McIDAS

Programmer's Manual

For McIDAS-X and McIDAS-OS2

Issued August 1997
Revised November 1998

Space Science and Engineering Center
University of Wisconsin-Madison
1225 West Dayton Street
Madison, WI 53706
Telephone (608) 262-2455
TWX (608) 263-6738
Permission is granted to make and distribute verbatim copies of this document, provided the copyright notice and this permission are preserved on all copies.

Permission is further granted to modify, amend or otherwise alter this document, and to distribute, including electronically, the modified, