Planck function, solar, flux and surface temperature coefficients

MDXX*nnnn*

MD files, where nnnn is a user-defined number.

MD files contain data records addressed by row and column coordinates. All records in a row also share information called a row header. Similarly, all records in a column share a column header. A complete record thus consists of the row header, column header and data. The headers hold parameters common to the data records. Row headers are located in column 0; column headers are in row 0. Up to 1.6 million columns are allowed in MD files and are supported by the point data clients and servers. Fields in a record are identified by 4-character names called keys. Data values are stored as integers. Each record can be up to 400 words long; individual fields in a record are one word (4 bytes) long.

The MD file structure is self-contained. All information for accessing one of these binary files exists in the 4096-word header, which contains the schema specifying the default number of rows and columns in an MD file; the composition of the row headers, column headers and data records; and the names, scale factors, and units of the keys.

A copy of the schema resides in the MD file and in the McIDAS-X disk file named SCHEMA, which contains all schemas recognized by the system. The copy held in SCHEMA serves as a blueprint for all MD files of a particular kind of data. When an MD file is created, the schema is copied from SCHEMA to the MD file header block with all appropriate modifications.

MD file numbers can be between 1 and 999999. If an MD file number is five or six digits, the file name begins with only MDX or MD. For example, MD file number 12345 has the file name MDX12345, but MD file number 123456 has the file name MD123456.

MD files do not have to follow this standard naming convention. The file masking option of DSSERVE may be used to access a data file of any name through the ADDE.

For MD file Application Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

Word allocation

Word	Description
0	schema name
1	schema version number
2	schema registration date, *yyyddd*
3	default number of rows
4	default number of columns
5	total number of keys in the record
6	number of keys in the row header
7	number of keys in the column header
8	number of keys in the data portion
9	1-based position of the column header
10	1-based position of the data portion
11	number of repeat groups
12	size of the repeat group
13	starting position of the repeat group
14	missing data code
15	integer ID of the file
16 - 23	text ID of the file
24	creator's project number

25	creation date, *yyyddd*
26	creator's ID
27	zero-based offset to the row header
28	zero-based offset to the column header
29	zero-based offset to the data portion
30	first unused word in the file
31	start of the user record
32	start of the key names
33	start of the scale factors
34	start of the units
35 - 38	reserved
39	beginning Julian day of the data, *ccyyddd*
40	beginning time of the data, *hhmmss*
41	ending Julian day of the data, *ccyyddd*
42	ending time of the data, *hhmmss*
43 - 60	reserved
61 - 62	file name
63	MD file number
64 - 463	user record, MD coordinates (0,0); not described by the schema; use for storing arbitrary information
464 - 863	names of the file keys
864 - 1263	scale factors for the keys
1264 - 1663	units of the keys
1664 - 4095	reserved

The row header begins at word 4096. The row header's length in words is *rows* * *number of keys in the row header*.

Following the row header is the column header. The column header's length is *columns* * *number of keys in the column header*.

The MD data follows the column header. The data's length is *rows* * *columns* * *number of keys in the data portion*. Any remaining words are available for user purposes.

Words 39-42 are filled in only during the production of real-time files. When real-time file data is copied to other MD files, words 39-42 in the destination files are set to null.

API functions

Function	Description
m0pthdr	opens a connection to read a point source data header

m0ptrdhdr	reads the point source data header
m0ptget	opens a connection to read point source data
m0ptrd	reads point source data
mdo	writes point source data
mdquit	deletes a point source file
m0psort	gets point source selection parameters from the command line

MSUSCRPF

MSU calibration reference parameters.

The parameters in MSUSCRPF compute MSU (Microwave Sounding Unit) brightness temperatures and check the quality of MSU data. This binary file is organized chronologically by date; the last record is the most recent. Each record contains 48 words. There may be more than one record for a particular satellite.

The first record is a header record. The following records contain data. Words 0 through 11 in the data records are the coefficients of second-degree polynomials used to convert platinum resistance thermistor resistance measurements to temperatures for the internal warm targets.

This file is supplied with McIDAS-X.

For MSU calibration reference parameters Application Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

Word allocation for record 1

Word	Description
0	total number of records in the file
1 - 47	unused

Word allocation for records 2 through n

Word	Description
0 - 3	resistance-to-temperature coefficients, zeroth order; if word 0 is the missing value, the record was deleted
4 - 7	resistance-to-temperature coefficients, first order
8 - 11	resistance-to-temperature coefficients, second order
12 - 23	coefficients to linearize MSU counts
24 - 27	high and low calibration points for electronic systems A and B
28 - 35	average space and target temperatures for channels 1 to 4
36 - 43	standard deviations of space and target temperatures for channels 1 to 4
44	nominal target temperature
45	MSU serial number
46	NOAA satellite number
47	date, *yyyddd*

API functions

Function	Description
msucon	initializes the MSU constants

OUTL*

*Base map files, where * is the map file name.*

These variable-length, binary files contain the base map data for drawing graphical map outlines. The map files and their descriptions are provided in the table below.

Map file	Description
OUTLSUPU	high resolution USA
OUTLSUPW	world coastal
OUTLUSAM	medium resolution USA
OUTLHPOL	world political boundaries
OUTLHRES	high resolution world coastal outline
OUTLUSAL	low resolution USA and North America
OUTLWRLD	world coastal

Word allocation for OUTLSUPU, OUTLSUPW, OUTLUSAM, OUTLHPOL and OUTLHRES

Words 1 to 6000 contain the directory for the line segments. Each directory block contains six words of information.

Word	Description
0	number of blocks (line segments)
1	minimum latitude, degrees * 10000
2	maximum latitude, degrees * 10000
3	minimum longitude, degrees * 10000; west is positive
4	maximum longitude, degrees * 10000; west is positive
5	beginning word of data start for segment 1
6	number of words to read
7 - 12	directory for line segment 2
...	...
5995 - 6000	directory for line segment 1000
6001	latitude degrees * 10000
6002	longitude degrees * 10000
6003	latitude degrees * 10000
6004	longitude degrees * 10000
...	...

The maximum number of line segments is 1000. The number of points in a segment is limited to 3000 by the arrays in the MAP program.

Word allocation for OUTLUSAL and OUTLWRLD

Word	Description
0 . . . n-1	n number of latitude/longitude pairs as halfwords; longitude, degrees * 10, 0 to 360 west is positive; latitude, degrees * 20, 0 to 180 north is positive; odd number means pen down; even number means pen up
n + 1	end of file marker; HEX FFFF0000

API functions

Currently, no API library functions exist for reading and writing base map files. There are, however, programs in McIDAS-XRD, MAKEMAP and MAP2TEXT, that convert McIDAS map files to and from a text format.

SATANNOT

Image annotation description file.

The satellite description text information in this binary file is displayed as text annotation when a satellite image is loaded onto a display device.

Each 80-character record contains the following:

- Satellite name, in columns 1 to 19
- Band specifier code, if applicable, in column 20
- Sensor source number, in columns 30 and 31

For image annotation description file Application Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

File size depends on the number of satellite description entries and is computed by (20**number of entries*) words. For example, two entries may look like this:

Columns 1 to 19	Column 20	Columns 30 and 31
GOES-7 IR	D	33
NOAA-10	K	60

This file is supplied with McIDAS-X.

API functions

Function	Description
m0ssdesc	retrieves the satellite sensor information

SATBAND

Image channel description file.

This file contains 80 character ASCII text records describing the instrument channel information. For each satellite/instrument combination, there is an instrument description block which must conform strictly to the following format:

Format	Description
Sat *ss1..ssn*	First line of instrument description block, where *ss1..ssn* is a list of SSEC assigned satellite sensor numbers
Cal *type*	Instrument calibration type
BRes *lres eres*	Line and element base resolution (in km)
nnnn* DESC='*description*' WL=*wavel* WN=*waven	Channel number (*nnnn*), text description of channel (*description*; appears in output of IMGLIST FORM=BAND or ALL), wavelength (*wavel*), and wave number (*waven*)
EndSat	Last line of instrument description block

This file is supplied with McIDAS-X.

There is one instrument description block for each supported satellite.

For satellites that support multiple calibration types, the combination of Cal, BRes, and *nnnn* lines may be repeated for each calibration type.

API functions

Currently, no API library functions exist for reading and writing this file.

SKEDFILE

Scheduled McIDAS-X commands.

This binary file has 7172 words and is divided into two sections.

- The first section contains basic directory information for the scheduler entries. The scheduler (SKED) uses this section to quickly scan all entries to decide which are due to run.
- The second section contains additional directory information and the text of the McIDAS-X command for each entry. This section is read when listings are made or a command is ready to run.

The file division is invisible to all parts of the schedule system accessing the file via the ID number.

For scheduled McIDAS commands Application Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

Word allocation for the file

Word	Description
0 - 7	unused
8 - 807	first section of scheduler entry information
808 - 871	unused
872 - 7171	second section of scheduler entry information

Word allocation for the first section

This section contains eight words for each entry (maximum of 100).

Word	Description
0	time of the next scheduled run; all times are kept internally, in seconds, since 1 January 1972
1	number of runs remaining
2	interval between runs
3	late tolerance if the scheduler is delayed
4	terminal at which the command runs
5	ID number or name assigned when the command is entered in the schedule
6	initials of the person who entered the command in the schedule
7	project number under which the command runs

Word allocation for the second section

This section contains 64 words for each entry.

Word	Description
0	time that the command first runs
1	total number of times the command runs
2	unused
3 - 63	text of the McIDAS-X command

API functions

Function	Description
skio	schedule file I/O routines

UC

McIDAS-X User Common.

User Common is a block of data stored in shared memory. Its contents are lost if McIDAS-X stops; otherwise, it retains values that allow for interprocess communication of information usually related to the state of the McIDAS-X session (for example, the number of frames being viewed or the location of the cursor). UC variables with negative subscripts are carried along from the calling process through all spawned processes. UC variables with positive subscripts and UC(0) are available to all tasks that are running whether spawned or not. Words -12 to -1 contain a snapshot of the state taken just before the command begins.

For McIDAS user common Application Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

Word allocation

Word	Description
-123 to -120	current font information
-82	logon initialization flag; 1 = I option used by PC
-58	ending column number in the current MD wind file
-57	ending row number in the current MD wind file
-56	starting row number in the current MD wind file
-55	current MD wind file used for core output
-54	current MD wind file used for imv, selector and core output
-53	element size of the target cursor
-52	line size of the target cursor
-51	space bar toggle
-50	error status word
-46	current TCOL
-45	current TWIN
-44	second cursor state control word; bits 0-2 are not used; bits 3-5 are cursor color; bits 6-8 are cursor type
-43	second cursor element position
-42	second cursor line position
-41	second cursor element size
-40	second cursor line size
-38	next record pointer for the file created with DEV=F
-37 to -35	the disk file name for the DEV=F file; on the workstation, the file descriptors for the printer or file dataset for sdest, edest and ddest, respectively
-34	current graphics virtual frame; if less than zero, doesn't plot
-33	current diagnostic message (ddest) device: 0 = suppressed; 1 = terminal screen; 2 = local printer; 3 = system printer
-32	current error message (edest) device: 0 = suppressed; 1 = terminal screen; 2 = local printer; 3 = system printer

-31	device for standard text messages: 0 = suppressed; 1 = terminal screen; 2 = local printer; 3 = system printer
-30	auto context table search: 0 = no keyword substitution; 1 = parameter fetching subroutines; for example, CKWP, resort to system string table for missing keyword parameters
-26	0 = program currently running is not a macro 1 = program currently running is a macro
-25	0 = program currently running is background 1 = program currently running is foreground -1= program currently running is background with Fortran main; can do Fortran I/O 2 = program currently running is a console started task, such as a tape job
-24	current level when starting another process: 0 = scanner; 1 = next program, etc.
-23	number of this initiator
-22	0 = command was not started by the scheduler 1 = command was started by the scheduler
-21	abort signal handling; see word 455
-16	project number the command runs under; may be different from the logged on project number in UC(1)
-13	cursor state control word: if bits 0-2 = 0, cursor is frozen (PC command) if bits 0-2 = 1, cursor is controlled by joystick (P) if bits 0-2 = 2, size/position are joystick controlled (Z) bits 3-5 are cursor color (DT); bits 6-8 are cursor type (DT)
-12	cursor element position
-11	cursor line position
-10	cursor element size
-9	cursor line size
-8	graphics state control word: if bit 0 = 1, graphics are connected to loop control (J) if bit 1 = 1, graphics frame is looping (L) if bit 2 = 0, graphics frame is blanked (W)
-7	graphics lower bound
-6	graphics upper bound
-5	current graphics frame
-4	frame state control word: if bit 0 = 1, frame is connected to loop control (Y) if bit 1 = 1, frame is looping (L) if bit 2 = 0, frame is blanked (K)
-3	image frame lower bound
-2	image frame upper bound
-1	current image frame
0	user's terminal number

1	project number under which the current user is logged on; may be different from UC(-16)
2	user's initials
4	current navigation file number
5	current MD file number
6	current grid file number
11	number of lines on the screen; for terminals where all frames are the same size
12	number of elements on the screen; for terminals where all frames are the same size
13	number of image frames
14	number of graphics frames
15	0 = terminal is local ProNET; 1 = terminal is remote bisync; 2 = terminal is asynchronous
16	0 = terminal is nonvideo; 1 = terminal is video; 2 = terminal is host
17	flag for the E key: 0 = lat/lon are displayed in dddmmss; 1 = lat/lon are displayed in decimal
20	set to 1 by the G key
21	set to 1 by the Q key
22	import pointer
37	image display hardware: 6 = X Windows
38	graphics state flag: 0 = no graphics; 1 = draws graphics
39	virtual graphics state flag: 0 = doesn't write virtual graphics; nonzero = virtual frame to write
46	default graphics line width
47	graphics dash length, in pixels
48	graphics dash gap length, in pixels
49	graphics dash gap color
50	loop control system; consists of the LS command and the A, B, J, Y, O and L keys; images and graphics are connected to and disconnected from the loop system via UC 54 and 59; 1 = loop control system is looping; 2 = not looping
51	current image frame
52	image frame loop, upper bound
53	image frame loop, lower bound
54	1 = image frames are connected to the loop control 0 = image frames aren't connected
55	1 = image frames are visible 0 = image frames are blanked (Alt K)

56	current graphics frame
57	graphics frame loop, upper bound
58	graphics frame loop, lower bound
59	1 = graphics frames connected to loop control 0 = graphics frames not connected
60	1 = graphics frames are visible 0 = graphics frames are blanked (Alt W)
61	cursor size, vertical
62	cursor size, horizontal
63	cursor position, line number
64	cursor position, element number
65	cursor type: 1 = box; 2 = crosshair; 3 = box and crosshair; 4 = solid box; 5 = star wars
66	cursor color (0 - 7)
67	joystick 1: 0 = disconnected; 1 = controls cursor position; 2 = vernier cursor control; 3 = controls cursor size; 4 = velocity cursor
69	second cursor size, vertical
70	second cursor size, horizontal
71	second cursor position, line number
72	second cursor position, element number
73	second cursor type; see word 65
74	second cursor color; see word 66
75	cursor mode: 0 = single cursor mode; 1 = dual cursor mode
76	saved cursor type; set when word 65 = 0 to turn off the cursor
77 - 79	reserved
87	McBASI list window size
88	McBASI auto-line number line number
89	McBASI run abort flag
98	current MD core output file for motion vectors (WINDCO)
99	nortle toggle
100	current MD selector file for motion vectors (WINDCO)
121	single-letter command; otherwise, zero
122	used by the scheduler (sked); last time through all entries flag
158	vdisk drive letter

160	TCP/IP address of session partner
161	TCP rate measurement word
162	TCP connection status code
163	TCP reserved
164	WINDCO velocity cursor toggle set by Alt V: 0 = off; 1 = on
165	WINDCO sampling flag set by Alt X
166	WINDCO mode is active
167	reserved
168	0 = asynchronous communication is OK 1 = async comm is suspended, port available
169	1 = update loop when new image enters it
170	1 = loop in both directions (1 2 3 4 5 4 3 2 1)
171	reserved
172	routing table modification flag: 0 = modified; 1 = unchanged
173	briefing control signature for channel 1
174	briefing control signature for channel 2
176	1 = system is running without a session manager
177	communication connection state flag: 0 = no carrier; 1 = communications connected; 2 = connection in progress
178	switch to tell mousef to ignore mouse button press
185	mouse button status; two 2-byte integer (I* 2) values that correspond to mouse buttons 1 and 2 respectively
186	mouse movement values (mickies); two 2-byte integer (I * 2) values that correspond to movement up/down and left/right respectively
187	switch to tell mousef to set mickies into UC(186)
188	image nortle toggle
189	1 = commands go to the host
190	outgoing spool-in-use flag
191	mouse PID so EXIT can kill it
192	outgoing message request pointer
193	outgoing message completed pointer
194	1 = system shutdown request
195	last text window displayed; see UC(200)
196	count of text waiting to be displayed
197	import pointer from the decoder

198	communications port name, binary 0 for ProNET
199	last written pointer for import decoder
200	reserved
201	first line in the text window to display; the range is 1 to 57
202 - 203	mouse RAW position
204	reserved
205	reserved
206	reserved
207	nonzero means kbdctl is in insert mode
208	PF key input only; nonzero means that command line text is not allowed and the user cannot change the text window; the program must change the window
209	address of a memory segment used for displaying text on windows 5 through 9; each text window uses 4000 bytes; each character is represented in a 2-byte format where the least significant byte is the character displayed and the most significant byte is the color attribute
210	number of commands that mctext should remember, as specified in the -ih option of the .mcidasrc file
211	1 = echo all GUI commands to the mctext window
212 - 214	reserved
215 - 305	reserved
315 - 350	briefing port, frame controls and movie flag
400	number of text screens in use
401	reserved
451 - 453	cursor color (red, green, blue); the range is 0 to 255
455	1 = traps abort signals; see UC(-21)
456	ID of the command window
457	ID of the image window
465	offset from the start of UC to the start of redirect memory
467	1 = independent graphics
470	1 = to use fonts
471 - 480	family name of the font
500	number of graphics levels
501	interpolation strategy
502	1 = image window should resize to the size of the current frame (Alt R)
503	zoom factor; used when UC(300) is one

504	mcimage process ID
600	number of gray levels
700	TCPIP module loaded
701	IMPORT module loaded
1000 - 1999	user/site defined
2000	if nonzero, the image window should raise itself and reset the value to zero
2001+	frame object offsets from the start of UC
3001+	terminal types with variable frame sizes; line and element sizes as halfwords
4000	relative pointer for random looping
4001+	list for random looping, where zero is the end
5001+	list of dwell rates
6000	relative pointer for random graphics loops
6001+	list for random graphics loop, where zero is the end
7001+	list of random opposites, where zero is the end
16383 -	end of segment

API functions

Function	Description
luc	reads a word from User Common
puc	writes a word to User Common
mcmoubtn	reports the state of the mouse buttons as soon as a mouse button event occurs
Mcluc	returns a value from User Common
Mcpuc	changes a value in User Common

VASTBLS

VAS calibration tables.

VASTBLS is a binary file that holds the current calibration values of the temperature, brightness and radiance for all 12 channels of GOES AA satellite data. Its fixed size is 1622016 words.

Word allocation for the file

The primary blocks for VASTBLS are shown below. Each block represents a table of calibrated values for each raw value.

Word	Description
0 - 540671	radiance calibration block (watts/meter**2/steradian*1000)
540672 - 1081343	temperature calibration block, in Kelvin
1081344 - 1622015	brightness calibration block, 0 to 255

Word allocation for the primary segment

Each primary segment contains secondary segments (12 maximum) allocated by band number. Word addresses are relative to the start of the primary segment.

Word	Description
0 - 45055	band 1 block
45056 - 90111	band 2 block
90112 - 135167	band 3 block
135168 - 180223	band 4 block
180224 - 225279	band 5 block
225280 - 270335	band 6 block
270336 - 315391	band 7 block
315392 - 360447	band 8 block
360448 - 405503	band 9 block
405504 - 450559	band 10 block
450560 - 495615	band 11 block
495616 - 540671	band 12 block

Word allocation for the secondary segment

Each secondary segment contains base segments allocated by the Delta-F value. This value ranges from -5 to +5, providing 11 base segments per secondary segment. Each of the word addresses below is relative to the start of the secondary segment. The values are the actual numbers used to calibrate the data.

Word	Description
0 - 4095	Delta-F = -5
4096 - 8191	Delta-F = -4
8192 - 12287	Delta-F = -3
12288 - 16383	Delta-F = -2

16384 - 20479	Delta-F = -1
20480 - 24575	Delta-F = 0
24576 - 28671	Delta-F = +1
28672 - 32767	Delta-F = +2
32768 - 36863	Delta-F = +3
36864 - 40959	Delta-F = +4
40960 - 45055	Delta-F = +5

API functions

Currently, no API library functions exist for reading and writing this file.

VIRT*nnnn*

Virtual graphics files, where nnnn is the virtual graphics number.

These variable-length, binary files contain one or more virtual graphics scenes. A scene contains graphics attribute information, such as page boundary, line width, pen position and color information.

For virtual graphics file Application Program Interfaces (APIs), refer to the *API functions* list at the end of this section.

Word allocation

Below is a description of one scene in a virtual graphics file.

Word	Description
0	top of the page boundary and virtual graphics version as halfwords
1	bottom of the page boundary and line width as halfwords
2	left edge of the page boundary
3	right edge of the page boundary
4	beginning pen position/color triples where (in an x-y plane): word n = pen y position word n + 1 = pen x position word n + 2 = pen color Decimal 256 marks the end of a scene.

API functions

Fortran Function	Description
vpout	writes to a virtual graphics file

Chapter 7

Writing ADDE Servers

This chapter describes the procedures that you will use to build data servers for McIDAS-X ADDE applications. You'll learn:

- Information about McIDAS-X servers that you should know before you begin writing a server
- Steps for creating a server and the McIDAS-X library functions that you will use
- How to debug your server
- The request syntax and transmission format for each McIDAS-X data type

This chapter is organized into the following sections:

Overview

ADDE (Abstract Data Distribution Environment) is the client/server mechanism for distributing data in McIDAS-X. In ADDE, an application sends a request for data through an API routine to a server. The server is the software running on a machine in a distributed system that stores data and supplies it to the client upon request.

What does a server do?

The server's job is to:

- Interpret the data request from the client
- Retrieve the requested data from disk
- Arrange the data into the proper format for the client
- Send the data back to the application

When the application reads the data sent from the server, the physical location of the original data and its stored format are transparent.

How many server categories does McIDAS-X have?

McIDAS-X has two server categories: primary and secondary.

- Primary servers are started directly by the ADDE communications module, **mcserv**. They read the client request from **stdin** (standard input).
- Secondary servers are started by primary servers. They don't read the client request from **stdin**, but rather from the server's argument list.

In short, a user enters a data request based on the ADDE group and descriptor name. **mcserv** starts the appropriate primary server based on the request type. The primary server extracts information about the requested dataset from the server mapping table, including the stored format of the data. If the stored data format is different from the standard McIDAS-X stored formats, the primary server starts a secondary server to convert the data to the format the client expects.

Getting started

Before writing an ADDE server, you should understand some basic concepts about servers in McIDAS-X. This section describes the following:

Client and server communication

The ADDE design is stream-oriented, so both the client and server can work simultaneously. An ADDE server reads a data request sent from the client via **stdin** (standard input) and sends data back to the client using **stdout** (standard output) through a pipe.

Although the size of the data sent from the server may be many megabytes, intermediate data storage on the server or client is not needed. Since the pipe is a finite size, the server will wait to write if the pipe is full. The client will wait for up to two minutes if the pipe is empty. If no activity takes place on the pipe after two minutes, the process stops. This is important for requests that take a long time to fulfill due to extensive searching. An environment variable, ADDETIMEOUT, can be set on the server to change this timeout length.

Rules for transmitting data

The rules below apply to all data transmissions between the server and client.

- All transactions from the server to the client are performed in pairs. A 4-byte value containing the total number of bytes to be sent is transmitted first, followed by the data.
- All numeric values are sent in network-byte-order, or big-endian. All client APIs for ADDE expect data to be transmitted in big-endian format and perform the necessary byte swapping where appropriate. Byte swapping is only performed on integer values; character strings are not flipped.
- If you transmit floating point values via ADDE, send the data as scaled integers. The platforms supported by McIDAS-X are not guaranteed to use the same bit representation for floating point values.

Primary servers

Primary servers, along with the client APIs, define the client selection syntax and the transmission format between the server and the client. The selection syntax and transmission format are different for the various data types in McIDAS-X. The table below lists the current primary servers provided in McIDAS-X.

Client APIs	Client request type	Data type	Server name	Description	Secondary server suffix
mcadel	ADEL	image	**adelserv**	deletes images from a dataset	
mcadir **mcadrd**	ADIR	image	**adirserv**	retrieves image header information	ADIR
mcaget **mcalin** **mcapfx** **mcanav** **mcacal** **mcacrd**	AGET	image	**agetserv**	retrieves the image header, navigation, calibration and data; data is returned line by line	AGET
mcaput **mcaout** **mcacou**	APUT	image	**aputserv**	writes an image object to a dataset	
mcaput	ATOK	image	**atokserv**	checks file permissions for writing image objects	
mcgdir	GDIR	grid	**gdirserv**	retrieves grid header information	GDIR

mcgfdrd **mcgdrd**					
mcgget **mcgridf** **mcgridc**	GGET	grid	**ggetserv**	retrieves the grid header and entire grid	GGET
mcgput **mcgoutf** **mcgoutc**	GPUT	grid	**gputserv**	writes a grid object to a dataset	
mcgput	GTOK	grid	**gtokserv**	checks file permissions for writing grid objects	
m0pthdr **m0ptrdhdr** **m0pthdrd**	MDFH	point	**mdfhserv**	retrieves point file header information	
ptcopy command	MDHD	point	**mdhdserv**	retrieves point header information	
m0ptget **m0ptrd**	MDKS	point	**mdksserv**	retrieves the point header and data	KS
m0navget **M0ReadNavBlock**	NAVG	nav	**navgserv**	retrieves satellite navigation	
M0obtxget **M0obtxread**	OBTG	text	**obtgserv**	retrieves observational weather text	OBTG
M0textget **M0txtread**	TXTG	text	**txtgserv**	retrieves an ASCII text file	
M0wtxget **M0wtxread**	WTXG	text	**wtxgserv**	retrieves textual weather information	WTXG

The ADDE communications module, **mcserv**, starts the primary servers based on the server's IP address, which tells **mcserv** if the request will be fulfilled locally or remotely. If the request is handled locally, **mcserv** finds the ADDE request type and runs the appropriate server process. The server process reads the body of the client request, acquires the data requested and sends the data back to the client.

If the request is handled remotely, **mcserv** opens a connection to the remote server and acts as a TCP-to-pipe bridge, sending out the request. On the server machine, **inetd** receives the connection and creates a child process running the same **mcserv** module with slightly different command arguments. Like local requests, this version reads the beginning of the client request, determines which server process is needed and runs it. The server processes the request and sends the data requested back to the client.

Secondary servers

In ADDE, the client requesting data doesn't care about the file format of the stored data. It only cares that the data is delivered in a well-known, predefined format. For example, if you have gridded data stored in a flat ASCII file, the McIDAS-X GRDDISP command can contour that data as long as the server delivers the gridded data to the client in the appropriate format. To accomplish this, you need a secondary server.

The secondary server's job is to:

- Interpret the client request
- Read the data from disk in its native format
- Convert the data to the format the client expects
- Send the data to the client

Sample data request

The example below shows the steps that a primary and secondary server will use to fulfill a data request. You should assume that the server administrator inserted the two commands below into the server mapping table:

```
DSSERVE ADD GFS/00 GRID 101 110 TYPE=GRID "00Z GFS Model Run DSSERVE ADD LOCAL/MODEL FLAT
TYPE=GRID "Local Model Grids
```

The first entry created the grid dataset GFS/00, which is stored in the standard McIDAS-X grid format in grid file numbers 101 to 110. The second entry created the grid dataset LOCAL/MODEL. The grids for this dataset aren't stored in the McIDAS-X grid format, but in a format called FLAT. If a user enters a GRDDISP command to display a grid from each of these datasets, the steps taken by the server to fulfill these requests are different.

Using a primary server

If the user enters the GRDDISP command below, the steps that follow are performed once the appropriate machine is identified.

Type: **GRDDISP GFS/00**

1. **mcserv** starts the primary server **ggetserv**.
2. **ggetserv** reads the server mapping table entry for the dataset GFS/00.
3. Because the grids from GFS/00 are in the standard McIDAS-X format, **ggetserv** processes the data request and sends the data to the client.

Using a secondary server

If the user enters the command below, the steps that follow are performed.

Type: **GRDDISP LOCAL/MODEL**

1. **mcserv** starts the primary server **ggetserv**.
2. **ggetserv** reads the server mapping table entry for the dataset LOCAL/MODEL.
3. Because the grids are stored in a non-standard McIDAS-X format called FLAT, **ggetserv** starts the secondary server **flatgget**.
4. **flatgget** processes the data request and sends the data to the client.

Building your ADDE server

Whether you're creating primary or secondary ADDE servers, they must perform these seven steps:

1. Initialize the McIDAS-X file system to recognize MCPATH and file redirection.
2. Read the ADDE client request.
3. Read the server mapping table.
4. Interpret the client request.
5. Retrieve the requested data from disk.
6. Send the data to the client.
7. End the transaction.

Steps 1, 2, 3 and 7 are the same for all data types; steps 4, 5 and 6 are different among the data types. All seven steps are described in this section, along with the function calls required to perform them.

This section is organized into these parts:

McIDAS library functions

The McIDAS-X library functions that you will use when creating an ADDE server are listed alphabetically in the table below.

C function	Fortran function	Description
	initblok	initializes the McIDAS-X environment for the server
M0isrtdataset	**m0isrtdataset**	determines if the dataset requested was flagged as a real-time dataset
M0IsTraceSet	**m0istraceset**	given a client request, activates tracing if TRACE=nonzero value
M0swbyt4	**swbyt4**	conditionally flips 4-byte data buffers
	swbyt2	conditionally flips 2-byte data buffers
M0sxdatasetinfo	**m0sxdatasetinfo**	extracts the server mapping table information
M0sxdone	**m0sxdone**	ends the ADDE transaction
M0sxGetClientRequest	**m0sxgetclientrequest**	reads the request header and request string sent from the client
M0sxread	**m0sxread**	reads data sent from the client to the server
	m0sxresolv	appends information from the server mapping table to the client request
M0sxsend	**m0sxsend**	sends data from the server to the client
M0sxSetTraceOff	**m0sxsettraceoff**	turns tracing off
M0sxSetTraceOn	**m0sxsettraceon**	turns tracing on
M0sxtrce	**m0sxtrce**	writes tracing information to a trace file

M0InitLocalServer		initializes the environment for secondary servers
Mctrace	**mctrace**	writes tracing information to a trace file

Initializing the McIDAS-X file system

Your server must be able to access McIDAS-X disk files, since most servers are built using disk file utilities. Use the **initblok** function to initialize the McIDAS-X disk file system so the server can recognize MCPATH and the redirection table, as shown below. Note that the parameter passed into **initblok** must be a 2-byte integer.

In Fortran:

```
integer*2 init_stat : ok = initblok (init_stat)
```

In C:

```
short init_stat; : ok = initblok_ (&init_stat);
```

For more information about disk files, see the section titled *McIDAS disk files* in *Chapter 5, Accessing Data*. For more information about file redirection and MCPATH, see the section titled *McIDAS user applications* in *Chapter 2, Learning the Basics*.

Reading the ADDE client request

Each ADDE client request received by the server contains a 160-byte *request header*, which contains system-level information built for the server.

The table below shows the input components included in the request header. The third column contains the starting word locations if the server is written in Fortran; the fourth column contains the variable names for the C structure, **servacct**, if the server is written in C.

Request header components	Length, in bytes	Word number in Fortran	servacct structure names in C
server IP address	4	1	server_address
server port used	4	2	server_port
client IP address	4	3	client_address
user ID	4 (ASCII)	4	user
project number	4	5	project
password	12 (ASCII)	6 - 8	password
transaction type	4 (ASCII)	9	transaction
number of bytes received by server	4	10	input_length
ADDE request string	120 (ASCII)	11 - 40	text

The last 120 characters of the request header is a buffer that may hold the client request string. The request string contains the group and descriptor names, and any selection conditions the server may need to fulfill the client request. If the request string is too long to fit in this buffer, the request string variable will contain a value of zero in bytes 4-7; bytes 0-3 will contain the actual length of the request string, which the server then must read from **stdin**.

The process for reading the ADDE client request is performed with one of two functions, depending on whether the server reading the request is a primary or secondary server.

Words 41 to 64 are for sending a return packet. The section *Ending the transaction* in this chapter provides more information.

Using a primary server

To read the client request from a primary server, call **M0sxGetClientRequest** or **m0sxgetclientrequest**, as shown in the code fragments below.

In Fortran:

```
include 'fileparm.inc' character*(MAXPATHLENGTH) request integer request_block(64) : : ok
= m0sxgetclientrequest(request_block, request)
```

In C:

```
#include "mcidas.h" : servacct request_block; char *request; int ok; ok =
M0sxGetClientRequest (&request_block, &request);
```

Upon successful completion, **`request_block`** contains the request header information sent to the server, and **`request`** contains the request string, including the group and descriptor names and selection conditions specified.

Using a secondary server

To read the client request from a secondary server, call the function **M0InitLocalServer**, as shown in the code fragment below.

```
const char Server[] = {"TESTSERV"}; servacct request_block; char *request; : : ok =
M0InitLocalServer (Server, &request_block, &request);
```

Upon successful completion, **`request_block`** contains the request header information sent to the server, and **`request`** contains the request string, including the group and descriptor names and selection conditions specified.

If the dataset requested by the client is stored in a non-standard McIDAS-X format on the server, the primary server starts the secondary server with a call to **m0subserv**. **m0subserv** builds an argument list for the secondary server and starts it with a call to **execlp**. The argument list contains both information from the client request block and the actual client request string. The **M0InitLocalServer** function extracts values from the secondary server's argument list and puts them in the client request block.

Reading the server mapping table

Once the server successfully receives the client's request, it must extract information from the server mapping table, which is a database that provides the server with information needed to fulfill a user request. The server mapping table is manipulated by the DSSERVE command and contains information about the location and format of the data on the server machine. This information is accessed by the server based on the group and descriptor names included in the client request.

The function **M0sxdatasetinfo** or **m0sxdatasetinfo** extracts information that the server needs from the server mapping table, including:

- ADDE group and descriptor names
- Stored data type and its format
- Minimum and maximum file numbers in the dataset
- Any additional information needed to fulfill the request
- Dataset comment section
- Flag indicating if the dataset is a real-time dataset

The code fragment below demonstrates the use of **M0sxdatasetinfo**. It assumes that the server administrator entered the following DSSERVE command, which created the ADDE dataset GMS/IR:

Type: **DSSERVE ADD GMS /IR AUST 3 8 TYPE=IMAGE INFO=east_ir.cfg "GMS Infrared Imagery**

```
char *request; char *group; char *dataset; char *type; char *format; char *info; char
*comment; int min_range; int max_range; int real_time; ... ... ... ok = M0sxdatasetinfo
(request, &group, &dataset, &type, &format, &info, &comment, &min_range, &max_range,
&real_time);
```

Upon successful completion, the output variables will contain the following values:

Variable	Value

group	GMS
dataset	IR
type	IMAGE
format	AUST
info	east_ir.cfg
comment	GMS Infrared Imagery
min_range	3
max_range	8
real_time	0 (default)

Interpreting the client request

After the server reads the client request and extracts the dataset information from the server mapping table, it parses the client request so the data can be located and the request fulfilled.

Since client requests are similar in form to a McIDAS-X command, it is easiest to use the McIDAS-X command line parsing routines to extract information from the request. To use common command line data retrievers, such as **Mccmdstr** and **Mccmdint**, you must initialize the command line subsystem. You only need to do this for primary servers. The **M0InitLocalServer** function automatically performs these steps for secondary servers.

The code fragments below demonstrate how to initialize the command line subsystem in both Fortran and C.

In Fortran:

```
character*256 request integer len_req : c--- request already contains the request string
to be parsed call m0cmdput (m0cmdparse (request, len_req))
```

In C:

```
char *request; int len_req; int stat; : /* request already contains the request string to
be parsed */ stat = M0cmdput (M0cmdparse (request, &len_req)) ;
```

Once the request string is parsed by the command line subsystem, it is your job, as the server developer, to find the data matching the request. The section titled *Request syntax and data transmission formats* later in this chapter explains the request syntax for each of the McIDAS-X data types.

Retrieving the requested data from disk

The method of retrieving data matching the client request will vary from file format to file format.

When determining the location of data files, you should be aware that the server mapping table may not have enough entries available for an individual dataset to fully explain the filing format. If this happens, use a configuration file and store the name of that file in the INFO field of the server mapping table. Then the server can get additional file location information there.

Sending the data to the client

When the request is parsed and the appropriate data is located, it can be sent back to the client. All data transactions are done in pairs (count + data) using the function **M0sxsend** or **m0sxsend**. First, a 4-byte value containing the length of the data being sent to the client is transmitted. Then that many bytes of data are transmitted.

To ensure that data sent between the server and the client is in network-byte-order (big-endian), include calls to **M0swbyt4** or **swbyt4** for all integer values. This function flips 4-byte memory segments on little-endian machines; it has no effect on big-endian machines.

The first value sent from the server to the client is read internally by the client function **M0cxreq** or **m0cxreq** before any API-level reading routines are called. For example, when serving image data, this value is the total number of bytes the server will send to completely transmit the data object. Some ADDE clients expect a dummy, nonzero value in this location. If a zero is sent back, **M0cxreq** or **m0cxreq** assumes the server was unable to fulfill the request. The complete transmission format for each McIDAS-X data type is described later in this chapter in the section titled *Request syntax and data transmission formats*.

The code fragment below demonstrates the appropriate way to send data from the server to the client. It assumes the client wants to receive a 4-byte dummy value of one, which will be read by **m0cxreq**, followed by two 40-byte records. Words 2 through 4 in the record are character values; the remainder of the record contains integer values.

```
integer n_bytes ! number of bytes to send integer n_bytes_nbo ! number of bytes in network
byte order integer dum_val integer buffer(10) c--- send the value 4 to the client
indicating that it is to read an c--- additional 4 bytes; the additional 4 bytes will
contain the c--- value 1; this pair is read on the client in m0cxreq. n_bytes = 4
n_bytes_nbo = n_bytes call swbyt4(n_bytes_nbo, 1) call m0sxsend (4, n_bytes_nbo) dum_val =
1 call swbyt4(dum_val, 1) call m0sxsend (n_bytes, dum_val) c--- now we will loop 2 times,
getting new data and sending it c--- back to the client; remember that words 2 through 4
are c--- character strings. do 10 I = 1 , 2 c--- get the data ok = datareader (buffer) !
fictitious function c--- send the record length back n_bytes = 40 n_bytes_nbo = n_bytes
call swbyt4(n_bytes_nbo, 1) call m0sxsend (4, n_bytes_nbo) c--- convert the appropriate
locations in the buffer to c--- network byte order call swbyt4(buffer(1), 1) call
swbyt4(buffer(5), 6) c--- send the data back to the client call m0sxsend (n_bytes, buffer)
10 continue
```

Ending the transaction

After the data is transmitted to the client, the server ends the transaction by calling **M0sxdone** or **m0sxdone**. This function sends a 96-byte trailer, which is filled on the server and sent to the client at the end of the transaction. Its contents are described in the table below.

Trailer description	Length, in bytes	Word number in Fortran	servacct structure names in C
number of bytes going out	4	41	reply_length
100 x CPU seconds	4	42	cpu
return code	4	43	returncode
error message	72 (ASCII)	44 - 61	errormsg
start date in IYD format	4	62	date
start time of transaction in hhmmss	4	63	start_time
end time of transaction in hhmmss	4	64	end_time

Before the transaction ends, you must set the return status code and any error messages that may have voided the transaction. The only successful return code is zero, which appears in word 43 of the client request block. If you send a nonzero value to the client, also put an error string in words 44-61 so the client API will print a message telling the user why the request couldn't be fulfilled. If your site uses the ADDE accounting software, also fill word 41 with the total number of bytes transmitted to the client.

The code fragment below shows the calling sequence that a server should include for a request that can't be fulfilled.

```
integer request_block(64) logical syntax_error : : c---there is a syntax error in the
client request if (syntax_error)then comm_block(43) = -1001 call movcw
(request_block(44),'Syntax error in the user request') endif call m0sxdone (request_block)
```

Using this example, the message below is printed by the function **m0cxreq** if the server encounters a syntax error.

```
COMMAND: Syntax error in the user request -1001
```

Debugging servers

Because ADDE servers read from **stdin** and write to **stdout**, you can't call the **sdest** or **Mcprintf** function to trace the progress of a server for two reasons:

- First, **sdest** and **Mcprintf** statements are sent via **stdout**. If you send debug messages through **stdout**, they will corrupt the data stream and cause catastrophic errors on the application side.
- Second, since you don't directly call the server, you won't be able to see the output and, thus, can't print the server progress. This is done from a mother task that may not even be running on your machine.

The solution is to dump messages needed for tracing errors to a file. If errors are reported and tracing is activated, the function **M0sxtrce** or **m0sxtrce** will write trace messages to a file named **trce**. If tracing is not activated, **M0sxtrce** or **m0sxtrce** does nothing.

The keyword TRACE is appended to each client request. If TRACE=0, which is the default, tracing is not performed. If TRACE=*nonzero value*, tracing is activated. The function **M0IsTraceSet** or **m0istraceset** automatically activates tracing on the server if the value for TRACE in the client request string is a nonzero number. If you write secondary servers, a call to **M0IsTraceSet** is made within the function **M0InitLocalServer**. To activate tracing within a server on your own, call **M0sxSetTraceOn** or **m0sxsettraceon**. To turn it off, call **M0sxSetTraceOff** or **m0sxsettraceoff**.

Since each account has only one trace file, prefix each line of your tracing strings with the name of the server generating that string. When you look through the contents of the trace file, you can easily find the trace information generated by the problem server.

You can also use the **Mctrace** or **mctrace** function to automatically append the server name to the trace message and set a flag for each message. The flag restricts tracing to only selected values of the keyword TRACE.

The sample code fragment below is from the server TOMGSERV.

```
const char Server[] = {"TOMGSERV"}; char *request; servacct request_block; char
trace_string[500]; short init_stat; initblok_ (&init_stat); /* get the client request and
set tracing */ ok = M0sxGetClientRequest (&request_block, &request); ok = M0IsTraceSet
(request); /* print a trace message containing the entire user request */ sprintf
(trace_string, "%s: %s", Server, request); M0sxtrce (trace_string) ; /* continue
processing */
```

If the user enters the following client request:

Type: **GMS AUST TIME=12 DAY=1996352 ID=YSSY TRACE=1**

Upon completion, the file trace will contain this line:

```
TOMGSERV: GMS AUST TIME=12 DAY=1996352 ID=YSSY TRACE=1
```

Request syntax and data transmission formats

This section is organized into the following topics:

In ADDE, the client will manipulate data regardless of the format in which it is stored on the server. For this to happen, however, you must follow a specified set of rules for processing data and delivering it to the client. This section describes the client request syntax and data transmission formats for each data type supported in McIDAS-X. The table below provides a summary of the necessary components for each request type, in alphabetical order.

Request type	Data type	Description	Client requester	Client reader	Sample McIDAS-X command
ADIR	image	image header information	**mcadir**	**mcadrd**	
AGET	image	image header, navigation, calibration and data; data is returned line by line; comments	**mcaget**	**mcalin** **mcapfx** **mcacrd** **mcacal** **mcanav**	
GDIR	grid	grid header information	**mcgdir**	**mcgfdrd** **mcgdrd**	
GGET	grid	grid header and entire grid	**mcgget**	**mcgridf**	
MDFH	point	point file header information	**m0pthdr**	**m0ptrdhdr**	FORM=FILE
MDHD	point	point header information			PTLIST FORM=PARAM
MDKS	point	point header and data	**m0ptget**	**m0ptrd**	PTLIST
NAVG	nav	satellite navigation	**m0navget**	**m0snlist** **m0nvblk**	
OBTG	text	observational weather text	**M0obtxget**	**M0obtxread**	
TXTG	text	ASCII text file	**M0textgt**	**M0txtread**	
WTXG	text	textual weather information	**M0wtxget**	**M0wtxread**	

All secondary servers must perform the four basic tasks below to deliver the appropriate data to the client.

1. Read the client request based on the syntax described for the data type.
2. Verify that the client request adheres to any special rules which apply to that data type.
3. Convert the data to the format the client expects.
4. Transmit the data to the client in the appropriate format.

Since ADDE client requests are sent as character strings with a format similar to McIDAS-X commands, it is easiest to use the McIDAS-X command line retrieving routines to extract information from the client request. The request syntax description for each data type described in this section uses the McIDAS-X command line notation.

Additionally, remember that transactions of integer values between the client and the server are performed in network-byte-order (big-endian). Unless otherwise noted, assume that all strings sent back to the client are blank padded.

Serving image data

The ADDE server, **agetserv**, processes a client request and returns a complete image object to the client. An image object includes an image header and the image lines, and may also contain a line prefix for each line, calibration, navigation and comment cards. Due to the volume of data associated with image objects, the server delivers the data portion of the object to the client one line at a time. The McIDAS-X command IMGDISP accesses image objects.

Image object request syntax

The table below lists the client request syntax options for an image object. Note that keywords 1 through 9 are placed in the request block as positional parameters.

Keywords	Description	Remarks
' ' 1	group name	
' ' 2	dataset name	
' ' 3	position in the dataset	relative position in the dataset; negative means Nth most recent
' ' 4	coordinate type and position	(E)arth, (I)mage, (A)rea; (C)entered or (U)pper; no default; this field is used with the next two parameters; for example, to return an image centered at latitude 43.5 and longitude 90.0, the client request will contain EC 43.5 90.0
' ' 5	latitude or line number	latitude of the Earth (dd:mm:ss) or line number
' ' 6	longitude or element number	longitude of the Earth (dd:mm:ss) or element number
' ' 7	image resolution	default=1
' ' 8	number of lines to transmit	default=480
' ' 9	number of elements to transmit	default=640
VERSION=	transmission version number	value as of 11/96 is 1
TIME=*btime etime*	time of day selection range	the format is *hh:mm:ss* (no default)
DAY=	the day that data is valid	not a range of days; format must be *yyddd* or *ccyyddd* (no default)
UNITS=	calibration type requested	not all calibration types are valid for a data type (default=stored units)
SPAC=	number of bytes per data point	be careful when reducing data point resolution, if the data can be sent back in lower resolution (default=stored format)
CAL=	set to QTIR to return TIROS in quick calibration	
STYPE=VISR	sets calibration to	converts data type to 1-byte data

	UNITS of BRIT and type of VISR	
BAND=*band* BAND=ALL	specific band number to send or ALL bands	
LMAG=	line magnification factor	blow ups are done on the client to conserve transmission bandwidth (default=1); values must be integers; negative numbers mean a blowdown must be performed
EMAG=	element magnification factor	blow ups are done on the client to conserve transmission bandwidth (default=1) values must be integers; negative numbers mean a blowdown must be performed
DOC=	if YES, include the line documentation block	default=NO
AUX=	if YES, additional calibration information is sent	

Server-specific keywords	**Description**
ID=	NEXRAD station ID (NEXRAD server)
TIME=*btime etime* C	**C**overage; accesses data for specified time range (realtime POES server)
WL=	wavelength (AIRS server)
WN=	wavenumber (AIRS server)

Special rules for transmitting an image object

You must adhere to the rules below when transmitting an image object. The first five are specific to the client request; the last three are specific to the transmission format sent back to the client.

- If the position in the dataset is greater than zero, the client is expecting an absolute position number within the dataset.
- If the position in the dataset is less than zero or equal to zero, it is a time relative offset, with 0 being the most recent, -1 being the second most recent, etc.
- If the number of elements is 99999, assume the entire image object will be transmitted.
- If STYPE=VISR, then SPAC=1 and UNITS=BRIT. The prefix is not sent back unless DOC=Y.
- If the UNITS requested is BRIT, SPAC=1. If UNITS is TEMP/ALB, SPAC=2. If UNITS is RAD, SPAC=4.
- If the user asks for data in 2-byte format, but it can be sent back as 1-byte, send it as 1-byte and let the client handle the data expansion. This reduces the data volume sent across the network by half.
- If the request is larger than the source image file, pad the return image with zeros. Padding zeros is only required at the top or to the left of the image being transferred. If zeros are also needed to the right or the bottom of the image, they can be sent as zeros, or words 9 or 10 of the directory can be modified to indicate a smaller image than requested is being sent back (see table below). The client interface will then pad to the right and/or to the bottom of the image.
- If the request is larger than the source image file, pad the return image with zeros. If the server has problems fulfilling the request, use the standard error values and messages found in **`~mcidas/data/ADDERROR.DOC`**. The range -11000 to -11999 lists error codes specific to image objects. Use enumerated error codes and messages when applicable and create new error codes and messages for error conditions that are unique to your site.

Converting the image object's format

The image object contains the image header and the image lines. It may also contain line prefixes, navigation, calibration and comment cards. Each of these components is created by the server. Their formats are described in Chapter 6 of this manual in the *AREAnnnn* data file documentation. Below are the common modifications needed for the image header.

Word	Description	Reason for modification
1	ADDE dataset relative position number	always modified
6	upper-left image line number	1) if client does not specify SIZE=ALL 2) if client requests coordinate base transfer
7	upper-left image element number	1) if client does not specify SIZE=ALL 2) if client requests coordinate base transfer
9	number of lines in the image	1) if client does not specify SIZE=ALL 2) instead of sending lines containing all zeros at the bottom of the image
10	number of elements in the image; must be divisible by 4	1) if client does not specify SIZE=ALL 2) instead of sending lines containing all zeros at the bottom of the image
11	number of bytes per band	1) if SPAC≠source value 2) if client requests STYPE=VISR
12	line resolution	if LMAG≠1
13	element resolution	if EMAG≠1
14	number of spectral bands	if BAND≠ALL
15	length of line prefix	1) if BAND=ALL is not specified 2) if SIZE=ALL is not specified 3) if DOC=YES is specified 4) if STYPE=VISR
19	spectral band map	if BAND=ALL is not specified
34	byte offset to the start of the data block	always modified (256+navlength+callength)
35	byte offset to the start of the navigation block	always modified (usually 256)
36	validity code	1) if request contains DOC=NO, set to 0 2) if SIZE=ALL is not specified, set to 0 3) if DOC=YES, must be set
49	length of the prefix documentation	1) if DOC=NO is specified, set to 0 2) if SIZE=ALL is not specified, set to 0
50	length of prefix calibration	if STYPE=VISR, set to 0
51	length of prefix band length	1) if STYPE=VISR, set to 0 2) if BAND≠ ALL
52	source type; satellite-specific (ASCII)	if STYPE=VISR, set to VISR

53	calibration type; satellite-specific (ASCII)	if STYPE=VISR, set to BRIT
54	sampling/averaging flag	set to 1 if blow down is performed by sampling
57	source's original satellite type (ASCII)	if STYPE=VISR, set to original satellite type
58	units of values returned	if AUX=YES and UNIT is specified
59	scaling of values returned	1) if UNIT is specified 2) if AUX=YES, set to 1
63	byte offset to the start of the calibration block	if calibration is sent (usually 256+navlength)
64	number of comment cards	if comment cards exist

Transmitting the image object to the client

Once the data is formatted correctly, the image object can be sent to the client. Because the transmission protocol is count+data, the server will send the following information to the client:

- 4-byte value containing the total number of bytes to be sent
- 256-byte image header
- Navigation (variable length)
- Calibration, if requested (variable length)
- Image data lines, one at a time:
  - line prefix (if requested)
  - line of data
- 80-byte comment cards

The sample code below shows you how to set up your server to transmit image objects.

```
integer image_header(64) integer nav_block(1024) integer cal_block(1024) integer
line(2000) integer prefix(100) integer comment(20) integer total_bytes integer nav_size
integer cal_size integer pre_size integer n_comments integer n_lines integer n_elements
integer data_size integer n_send integer n_send_nbo c--- assume the image_header,
nav_block, cal_block have already been loaded n_lines = image_header(9) n_elements =
image_header(10) data_size = image_header(11) nav_size = image_header(63) -
image_header(35) cal_size = image_header(34) - image_header(63) pre_size =
image_header(15) n_comments = image_header(64) total_bytes= (n_lines * n_elements *
data_size) + & (n_rows * pre_size) + & (n_comments * 80) & (nav_size + cal_size) c--- send
the total number of bytes in the image object n_send_nbo = total_bytes call
swbyt4(n_send_nbo, 1) call m0sxsend(4, n_send_nbo) c--- send the image header; assume
imageheadernbo converts the header c--- to network-byte-order call
imageheadernbo(image_header) call m0sxsend(256, image_header) c--- send the nav block;
assume navnbo converts the nav block to c--- network-byte order if (nav_size .gt. 0)then
call navnbo(nav_block) call m0sxsend(nav_size, nav_block) endif c--- send the cal block;
assume calnbo converts the cal block to c--- network-byte order if (cal_size .gt. 0)then
call calnbo(cal_block) call m0sxsend(cal_size, cal_block) endif c--- now we will send the
image data lines, one line at a time; c--- assume getoneline retrieves one line of data as
packed bytes do 10 i = 1 , n_lines ok = getoneline(i, line, prefix) if (pre_size .gt.
0)then call m0sxsend(pre_size, prefix) endif call m0sxsend(n_elements, line) 10 continue
c--- send the comments; assume readcomment reads one comment card if (n_comment .gt.
0)then do 20 i = 1 , n_comments call readcomment(i, comment) call m0sxsend(80, comment) 20
continue endif
```

Serving image directory data

The ADDE server, **adirserv**, processes a client request and returns image directories and comment cards to the client. The McIDAS-X command IMGLIST accesses image directories.

Image directory request syntax

The table below lists the client request syntax options for image directories. Note that keywords 1 through 4 are placed in the request block as positional parameters

Keywords	Description	Remarks
'' 1	group name	
'' 2	descriptor name	
'' 3	beginning file position number	can be the value ALL; positive numbers represent absolute locations; negative numbers are time-relative offsets (no default)
'' 4	ending file position number	default=beginning file number
AUX=	when YES, sends the center lat/lon, latitude resolution in km, longitude resolution in km, and valid calibration types as comment cards	default=NO
BAND=	spectral band, if the image has multiple bands	
DAY=*bday eday*	day range to search	*ccyyddd* or *yyddd* format (no default)
ID=	NEXRAD station ID	NEXRAD server only
SS=*ss1 ss2*	satellite sensor number	no default
TIME=*btime etime*	time range to search	*hh:mm:ss* format (no default)
WL=	wavelength	AIRS server only
WN=	wavenumber	AIRS server only

Special rules for transmitting the image directory

If the server has problems fulfilling the request, use the following standard error values and messages found in **`~mcidas/data/ADDERROR.DOC`**. The range -12000 to -12999 lists error codes specific to image directories. Use enumerated error codes and messages when applicable and create new error codes and messages for error conditions that are unique to your site.

Converting the image directory's format

The image directory sent to the client is a 65-word directory. Word 1 contains the AREA number from the server. Words 2 through 65 are the same as words 1 through 64 in the image directory described in the *AREAnnnn* data file documentation in Chapter 6.

Transmitting image directory objects to the client

Once the data is formatted correctly, the image directory can be transmitted. Because the transmission protocol is count+data, the server will send the following information to the client:

- 4-byte value containing the total number of bytes of data for the directory being sent (260 + (80 * *NumberOfComments*))
- 4-byte value containing the file number for this image directory
- 256-byte image directory
- Comment cards, blank padded to 80 characters

The information is repeated until there are no new image directories to send. The sample code on the facing page shows you how to set up your server to transmit an image directory.

```
integer image_header(64) integer n_comments integer comment(20) integer n_send_nbo integer
total_bytes c--- loop through all image objects matching selection conditions c--- assume
readimagedirectory reads a 64-word image in area format do 10 i = 1 , n_files ok =
readimagedirectory (i, image_header) n_comments = image_header(64) c--- calculate the
number of bytes to be sent for this directory total_bytes = 260 + (n_comments * 80)
n_send_nbo = total_bytes call swbyt4(n_send_nbo, 1) call m0sxsend(4, total_bytes) c---
send the file number file = i call swbyt4(file, 1) call m0sxsend(4, file) c--- send the
image directory; assume imageheadernbo converts c--- the header to network-byte-order call
imageheadernbo(image_header) call m0sxsend(256, image_header) c--- send comment cards
```

```
backlashes readcomment reads once c--- comment card if (n_comments .gt. 0)then do 20 j = 1
, n_comments call readcomment(i,j,comment) call m0sxsend(80,comment) 20 continue endif 10
continue
```

Serving grid data

The ADDE server, **ggetserv**, processes a client request and returns complete grid objects to the client. The grid object contains the grid header and the data. Unlike the image server, which can only send one image object to the client, grid requests may include as many grid objects as desired. The McIDAS-X GRDDISP command accesses grid objects.

Grid object request syntax

The table below lists the client request syntax options for grid objects. Note that keywords 1 through 3 are placed in the request block as positional parameters.

Keywords	Description	Remarks
' ' 1	group name	
' ' 2	descriptor name	
' ' 3	maximum number of bytes that can be stored in the destination grid buffer on the client	
DAY=*day1 .. dayn*	list of model-run days	*ccyyddd* format (no default)
DERIVE=	derived parameters	
DRANGE=*bday eday dinc*	range of model-run Julian days	*ccyyddd* format
FDAY=	model-forecast valid day	*ccyyddd* format
FRANGE=*fhr1 .. fhrn*	range of model-forecast hours	
FTIME=	model-forecast valid time	*hhmmss* format
GRIB=*sgrib egrib*	range of grib numbers to get from a McIDAS-X grid file	
GRID=*sgrid egrid*	range of grid numbers to get from a McIDAS-X grid file	
LEV=*lev1 .. levn*	list of data levels to retrieve	can be a number, SFC, MSL or TRO
NUM=	number of grids to retrieve	can be a number or ALL
PARM=*p1 .. pn*	list of parameters to retrieve	
PARSE=*select1 .. selectn*	list of grid selection conditions for multiple grid parsing	format is a subset of other selection conditions specified; each selection is isolated by single quotes
PNUM=	number of parseable grids specified in the selection	
POS=	relative file position number in the dataset	
PRO=	model-projection type	

SRC=*src1 .. srcn*	list of model grid sources	
TIME=*time1 .. timen*	model run times to retrieve	*hhmmss* format
TRANGE=*btime etime tinc*	range of model-run times	*hhmmss* format for time
VERSION=	grid transmission version	value is A as of 11/96
VT=*vt1 .. vt2*	valid hour offsets	*hhmmss* format

Special rules for transmitting a grid object

You must adhere to the rules below when transmitting a grid object. The first two are specific to the client request. The last three are specific to the transmission format sent back to the client.

- If PAR=STREAML, WINDB or WINDV, you must send the u- and v-wind component.
- If DERIVE=TTIN, VOR, DVG, SPD, DIR, ABV, DSH, DST, TD, KINX, COR, BETA or ADV, you must calculate the appropriate grid before sending the results back to the client.
- If portions of the grid contain missing values, fill those locations with the value 0x80808080.
- Data must be delivered to the client in column-major format with the first data point in the upper-left corner of the grid.
- If the server has problems fulfilling the request, use the standard error values and messages found in **`~mcidas/data/ADDERROR.DOC`**. The range -21000 to -21999 lists error codes specific to grid objects. Use enumerated error codes and messages when applicable and create new error codes and messages for error conditions that are unique to your site.

Converting the grid object's format

The grid object contains a grid header and gridded data. The grid header format is described in Chapter 6 of this manual in the *GRIDnnnn* data file documentation.

Transmitting grid objects to the client

As the server sends data to the client, TCP may timeout during periods of no data transmission. To avoid timeouts, the server can send a heartbeat value (11223344) periodically to the client to keep the connection active. The heartbeat values are sent at the beginning of the transmission only. Once all the data is found and formatted, the grid object can be sent to the client.

The server sends the client the following information:

- Multiple cycles of size 4, each followed by a heart beat; the cycles of size 4 and the heartbeat are optional
- 4-byte value containing the total number of bytes in the grid objects; the number of bytes transmitted is calculated as follows: (*numgrids* *260 + (*total number of data points* * 4)) + 8;
  the data size must be sent twice
- 4-byte value containing the number of grids to send to the client
- 256-byte grid header
- Grid data object, which has a variable length, (*number of rows* *number of coulmns*)*4
- 4-byte value containing zero

The grid header, grid object, and 4-byte value containing zero are repeated until all the grid objects are sent.

The sample code below shows you how to set up your server to transmit a grid object.

```
include 'gridparm.inc' integer grid_header(64) integer grid(MAXGRIDPT) integer total_bytes
integer grid_size c--- read the grid; assume readgrid reads a grid header's grid c--- into
McIDAS format ok = readgrid(grid_header, grid) grid_size = grid_header(1) c--- send the
total number of bytes total_bytes = 256 + (grid_size * 4) + 8 temp_int = total_bytes call
swbyt4(temp_int, 1) call m0sxsend(4, temp_int) call M0sxsend (4 temp_int) c--- send the
number of grids temp_int = 1 call swbyt4(temp_int, 1) call m0sxsend(4, temp_int) c--- send
the grid header; assume gridheadernbo switches integer c--- values in the header to
network byte order call gridheadernbo(grid_header) call m0sxsend(256, grid_header) c---
send the data call swbyt4(grid_data,grid_size) call m0sxsend(grid_size*4, grid) c---send 0
separate temp = 0 call swbyt4(temp_int, 1) call M0sxsend (4,temp_int)
```

Serving grid directories

The ADDE server, **gdirserv**, processes a client request and returns grid directories to the client. The grid directory contains information about the contents of the grid. The McIDAS-X command GRDLIST is one that accesses grid directories.

Grid directory request syntax

The table below lists the client request syntax options for grid directories. Keywords 1 and 2 are placed in the request block as positional parameters.

Keywords	Description	Remarks
' ' 1	group name	
' ' 2	descriptor name	
DAY=*day1 .. dayn*	list of model-run days	*ccyyddd* format
DERIVE=	derived parameters	
DRANGE=*bday eday dinc*	range of model-run Julian days	*ccyyddd* format
FDAY=	model forecast valid day	*ccyyddd* format
FRANGE=*fhr1 .. fhrn*	range of model-forecast hours	
FTIME=	model-forecast valid time	*hhmmss* format
GRIB=*sgrib egrib*	range of grib numbers to get from a McIDAS-X grid file	
GRID=*sgrid egrid*	range of grid numbers to get from a McIDAS-X grid file	
LEV=*lev1 .. levn*	list of data levels to retrieve	can be a number, SFC, MSL or TRO
NUM=	number of grids to retrieve	can be a number or ALL
OUT=	output format	default=ALL
PARM=*p1 .. pn*	list of parameters to retrieve	
PARSE=*select1 .. selectn*	list of grid selection conditions for multiple grid parsing	format will be a subset of other selection conditions specified; each selection is isolated by single quotes
PNUM=	number of parseable grids specified in the selection	
POS=	relative file position number in the dataset	
PRO=	model-projection type to retrieve	
SRC=*src1 .. srcn*	list of model grid sources	
TIME=*time1 .. timen*	list of model-run	*hhmmss* format

	times to retrieve	
TRANGE=*btime etime tinc*	range of model-run times	*hhmmss* format for time
VERSION=	ADDE grid transmission version	value is 1 as of 11/96
VT=*vt1 .. vt2*	list of valid hour offsets	*hhmmss* format

Special rules for transmitting the grid directory

If PAR=STREAML, WINDB or WINDV, you must send the u- and v- component wind.

If DERIVE=TTIN, VOR, DVG, SPD, DIR, ABV, DSH, DST, TD, KINX, COR, BETA or ADV, you must calculate the appropriate grid before sending the results back to the client.

Converting the grid directory's format

The grid directory's format is defined in Chapter 6 of this manual in the *GRIDnnnn* data file documentation.

Transmitting the grid directory to the client

As the server sends data to the client, TCP may timeout during periods of no data transmission. To avoid timeouts, the server can send a heartbeat value (11223344) to the client periodically to maintain an active connection. The heartbeat values are sent at the beginning of the transmission only. Once all the data is found and formatted, the grid directory can be sent to the client.

The server sends the client the following information. Except the first item, all the items listed below are repeated for each grid file. Within each grid file, the grid file header and the two 4-byte values are repeated for each grid in the grid file.

- 4-byte value containing the total number of bytes to transmit
- 4-byte value of zero indicating a new file header is being sent; this information is repeated for each new file header to transfer and is sent after all grids from the previous file are transmitted
- 256-byte grid file header; this information is repeated for each new grid file header to transmit and is sent after the grids from the previous file are transmitted
- 4-byte value of zero indicating a new grid directory is being sent; this information is repeated before each grid directory in the file is sent
- 256-byte grid header; this information is repeated for each grid in the file being sent
- 4-byte value of 1 indicating no more grid directories in the file

A 4-byte value of 2, indicating no more grid files, is sent to end the transaction.

If the server has problems fulfilling the request, use the standard error values/messages found in **`~mcidas/data/ADDERROR.DOC`**. The range -22001 to -22999 lists error codes specific to grid directories. Use enumerated error codes and messages when applicable and create new error codes and messages for error conditions that are unique to your site.

The sample code below shows you how to set up your server to transmit a grid directory.

```
integer grid_header(64) integer grid_file_header(64) c--- send total byte count temp_int =
(260* n_grids) + (n_files * 260) call m0sxsend(4, temp_int) c--- loop through the grid
file headers do 10 file = 1 , n_files c--- read the grid file header; assume
readfileheader reads c--- a grid file header in McIDAS grid file format call
readfileheader(file, grid_file_header, n_grids) c--- send the value 0 indicating that a
grid file was found temp_int = 0 call m0sxsend (4, temp_int) c--- send the grid file
header; assume gridfileheadernbo c--- switches integer words to network byte order nbytes
= 256 call gridfileheadernbo(grid_file_header) call m0sxsend(nbytes, grid_file_header) do
20 i = 1 , n_grids c--- read the grid header; assume readgridheader reads a c--- grid in
McIDAS grid format call readgridheader(file,i,grid_header) c--- send the value 0
indicating that a grid file was found temp_int = 0 call m0sxsend (4, temp_int) c--- send
the grid header; assume gridheadernbo switches c--- integer words to network byte order
nbytes = 256 call gridheadernbo (grid_header) call m0sxsend(nbytes, grid_header) 20
continue c--- no more data from this grid file temp_int = 1 call swbyt4(temp_int, 1) call
m0sxsend (4, temp_int) 10 continue c--- no more data temp_int = 2 call swbyt4(temp_int, 1)
call m0sxsend (4, temp_int)
```

Serving point data

The ADDE server, **mdksserv**, processes a request and returns point data to the client. In addition to the data values, the server also sends information about the units, scaling factor, and name of each parameter returned to the client. The McIDAS-X command PTLIST accesses point data.

Point data request syntax

The table below lists the client request syntax options for point data. The table below lists the client request syntax options for image directories. Note that keywords 1 through 2 are placed in the request block as positional parameters

Keywords	Descriptions	Remarks
'' 1	group name	
'' 2	descriptor name	
MAX=	maximum number of matches to find given the selection conditions	
POS=	file position number in a dataset	
SELECT=	data selection clause	see comments below
TRACE=	trace file activation	set to 1 to activate tracing
VERSION=	transmission version number	value is 1 as of 11/96

Special rules for transmitting point data

The client can request point data either by specifying no specific parameter names, which sends all the parameters to the client, or by specifying only the parameter names of interest. Because the user may ask for an unlimited number of individual parameters, the parameter list is sent after the client request string. Thus, a server must make additional calls to **M0sxread** to get the list of 4-byte, blank-padded parameter names. You can also call the **McExtractPointRequest** function to extract the client request and put it in the C structure PTREQUEST.

The SELECT clause contains the entire data selection clause format. You can use as many selection clauses as needed for each request. The selection clause format is described in the section titled *Reading point objects: Defining selection conditions* in *Chapter 5, Accessing Data*.

Transmitting point data to the client

When the point data is formatted correctly, it can be sent to the client. The server sends the following information:

- 4-byte value containing the number of bytes of parameter names to be sent
- Character string of NULL-separated parameter names
- 4-byte value containing the number of unit-string bytes to be sent
- Character string of NULL-separated units
- 4-byte value containing the number of bytes of scaling factors to be sent
- Array of integer scaling factors for each parameter
- 4-byte value containing the number of bytes of data to be sent
- *n* bytes of data

The last two pieces of information are repeated until there is no more data to be transmitted. Then the 4-byte value containing the number of bytes of data to be sent will be zero.

If the server has problems fulfilling the request, use the standard error values and messages found in **`~mcidas/data/ADDERROR.DOC`**. The range -31000 to -31999 lists error codes specific to point data. Use enumerated error codes and messages when applicable and create new error codes and messages for error conditions that are unique to your site.

The sample below shows how to set up your server to transmit point data.

```
parameter (N_KEYS = 4) integer buffer(N_KEYS) character*256 keys character*256 keys
integer scales(N_KEYS) integer length character*1 NULL data NULL=/'0'/ integer temp c---
send four parameters to the client: station ID, temperature c--- in Celsius, dew point in
Kelvin and wind direction keys='ID'//NULL//'T'//NULL//'TD'//NULL//'DIR'
units='CHAR'//NULL//'C'//NULL//'K'//NULL//'DEG' scales(1) = 0 scales(2) = 2 scales(3) = 2
scales(4) = 0 c--- send the key names length = lentrim(keys) temp=length call
swbyt4(length, 1) call m0sxsend(4, length) call m0sxsend(temp * 4, keys) c--- send the
```

```
units length = lentrim(units) temp=length call m0sxsend(4, length) call m0sxsend(temp * 4,
units) c--- send the scalings call swbyt4(n_vals, 1) call m0sxsend(4, n_vals) call
m0sxsend(N_KEYS, scales) 10 continue c--- read the new data; assume readnewdata reads a
record of c--- data values ok = readnewdata(buffer) if (ok .eq. 0)then c--- flip the temp,
dewpt and direction call swbyt4(buffer(2), 3) call m0sxsend(4, n_vals) call
m0sxsend(N_KEYS * 4, buffer) goto 10 endif
```

Serving weather text data

The ADDE server, **wtxgserv**, processes a client request and sends back the text header containing information about the data and the actual weather text data. The McIDAS-X command WXTLIST is one command that accesses weather text data.

This server is delivered with the McIDAS-XCD package.

Weather text request syntax

The table below lists the client request syntax options for weather text data.

Keywords	Description	Remarks
APRO=	AFOS/AWIPS product headers to match	three characters; don't use APRO with the WMO keyword
ASTN=	AFOS/AWIPS stations to match	two or three characters
DAY=	most recent day to begin the search	*ccyyddd* format (default=current day)
DTIME=	maximum number of hours back in time to search	no default
MATCH=	list of character match strings to find from the text	
NUM=	number of matches to find	default=1
PROD=	predefined product name	
SOURCE=	circuit source	default=ALL
WMO=	WMO product headers to match	at least two characters; wildcard characters are allowed; don't use WMO with the APRO keyword
WSTN=	WMO stations to match	four characters

Special rules for transmitting weather text

If the day specified does not equal the current day, the search begins from the end of the day instead of the current time.

Text data is sent to the client as a blank-padded buffer.

The valid standard error values and messages for weather text are located in the range -45000 to -46999. See the **wtxgserv** source code for error message descriptions.

Converting the text header's format

The information in the table below must be included with each text header sent to the client.

Bytes	Description	Remarks
0 - 3	circuit source	blank-padded character string
4 - 7	number of bytes in the data section	

8 - 11	file address where the data is located	usually not important to the client
12 - 15	time the data was ingested	*hhmmss* format
16 - 19	four-character WMO header	product code and country code
20 - 23	WMO product number	integer value
24 - 27	WMO station origin	four-character ICAO ID
28 - 31	AFOS product header	three characters
32 - 35	AFOS product location	two or three characters
36 - 39	AFOS product issuing station	three characters
40 - 43	Julian day of the data	*ccyyddd* format
44 - 47	number of bytes per line	usually 80
60 - 63	FAA catalog number	

Transmitting weather text data to the client

Once all the data is found and formatted, the weather text can be sent to the client. Because TCP may timeout during periods of no data transmission, a heartbeat value (11223344) can be sent to the client periodically client maintain an active connection.

The server sends the client the following information:

- 4-byte value containing the length of the client request string; this lets users know how their request was expanded when the PROD keyword is specified
- Expanded client request string
- Heartbeat value if needed
- 4-byte value containing the total number of bytes of data for this text block, including the 64-byte header and the text
- 64-byte text header
- *n* bytes of 80-character text, blank padded

The last three pieces of information are repeated until no more data is found.

The sample code below shows you how to set up your server to transmit weather text data.

```
char *actual_request; int text_header[16]; char *text; int req_length; int temp_int; /*
send back the client request */ req_length = strlen (actual_request); temp_int =
req_length; M0swbyt4 (&temp_int, 1); M0sxsend (4, &temp_int); length = 1; while (length >
0) { int total_bytes; /* read the length of the next text block */ length =
ReadNewTextBlock (text_header, &text); if (length > 0) { total_bytes = sizeof
(text_header) + length; /* send the total data block length */ M0swbyt4 (&total_bytes, 1);
M0sxsend (4, &total_bytes); /* send the text header */ M0sxsend (sizeof (text_header),
text_header); /* send the text */ M0sxsend (length, text); free (text); } else {
temp_int=length; M0swbyt4 (&temp_int,1); M0sxsend (4, &temp_int); } }
```

Serving observational weather-text data

The ADDE server, **obtgserv**, processes a client request and returns the text header containing information about the retrieved data, along with the observational weather-text data. The McIDAS-X command OBSRPT is one that accesses observational weather-text data.

This server is delivered with the McIDAS-XCD package.

Observational weather-text request syntax

The table below lists the client request syntax options for observational weather-text data.

Keywords	Description	Remarks
CO=*co1 .. con*	list of countries	2- character country IDs are read from COUNTRY.DAT, which is provided with McIDAS-XCD
ID=*id1 .. idn*	list of stations	
IDREQ=	ID request type	must be set to LIST if CO, REG, or ID is specified; if a LATLON option is added to this server, specify GEO
NEWEST=*day hour*	most recent observation time to allow in the request	default=most recent observation filed
NHOURS=	maximum number of hours back in time to search from the value in NEWEST	no default
NPERIOD=	number of time periods to list	varies among data types
NUM=	number of matches to find for each station	default=1
OLDEST=*day hour*	oldest observation time to allow in the request	
REG=*reg1 .. regn*	list of station regions	the regions list is stored in GROUPS.DAT, which contains the stations for a particular state/province
TYPE=	numeric value for the type of observation	varies among data types

Special rules for transmitting observational weather text

Don't specify both NUM and NPERIOD, or NPERIOD and NHOURS in the same request. If you specify NHOURS, you must specify NEWEST.

The valid standard error values and messages for observational weather text are located in the range -48000 to -48999. See the **obgtserv** source code for error message descriptions.

Converting the text header's format

The information in the table below must be included with each text header sent to the client.

Bytes	Description	Remarks
0 - 3	version number	currently zero
4 - 7	observation type number	varies among data types
8 - 11	active observation flag	0=inactive, 1=active
12 - 15	starting observation day	*ccyyddd* format
16 - 19	starting observation time	*hhmmss* format
20 - 23	starting observation hour	*hhmmss* format
24 - 27	ending observation day	*ccyyddd* format
28 - 31	ending observation time	*hhmmss* format
32 - 35	ending observation hour	*hhmmss* format
36 - 39	ID type flag	1 if ICAO ID

40 - 47	station ID	*ccyyddd* format
48 - 51	number of bytes of data	
52 - 55	number of lines of data	
56 - 95	reserved for future use	

Transmitting observational weather-text data to the client

Once the data is formatted, the text data can be sent to the client. The server sends the client the following information:

- 4-byte value containing the length of the text header, which is currently 96; when all the data is sent, this value is reset to zero
- 96-byte text header
- 4-byte value containing the number of bytes of text to be sent
- *n* bytes of text; 80 characters per line, blank padded

This information should be repeated until no more data is found.

The sample code below shows you how to set up your server to transmit observational weather text data to the client.

```
integer header(24) integer n_bytes integer n_bytes_nbo character*80 line(1000) 100
continue ok = readnewdata(header, line) if (ok .eq. 0)then n_bytes = 96 n_bytes_nbo =
n_bytes call swbyt4(nbyte_nbo, 1) call m0sxsend(4, n_bytes_nbo) call m0sxsend(n_bytes,
header) n_bytes = header(13) n_bytes_nbo = n_bytes call swbyt4(n_bytes_nbo, 1) call
m0sxsend(4, n_bytes_nbo) call m0sxsend(n_bytes, line) goto 100 else n_bytes = 4
n_bytes_nbo = n_bytes call swbyt4(n_bytes_nbo, 1) call m0sxsend(4, n_bytes_nbo) call
m0sxsend(4,0) endif
```

Appendix A

Guidelines for Writing Helps

This appendix provides the guidelines to follow when creating or modifying a help to meet the McIDAS-X core standards. You will learn:

- General guidelines to use for all command helps
- Specific guidelines to use for parameters, keywords and remarks

A sample help for the McIDAS-X command ZA is also included. See the *McIDAS User's Guide* and online command helps for additional examples.

This appendix consists of these sections:

General guidelines for all helps

Confirm that all spelling and punctuation are correct.

Verify that your help contains the sections listed below. Only include examples if the help is very short.

- Command name with functional description
- Command formats
- Parameters
- Keywords
- Remarks

Keep the length of each line to 78 columns or less. This will accommodate the width of the default McIDAS-X Text and Command Window, which is formatted to 80 columns.

Verify that the first line of the help contains the command name, one or two hyphens and an accurate functional description. For example:

```
SFCMG - Displays a surface meteorogram -or- AVGI -- Reduces image resolution by averaging
```

Use a series of 10 hyphens to indicate the end of the help. For example:

```
C ? ----------
```

Don't include the owner's initials in the help.

Don't put tabs in the help. The programs that make the help files will skip lines with tabs, resulting in help files with missing information. To check for tabs in **`vi`**, use **`:set list`**.

See the following commands as examples if you're writing a complex help: MDX, SCFPLOT, BATCH, ZA, SFCMG, LVF, IMGCOPY, IMGOPER and GRDDISP.

Specific guidelines

Parameters and keywords

Check existing helps when naming parameters and keywords, since many McIDAS-X commands share parameter and keyword names and definitions. If you use a parameter or keyword with the same meaning as one in another core program, use the same name. For example, use the parameter *day* and keyword DAY instead of *date* or DATE.

Choose parameter and keyword names that are unambiguous. For example, *sarea* and *darea*, for source area and destination area, are better choices for parameter names than *area1* and *area2*, or *source* and *dest*.

Don't write parameter and keyword descriptions as complete sentences. If the description has more than one comment, link them with semicolons.

Place default values for the parameters and keywords in parentheses at the end of the line using this format: (def=*value1 value2 ...*).

Use a vertical bar character (|) between the end of a parameter name or keyword name and its description. If possible, align these vertical bars. The vertical bar is required by the help program to list the parameters and keywords when a user requests an abbreviated help by pressing **Alt ?**.

Parameters only

List parameters individually, in the order they appear in the command formats, not alphabetically.

Specify a parameter name in uppercase only if it is the exact value that the user will enter; for example, the ADD, DEL, LIST, and HOST options of the DATALOC command. Otherwise, parameter names are lowercase.

Keywords only

Alphabetize keywords. If a command has many keywords and you group them by function. Alphabetize them within each group. See the GRDDISP and IMGDISP commands.

Specify each keyword with its required letters in uppercase and optional letters in lowercase, followed by an equal sign, a space, a vertical bar, another space, and then the description. In the example below, COLor= means the user can enter COL=, COLO=, or COLOR=.

```
COLor= | graphics color level for contours; use positive numbers for solid, negative numbers for dashed (def=2)
```

If a keyword has several values, don't repeat the keyword name on every line. Simply list subsequent values below the first one. For example:

```
FORmat=ID | plot the 3- to 5-character ID at each station location; see the Remarks (def) =IDN | plot the 5-digit ID number at each
station location =NAME | plot the station name at each station location =POINT | plot a point at each station location
```

If a keyword accepts many values or a range of values, use the format shown below.

```
LEV=l1..ln | copies grids with these levels; for example, SFC, 1000, 850 BAND=b1..bn | applies the function to the specified image
bands (def=8 for GOES IR, 4 for POES)
```

Remarks

Place remarks in the order of importance to the user. Since space is limited in online helps, include only those remarks that help the user understand how the program works or provide a useful suggestion.

Write the remarks section in complete sentences with proper punctuation. A remark can be several sentences long. Do not indent the beginning of the remark; instead, separate the remarks with a single blank line.

Sample help

Help for the ZA command is provided below as an example. Also look at the helps for the DSINFO, DATALOC, GRDINFO, GRDCOPY, MFPLOT, NAVDISP and TXT2MD commands. They contain a variety of standard parameters and keywords and are good examples of how to write your helps.

ZA command

```
C ? ZA -- Writes text and/or symbols on the graphics frame C ? ZA color height justify
"text C ? Parameters: C ? color | graphics color level; 0 for erase (def=1) C ? height |
pixel height to draw text/symbols (def=10 for text; cursor C ? height for TCYC= symbol) C
? justify | text justification; specify one of the following options: C ? LA - left
justified above center of cursor C ? LC - left justified at center of cursor C ? LB - left
justified below center of cursor (def) C ? CA - center justified above center of cursor C
? CC - center justified at center of cursor C ? CB - center justified below center of
cursor C ? RA - right justified above center of cursor C ? RC - right justified at center
of cursor C ? RB - right justified below center of cursor C ? M - mouse justified; see the
Remarks C ? N - same as CC option, but allows numerous plots of the C ? text/symbols by
pressing mouse buttons; see Remarks C ? V - vertically aligned; the upper-left corner of
the C ? first letter or symbol is placed at the cursor center; C ? subsequent letters and
symbols are written below C ? "text | text to write on the graphics frame; the following
placeholders C ? can be specified for text substitution using information from the C ?
frame, dataset or displayed image: C ? (FRAME) - frame number of display, nnn C ?
(DATASET) - ADDE dataset name of image, 24 char max C ? (POS) - ADDE dataset position of
image, nnnn C ? (BAND) - band number of image, nn C ? (WL) - wavelength of image, nn.nn
unit C ? (WN) - wave number of image, nn.nn unit C ? (FREQ) - frequency of image, nn.nn
unit C ? (ID) - NEXRAD station ID of image, 3 char C ? (MEMO) - memo field of image, 32
char max C ? (SS) - sensor source number of image, nnn C ? (STYPE) - image type of image,
12 char max C ? (ERES) - element resolution of image, nn.nn C ? (LRES) - line resolution
of image, nn.nn C ? (RES) - resolution of image, nn.nn C ? (EGEORES) - geographical
element resolution of image in km, n.nn C ? (LGEORES) - geographical line resolution of
image in km, n.nn C ? (GEORES) - geographical resolution of image in km, (n.nn X n.nn KM)
C ? (DAY) - day of image, dd mon yy C ? (MDY) - day of image, mon dd yy C ? (JDAY) -
```

Julian day of image, ccyyddd C ? (TIME) - time of image, hhmmss C ? (HH) - time of image, hh C ? (HHMM) - time of image, hh:mm C ? (HHMMSS) - time of image, hh:mm:ss C ? "$text | displays weather symbols or arrows on the graphics frame; see C ? the McIDAS User's Guide for valid choices C ? Keywords: C ? Ang= | angle to write text; do not use with the FONT command or global C ? keyword, or keywords ENCL or TCYC C ? Encl=C | draws a circle around the text in the N mode C ? =E | draws an ellipse around the text in the N mode C ? FULlframe=YES | use the entire frame for output, regardless C ? of the frame's panel configuration; do not use C ? FUL=YES with the global keyword PANEL C ? =NO | limit output to the current panel or the panel C ? specified in the global keyword PAN (def) C ? Gra= | graphics frame number (def=current) C ? IMA= | image frame number to use for determining values of placeholders C ? in "text parameter (def=current) C ? POS=lin ele | TV line and element position to draw text, symbols or C ? arrows (def=current cursor position) C ? TCYc=color hemi | plots a hurricane or tropical storm symbol centered C ? on the cursor or the position specified in the C ? POS keyword, where: C ? color - graphics color level of the symbol's center; C ? specify X to leave the center blank C ? hemi - N or S for northern or southern hemisphere; C ? determines direction of the spirals (def=S) C ? Wid= | graphics line width, in pixels; the maximum is 64 (def=current) C ? Remarks: C ? In the "justify" parameter: C ? - left justified (LA, LC and LB options) means the text begins at the C ? cursor position and extends to the right C ? - right justified (RA, RC and RB options) means the text ends at the C ? cursor position and extends to the left C ? - center justified (CA, CC, CB and N options) means the text is C ? horizontally centered at the cursor position so it extends equal C ? distances to the left and right. C ? C ? The (GEORES), (LGEORES) and (EGEORES) placeholders can be used only C ? if GEORES=YES was specified in the IMGDISP command that displayed C ? the image. The resolution values, listed as "n.nn" in the Parameters C ? section above, are in kilometers and are displayed in the format 0.nn, C ? n.nn, nn.n or nnn (depending on the magnitude of the resolution). C ? The two resolution values displayed with the (GEORES) placeholder C ? are the same as (LGEORES) and (EGEORES), in that order. C ? C ? If you choose the M(ouse) option, clicking the middle mouse button C ? repositions the start of the text at the cursor location. Clicking the C ? right button repositions the end of the text at the cursor location. C ? The height is scaled as needed. Using the FONT command or global keyword C ? with the M option positions the text horizontally at various sizes. You C ? can't change the angle of the text. C ? C ? If you choose the N(umerous) option, clicking the middle mouse button C ? draws the text, centered at the cursor location. Clicking the right C ? button erases the last drawn text. C ? C ? When ENCL= is used, the practical maximum number of characters is 3. C ? ----------

Appendix B

Satellite Information

This appendix contains links to the current SSEC-assigned sensor source numbers used in McIDAS-X. It also contains links to the current band information for satellite data ingested in real time by the SSEC Data Center servers.

Sensor source numbers

For a current list of Sensor Source numbers, see **Appendix C: Sensor Source Numbers**, in the McIDAS User's Guide.

Bands for satellite imagery

For a current description of the bands for satellite data ingested in real time by the McIDAS-X servers, see **Appendix D: Bands for Satellite Imagery**, in the McIDAS User's Guide.

The tables containing the band information use terms that you may be unfamiliar with. They are defined below.

- The *band* is the number given to the wavelength channel of the satellite data.
- The *central wavelength* is the wavelength falling in the middle of the range for the given band.
- The *absorbing gases* column lists the primary gases that absorb and emit radiation at the central wavelength. The term *window* used in this column refers to a channel that is transparent to atmospheric radiation.
- The *sounding level* is the average level where most of the radiant energy sensed by the satellite originates, assuming a cloud-free atmosphere.
- The column labeled *meteorological objective* or *primary use* provides examples of how a band is used in physical applications.
- The term *resolution* refers to the maximum resolution of the instrument, which is not always the same as the resolution of the ingested data in the image file.

For more information about the format of the satellite image data, see the *AREAnnnn* data file described in *Chapter 6,* Format of the Data Files.

See the image-specific characteristics section in Chapter 6 for aspects of McIDAS-X area files that are specific to their image type.

Appendix C

Common Data Parameter Names

The table provided in this appendix lists common data parameter names used in McIDAS-X, along with a description of each.

Parameter	Description
ACFT	aircraft type (ASCII)
ASIZ	aircraft size
CA	cloud amount; octants of the sky covered
CAI1	cloud amount of the first layer
CAI2	cloud amount of the second layer
CAI3	cloud amount of the third layer
CAI4	cloud amount of the fourth layer
CC	cloud cover
CC1	cloud cover; first cloud layer non-ceiling
CC2	cloud cover; second cloud layer non-ceiling
CCH	high cloud cover (0=CLR, 1=SCT, 2=BKN, 3=OVC)
CCL	low cloud cover (0=CLR, 1=SCT, 2=BKN, 3=OVC)
CCM	middle cloud cover (0=CLR, 1=SCT, 2=BKN, 3=OVC)
CG1	genus of the first cloud (ASCII)
CG2	genus of the second cloud (ASCII)
CG3	genus of the third cloud (ASCII)
CG4	genus of the fourth cloud (ASCII)
CH	type of high level clouds (ASCII)
CIG	categorical forecast of the ceiling height
CIGC	cloud cover of the ceiling
CIGH	ceiling height
CL	type of low level clouds (ASCII)
CM	type of middle level clouds (ASCII)
CMAX	number of columns containing data
CO	country ID (ASCII); see the McIDAS-X CCODE command for more information
DAY	year and Julian day
DIR	wind direction
FLV1	first flight level

FLV2	second flight level
GUS	wind gusts
HMS	actual time of the observation
ICE1	first icing code
ICE2	second icing code
ICTP	icing type (ASCII)
ID	station identifier (ASCII)
IDA	part one of a two-part ID (ASCII)
IDB	part two of a two-part ID (ASCII)
IDN	station ID number
IFQ1	first icing frequency (ASCII)
IFQ2	second icing frequency (ASCII)
IL11	first icing base
IL12	first icing top
IL21	second icing base
IL22	second icing top
LAT	latitude, +North (-90 to +90)
LEV	level: SFC, TRO, or same as P (ASCII)
LLWS	low level wind shear
LON	longitude, +West (-180 to +180)
MOD	modification flag (1=modified)
NREC	number of records in a row
OBSV	categorical forecast of obstructions to vision
P	atmospheric pressure
P1	temperature for significant temperature level
P1	wind direction for significant wind level
P2	dew point temperature for significant temperature level
P2	wind speed for significant wind level
P3	pressure for significant temperature level
P3	height for significant wind level
PC	characteristic of pressure tendency
PCP	3-hour precipitation total

PP06	6-hour probability of precipitation
PP12	12-hour probability of precipitation
PRE	pressure
PSD	direction the predominant swell comes from
PSH	height of the predominant swell
PSL	sea level pressure
PSNO	probability of snow
PSP	period of predominant swell
PST	station pressure
PT	3-hour pressure change
PTIM	time period for precipitation
PTYP	precipitation type (ASCII)
PZR	probability of freezing rain
QP06	6-hour quantitative precipitation forecast
QP12	12-hour quantitative precipitation forecast
SN06	6-hour snow probability
SN12	12-hour snow probability
SNO	cumulative snow depth
SPD	wind speed
ST	state ID (ASCII)
STI	insitu sea surface temperature
SV06	6-hour severe weather probability
SV12	12-hour severe weather probability
SWH	height of sea or wind waves
SWP	period of sea or wind wave
T	temperature
TBTP	turbulence type (ASCII)
TD	dew point temperature
TFQ1	first turbulence frequency (ASCII)
TFQ2	second turbulence frequency (ASCII)
TH06	6-hour thunderstorm probability
TH12	12-hour thunderstorm probability
TIME	time

TL11	first turbulence base
TL12	first turbulence top
TL21	second turbulence base
TL22	second turbulence top
TMAX	maximum temperature
TMIN	minimum temperature
TRB1	first turbulence intensity
TRB2	second turbulence intensity
TYPE	type of data
VDAY	Julian day of forecast verification
VIS	visibility
VTIM	time of forecast verification
WX1	weather - first four characters (ASCII)
WX2	weather - second four characters (ASCII)
WXP	present weather type (ASCII)
Z	height above sea level
ZC1	base of the first cloud layer
ZC11	first cloud base
ZC12	first cloud top
ZC2	base of the second cloud layer
ZC21	second cloud base
ZC22	second cloud top
ZC3	base of the third cloud layer
ZC4	base of the fourth cloud layer
ZCB	height of the cloud base
ZCL1	height of the first non-ceiling
ZCL2	height of the second non-ceiling
ZS	surface elevation

Appendix D

POES AVHRR Calibration Information

This appendix contains the following additional POES AVHRR calibration information resulting from the launch of NOAA-15 satellite in May 1998.

AVHRR Calibration Background

Since NOAA-12 and -14 AVHRR use the older TIRO calibration while the NOAA-15 AVHRR uses the newer AVHR calibration, changes have been made in McIDAS-X area structure between the NOAA-12 and -14 areas and the NOAA-15 areas. All bands of the NOAA-15 AVHRR/3 has calibration that differs from NOAA-14 AVHRR, but most of the changes are internal to the area structure and calibration module and are transparent to McIDAS-X users.

All POES images are now produced and delivered only through SDI (SSEC Desktop Ingestor). Areas created from NOAA-15 data will have the AVHR calibration type; areas created from NOAA-12 and -14 data will have the TIRO calibration type. The AVHR calibration exactly duplicates the output of the TIRO calibration in all respects. For NOAA-12 and NOAA-14, the TIRO and AVHR calibration appears interchangeable.

The advantage of the new calibration type is that only relatively minor changes need to be made once to SDI software (assuming the raw data stream format does not change). Any further changes required by NOAA-15 orbit performance is made only in the AVHR calibration module, rather than the SDI ingestor. The TIRO and QTIR calibration modules will gradually become obsolete; they will be used only for archived data.

SDI can now deliver NOAA-12 and -14 ADDE areas in either the TIRO or AVHR format; a logical switch has been installed to produce either calibration type. Only the AVHR format can be delivered by SDI for NOAA-15. For those installations that do not use SDI, an updated **`SATBAND`** file, an updated **`kbprep.f`**, and **`kbxavhr.dlm`** still will not be able to manipulate NOAA-15 imagery,because **`kbxtiro.dlm`** will not be upgraded.

TIRO and AVHR Differences

The changes from TIRO formats for AVHR are summarized below:

1. The TIRCAL line-by-line calibration code is now moved to the SDI. This allows retension of the use of the platinum resistance thermometers (PRT) and samples from target and space looks, but it identifies channel three as either a near IR or thermal IR sensor (using bit 10 of word 7 in the HRPT minor frame). It renames channel three to channel six if it is identified as near IR by bit 10 of word 7, which can switch while going from one line to another, but normally switches only twice per orbit.

   Following a switch of bit 10 in word 7, there may be one or two bad lines in the raw data, and up to 10 bad lines of poorly calibrated data (worst case), due to the five-point subcommutation of the PRT temperatures between reference values. These bad lines appear totally black or totally white. After a reference line is encountered, it takes four more lines to acquire all of the PRT temperatures and to calculate their weighted average. If this is done *on-the-fly*, there is no way to recover any bad or poorly calibrated lines already sent. However, if you use post-processing of a completed area, it is possible to backfill the erroneous line prefix data.

   Each image still contains five bands or less, but they are numbered 1,2,3,4,5 or 1,2,6,4,5 with corresponding changes in the line prefix LEV section. All calibration constants that are generated by SDI and passed in the line prefix CAL section are converted to scaled integers with scaling of 100,000 for slopes and offsets (except for the band 3 slope, which is scaled by 10,000,000) and a scaling of 100 for temperatures.

   The DOC section content is unchanged, as compared to XSD; it is copied just as received, but left-shifted five bits. The length of the DOC section does not change with NOAA-15. For each line, the DOC section captures the raw data in an HRPT minor frame that does not constitute imagery that is converted to interleaved pixels.

2. Prior to NOAA-15, every AVHRR channel had a linear calibration that required a single gain and a single offset constant in the line prefix for each channel. With NOAA-15, this changes to a bi-linear calibration for the visible (1,2) and near IR (6) channels, which requiries two slopes and two offsets (four constants and two conversion equations) for each visible and near IR channel identified in the line prefix.

3. The three thermal IR channels (channels 3, 4 and 5) from the AVHRR/3 instrument have a new non-linear correction method applied, which requires a constant from the line prefix CAL section that contains the average space count. TIRCAL used this constant to calculate the slope-

offset constants for previous AVHRR data in bands 3,4, and 5. Since the NOAA-15 raw data stream continues to look exactly like NOAA-12 and NOAA-14, the slope-offset calculations currently done in TIRCAL are preserved in SDI for all AVHRR data, including AVHRR/3. This permits the AVHR calibration module to be used for NOAA-12 and NOAA-14 data from SDI as well as for NOAA-15 data. At SSEC, all POES data has been switched from TIRO to the new AVHR calibration type within SDI after NOAA-15 began operation; TIRO is calibration used only as a fall-back position.

An inverse Planck function developed by SSEC is used for converting to emission temperature in both SDI and kbxavhr.dlm.

4. The line prefixes for AVHR calibration data have been reformatted to allow eight constants per line for each of the channels (160 bytes total) instead of two constants (40 bytes) for TIRO. For the visible channels, the first four constants parameterize the bilinear calibration, while the last four constants are unused. For the IR channels, we use only the first two constants for linear calibration. The fifth and sixth constants are five scan averages of the space look and internal target. The internal target temperature is now redundantly written into the CAL section of the line prefix for bands 3,4, and 5 (at constant position 7) instead of overwriting the channel 3 patch temperature and spare word. The eighth constant is an unused spare in all channels. Note that the scaling factors for the constants are also different from TIRO. These line prefix changes will make the AVHR calibration type images incompatible with TIRO calibration.

5. The CAL=QUICK option is no longer available for AVHR.

Glossary

Miscellaneous, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

Miscellaneous

0-based
A counting sequence that begins with zero.

1-based
A counting sequence that begins with one.

A

ADDE
Abstract Data Distribution Environment software in McIDAS-X that lets a workstation act as a client, efficiently accessing data from multiple McIDAS-X servers.

alias
A short, user-defined name representing an ADDE dataset name; for example, the alias GV1 could represent the dataset name SSEC-RT/GOES8-1KVIS.

ancillary data
Additional information needed to identify, quantify and manipulate data; for example, directory, navigation and calibration blocks.

API
Application Program Interface.

applications program
A program that runs from the McIDAS-X command line.

area
The McIDAS-X image file format.

ASCII file
American Standard Code for Information Interchange file containing only text; for example, schema definition files and scripts.

asynchronous
McIDAS-X commands that run asynchronously will return control to the original calling program before they have run to completion. Also see synchronous.

B

band
The spectral channels measured by a scanning instrument; for example, band 4 for the GOES-9 imager is 10.7 microns (infrared). Also see spectral band.

band map
Region of the line prefix that contains an ordered list of the spectral bands comprising the data portion of an image line.

big-endian
Used interchangeably with network-byte-order to mean the most significant byte in a word comes first; opposite of little-endian where the least significant byte comes first.

binary file
A file containing binary information; for example, areas and executable programs.

blank-padded
Describes the practice of replacing unused characters at the end of a string with spaces.

block
A collection of data records.

blow down
To decrease image resolution by sampling or averaging data. For example, a blowdown of two drops out every other data point along the line and every other line in an image.

blow up
To increase image resolution by replicating data point values, much like enlarging a 3 x 5 photograph to an 8 x 10.

buffer
Any memory storage.

byte
An 8-bit memory segment; a 16-bit memory segment is called a half-word; a 32-bit memory segment is called a word.

C

calibration
The conversion of data values received from an instrument to useful, physical quantities such as temperature, radiance or albedo.

calibration block
The image object block that holds the information for transforming image data from its internal quantities to more common physical quantities, such as radiance or albedo.

calibration module
A group of subroutines that are specific to a type of image data; this module is used to perform a calibration.

celestial coordinates
Identical to earth coordinates except the x-axis passes through the longitude of vernal equinox rather than the prime meridian so that the celestial system is fixed relative to the stars. The transformation from celestial to terrestrial involves a single axis rotation about the z-axis, equivalent to a scalar shift in longitude. Satellite orbital predictions are typically made in a celestial system.

client
The workstation in a distributed system that initiates a request, then receives and displays the requested data.

client routing table
The table that holds the list of group names configured by the user with the McIDAS-X DATALOC command.

clipping region
See viewport.

command line
The command or series of commands entered in the McIDAS-X Text and Command Window. It may consist of positional parameters, keywords and quoted text. In McIDAS-X, the number of characters permitted in a command line is workstation dependent, although there is no practical limit.

comment block
An image object block containing a variety of textual information, such as a list of commands run on the image object to-date.

compatibility library
A file where obsolete McIDAS-X library functions are placed for one year when they are no longer referenced by any core programs. Also see local library and McIDAS-X library.

conformal projection
A map projection in which angles are preserved; for example, parallels of latitude and meridians of longitude intersect at right angles. McIDAS-X supports Mercator, Lambert conformal, polar stereographic and tangent- cone conformal projections.

connection
The initialization that occurs in a distributed system when a client determines the location of the dataset server and 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.

contrast stretching
The process of changing an image's gray scale to emphasize a feature for analysis; for example, thunderstorm cloud tops. Unlike data stretching, contrast stretching does not change the image data values.

coordinate systems
The four systems used to define the location of data points within an image; they include image, file, earth and frame coordinates. A fifth coordinate system, called world coordinates, is used with graphics.

cursor
The mouse-driven, highlighted mark that appears on the McIDAS-X display. Users manipulate the cursor to interact with McIDAS-X commands and the McIDAS-X Image Window. Several cursor sizes, types and colors are available.

D

data block
The block containing the actual data values.

data point
A collection of one or more bytes.

data point size
The number of bytes needed to accurately represent a data point; usually 1, 2 or 4 bytes.

data stretching
The process of changing an image's gray scale by stretching image data values to brightness values. To stretch image data values, a table defining the values to stretch must be created with the McIDAS-X SU command.

dataset
A collection of one or more files with a common format; for example, one dataset may contain image data, while another dataset may contain point data.

dataset name
The name used by the ADDE server to identify the type of data the user wants to access and the range or names of files to search. It consists of a group name and a descriptor name separated with a slash, such as SSEC-RT/GOES8-1KVIS.

decoder
The 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.

default
The parameter value accepted by the program if the user doesn't specify a value. To use the default for a positional parameter, the user types the letter X in the command line.

descriptor
The name used to reference a dataset in ADDE; for example, a dataset of images containing GOES-7 visible data at 4-km resolution might have the descriptor G7-VIS-4K.

directory block
An image object block containing the list of ancillary information about the image, such as the number of lines and data points, the satellite ID and the number of spectral bands.

disk file
McIDAS-X file for storing information that applications can randomly access by byte address using standard system library calls. Formerly called LW (Large Word) array files.

display
The device used to output image and graphical data in McIDAS-X; usually a workstation monitor or an X Terminal.

distributed data system
A computing system in which data is received, processed and stored, and then distributed among multiple workstations. Data can be received and processed on the same machines that store and serve it.

DLL
Dynamic Link Library; the library used in dynamic linking. OS/2 has true dynamic linking, while Unix modules are statically linked only giving the appearance of dynamic linking.

double precision
A two-word storage representation for floating-point numbers.

dynamic linking
Subprograms loaded at run time.

E

earth coordinates
A coordinate system having its origin at the Earth's center, its x-axis through the intersection of the equator and prime meridian, its z-axis through the north pole, and its y-axis completing a right-handed system. Locations in this system may be Cartesian (distances x, y and z from the origin along each axis) or spherical (a distance from the origin or a reference radius, and two angles from the x-axis). The most common spherical form is longitude, geodetic latitude and height above the reference geoid.

element
The image coordinate that makes up each division of the image along a scan line. Elements run vertically up and down the frame; they are numbered left to right with the leftmost element numbered one.

environment
The place where applications programs reside, along with McIDAS-X resident programs and shared memory.

equal-area projection
A projection in which areas are preserved; two equal areas on the Earth are also equal on the projection, even though their shapes are different. McIDAS-X supports the sinusoidal equal-area projection.

extended format
McIDAS-X commands run with an extended format can contain a semicolon, indicating the start of a sequence of commands, or one or more pound signs, indicating a required string substitution.

F

file coordinates
The coordinates of a data point in an area file referenced sequentially by lines and elements. The top line and leftmost element have the file coordinates (0,0).

file redirection
The process that lets users identify the location of individual files on a workstation.

format
File format. McIDAS-X file formats include image, grid, point and text; non-McIDAS-X formats include HDF and NetCDF.

frame
Contains a representation of an image sector displayed on the McIDAS-X Image Window. Users can define the number and size of frames; the default is four frames that are 480 lines by 640 elements.

frame coordinates
The native coordinates of a frame referenced sequentially by lines and elements. The frame's upper-left corner has coordinates (1,1). The number of lines and elements on the frame is determined by the frame size.

frame object
A memory-based collection of information that completely describes the contents and appearance of a frame to the mcimage process, which realizes it into a visible picture. Frame objects are stored in McIDAS-X shared memory.

FTP
File Transfer Protocol. A method of transferring files between workstations on a network.

full image
The entire image transmitted by a sensor source.

full resolution
One image data point represents one satellite sensor data point. Also see image resolution and satellite resolution.

function
The term used in this manual to describe C procedures and functions, and Fortran functions and subroutines.

G

geocentric latitude
The angle between the equatorial plane and a ray through the point from the Earth's center.

geodetic latitude
The angle between a line perpendicular to the surface of the geoid through a point and the Earth's equatorial plane. Due to the Earth's oblateness, geodetic latitudes (the most common form of earth location) are slightly greater than geocentric latitudes except at the equator and poles where they are identical.

geoid
The spheroid (surface formed by rotating an ellipse about the polar or Z-axis of the terrestrial coordinate system) that most closely approximates the Earth's surface.

geostationary satellite
A satellite that remains above a fixed location on the Earth's surface, usually about 36,000 km above the equator. It is limited in view, approximately 60° either side of the equator. GOES-8 and -9 view North America; Meteosat views Europe and Africa; GMS views the western Pacific.

global keyword
A keyword that can be used with any McIDAS-X command.

global string
A string name whose first character is a question mark; useful for defining strings that you don't want accidentally deleted. Global strings remain in the string table even if the current string table is replaced with another.

GMS
Geostationary Meteorological Satellite.

GOES
Geosynchronous Operational Environmental Satellite.

graphics
Text, symbols and line segments drawn in color on the McIDAS-X Image Window.

gray scale
The range of black-to-white gray shades available for displaying image data on the McIDAS-X Image Window. The range is 0 (black) to 255 (white).

gray shading
The most common method of displaying image data.

grid
A lattice of regularly spaced data points superimposed on a projection of the Earth. Grids are generated from numerical models or observational data.

grid header
The part of the grid object that contains the ancillary information about the grid, such as the parameters and physical quantities of the data in the grid, the level in the atmosphere or ocean the data represents, the grid navigation information and the time.

grid object
The actual gridded data along with the ancillary information contained in the grid header.

group name
In ADDE, the name used by the client to identify the server machine to get the data from. The server uses it to identify the data that the client is requesting.

H

half-word
A 16-bit memory segment; a 32-bit memory segment is called a word; an 8-bit memory segment is called a byte.

handle
A computer term used to describe a variable in a program that points to a specific structure. Handles are often used with input and output events.

help
A block of comments describing an applications purpose, its positional parameters and keywords, and other notable remarks.

I

image
Information that is usually represented as shades of gray in a two-dimensional matrix, such as satellite images, radar images or images derived from grids.

image coordinates
The native coordinates of remotely sensed data expressed as lines and elements. Each image is a series of lines and elements arranged from top to bottom, forming a grid for displaying data points on a McIDAS-X frame. Lines run horizontally across the frame; elements run vertically up and down the frame. The top line and leftmost element have the image coordinates (1,1). This coordinate system is independent of McIDAS-X and forms the basis for other McIDAS-X coordinate systems.

image object
The actual image along with its ancillary data.

image object block
A collection of image objects; each block contains image data or ancillary information.

image resolution
The number of satellite scan lines represented in each data point of an image line. Resolution can be increased or decreased; see blow up and blow down. Also see full resolution and satellite resolution.

image sector
A rectangular subset of a full image with the same coordinate system.

Image Window
The window used for displaying frames containing McIDAS-X-generated images and graphics.

include files
Files that hold definitions of constants specific to the McIDAS-X software; for example, **`mcidas.h`** and **`fileparm.inc`**.

ingestor
A process that listens to data received by a communications port and reformats the information for further processing.

interface documentation block
The template to use when writing a new McIDAS-X library function in Fortran or C.

I/O
Input/Output.

J

Julian day
Calendar date based on a 365-day year, usually in the form ccyyddd; for example, 1996056 is February 25, 1996.

K

keywords
Alphanumeric values that provide input to a McIDAS-X command; useful for clarifying commands with many complicated options. Keywords are always followed by an equals sign or a comma and the assigned value. They are optional for most commands and can be entered in any order as long as they follow command positional parameters and precede quoted text in the command line.

L

line
Image line; each image line contains an optional line prefix and the actual data values. Lines run horizontally across a McIDAS-X frame; they are numbered from top to bottom with the top line numbered one.

line prefix
Optional information that precedes the data on an image line; contains ancillary data about the line, such as navigation or calibration parameters.

line prefix block
The image object block containing information about an image that may vary on a line-by-line basis, such as documentation or calibration information.

little-endian
The least significant byte in a word comes first; opposite of big-endian where the most significant byte comes first.

local library
A file for keeping your local functions with their application's source code. A local library is useful for referencing functions that SSEC moves to the compatibility library. Also see compatibility library and McIDAS-X library.

lock
A unique, alphanumeric name (usually a legal file name) used to ensure that two programs can't access the same resource simultaneously.

looping
Continuous, automatic stepping through a sequence of image and/or graphics frames, much like a movie loop.

M

macro
A McIDAS-X command that runs a series of McIDAS-X commands (embedded in Fortran code) in a predefined sequence.

makefile
A description file that defines the relationships or dependencies between applications and functions; it simplifies the development process by automatically performing tasks necessary to rebuild an application when you modify code.

map files
Outlines of political or geographic boundaries that can be superimposed on the McIDAS-X Image Window using the MAP command.

McIDAS
Man computer Interactive Data Access System; a collection of tools for acquiring, analyzing and displaying meteorological data, created by the Space Science and Engineering Center of the University of Wisconsin-Madison. McIDAS-X runs on Unix workstations.

McIDAS-X library
A file called **`libmcidas.a`** that contains all the object code for the functions and subroutines that make up the McIDAS-X Application Program Interface (API). Also see local library and compatibility library.

McPATH
The environment variable in McIDAS-X that defines directories for commands to search when looking for data and help files.

memory overflow
Writing beyond the memory allocated for a variable.

Meteosat
European geosynchronous meteorological satellite.

mouse
See pointing device.

N

navigation
The process of transforming image coordinates (lines and elements) to earth coordinates (latitude and longitude) and vice versa.

navigation block
A McIDAS-X data structure containing the projection type and set of parameters for computing transformations between earth and image coordinates. Sometimes called a navigation codicil.

navigation module
A group of subroutines that are specific to a type of image data; this module is used to perform navigation.

navigation transform
A set of equations for converting a dataset's image or grid coordinates to and from earth coordinates.

network-byte-order
See big-endian.

null-terminated
Describes the practice of placing a zero (ASCII NULL character) at the end of a character string; this is the standard representation in the C language.

P

parsing
Breaking down an observation into its most elementary parts.

pipe
Shared space that accepts the output of one program for input into another.

pixel
A point on a McIDAS-X frame assigned a unique pair of line and element coordinates.

POES
Polar Orbiting Environmental Satellite.

point object
The actual point data values along with their ancillary information.

point data
Atmospheric or oceanographic data occurring at irregularly spaced locations on the Earth or vertically within the atmosphere or ocean. Most data gathered by direct measurements, such as weather balloons and synoptic reports, is stored as point data.

pointing device
A three-button mouse; the leftmost button is used by the window manager and the middle and right buttons are used by the McIDAS-X mouse interface.

polar orbiting satellite
A satellite that provides complete coverage of the Earth's surface twice per day. It normally orbits 800 to 900 km above the Earth and has a field of view that is about 2400 km, centered on the orbit path.

position
In ADDE, the absolute or time-relative position of a file in a dataset; position numbers greater than zero represent an absolute position in the dataset; numbers less than or equal to zero represent a relative position, 0 is most recent and -1 is next most recent; for example, if a dataset has four images with times 13, 14, 15 and 12, they have the positions -2, -1, 0 and -3.

positional parameters
Alphanumeric values that provide input to a McIDAS-X command; they must be entered in the exact order specified. Useful for minimizing the number of keystrokes a user types.

physical quantity
Radiance, temperature, albedo, etc.; sometimes, in error, called unit.

process chain
A series of processes run synchronously.

projection
A set of equations relating earth locations (three variables) to a location in Cartesian coordinates on the projection plane. Also see conformal projection, equal-area projection and earth coordinates.

projection parameters
One or more constants contained in projection equations; specifying values for these constants defines an instance of the projection.

pseudo-Mercator projection
A projection in which latitude and longitude vary uniformly with line (or row) and element (or column). This projection is distinct from a true Mercator and is neither conformal nor equal-area.

Q

quoted text
The last part of the command that the user enters; each application can contain only one quote string. Quoted text is preceded by double quote marks (") and is most often used when strings entered by a user require whitespace.

R

radar data
Information related to the strength of the reflected radar signal; usually correlated with rainfall intensities. Radars use active sensors that emit short-wave radiation and sample the signals reflected back to the radar antenna. Modern radars also sense the radial component of droplet velocity.

real-time data
Data that is available to users as soon as it is received by the system.

redirect tables
Part of the shared memory block that resides in the McIDAS-X environment; file redirection information from the McIDAS-X REDIRECT command is stored in them. Also see file redirection.

resident programs
Programs that are basic to McIDAS-X and its environment. For example, in McIDAS-X, the resident program mctext controls command line input from the keyboard and displays text output on the Text and Command Window.

resolution
See full, image, or satellite resolution.

S

satellite resolution
The size of the smallest feature that the satellite's sensors can detect; this is determined by the geographic width of each scanned slice of the Earth's surface observed by the satellite. Also see full resolution and image resolution.

scheduler
The part of the McIDAS-X system that initiates and ends user-defined command sequences.

selection clause
A text string used by an application to restrict information sent from the server to the client.

sensor source
A satellite device that collects a specific wavelength of radiation; for example, visible, infrared, microwave, solar protons or X-ray.

sensor source number
The number assigned to an image data source; it is stored in word 3 of the area directory; for example, 70 is the GOES-8 imager.

sensor type
Sensor name consisting of up to four characters, stored in word 52 of the area directory; for example, GVAR is the sensor on GOES-8.

server
The machine in a distributed system that stores data and supplies it to the client upon request. Each McIDAS-X workstation session acts as both a client and a local server. It can also be configured as a remote server, supplying data to all clients.

server mapping table
The table containing the list of dataset names on the server. Users assign these names with the McIDAS-X DSSERVE command.

shared memory
A component of the McIDAS-X environment that consists of User Common, redirect tables and frame objects. Resident programs use it to communicate with applications.

shell script
A program containing a set of executable commands; useful for running a series of McIDAS-X commands outside of McIDAS-X.

sleep
An application tells the operating system it doesn't want to be considered ready to be dispatched for a period of time.

slot number
The number 1, 2 or 3; allows loading of up to three navigation and calibration modules.

spectral band
The wavelength in which a scanning instrument measures data; for example, band 4 for the GOES-8 imager is 10.7 microns (infrared).

static data
Database information that changes little over time. Examples include map files, station tables, or font files.

static linking
Subprograms included at compile/link time.

station tables
A cross reference list of reporting stations.

stdin
Standard input.

stdout
Standard output.

stretching
See contrast stretching and data stretching.

string
A named character string defined by a user with the McIDAS-X TE command. A character string can be accessed by programs using parameter retrieving functions. It has two uses: it provides a shorthand method of entering commands and it allows programs to access keyword values predefined by the user that are not actually entered when the program begins. String names may contain no more than 12 alphanumeric characters; strings may not exceed 160 characters.

string table
A table of named character strings; useful for passing information between commands run at different times. An individual string table may contain no more than 256 strings.

synchronous
McIDAS-X commands that run synchronously will run to completion before control is returned to the original calling program. Also see asynchronous.

T

TCP/IP
Transmission Control Protocol/Internet Protocol. A set of communications protocols used to network dissimilar systems. The TCP protocol controls the transfer of data. The IP protocol provides the routing mechanism.

text output
Usually refers to the three types of messages that applications use to communicate status information to the user: text, error and debug messages.

Text and Command Window
The window used for entering McIDAS-X commands, displaying command output and showing workstation status information. When a session is started, 10 different text frames can be displayed in this window.

toggle
Turning on and off a McIDAS-X function, such as graphics or image frames; similar to turning a light switch on and off.

token
The smallest entity to which an observation may be parsed.

transaction
Any ADDE exchange; it implies a transfer between an ADDE client and server.

transaction logging
The record keeping done by ADDE servers for each transaction.

type
Data type: image, grid, point or text.

U

unit
See physical quantity.

Unix
Multitasking operating system originally developed by AT&T; McIDAS-X requires this system.

User Common (UC)
A component of the McIDAS-X shared memory that is used in applications to alter the display and make the applications interact with each other in predictable ways.

UTC
Coordinated Universal Time; same as GMT (Greenwich Mean Time).

V

validity code
Region of the line prefix that determines if data exists for an image line.

viewport
The region of a frame to be displayed; graphics outside this region will not appear even if drawn. Viewports are used in McIDAS-X programming to generate graphical output in panels. Also called a clipping region.

W

weather text
Data transmitted in alphanumeric form; it can be user-generated or computer-generated and contains forecasts, observations, weather advisories or other public information.

whitespace
A subset of the ASCII character set, including space, end-of-line, vertical tab, horizontal tab and form-feed characters.

word
A 32-bit memory segment; a 16-bit memory segment is called a half-word; an 8-bit memory segment is called a byte.

world coordinates
The coordinate system as viewed by a graphics program; world coordinates may be defined to be convenient for the application. Their purpose is to generate attractive, properly positioned output regardless of the size of the frame.
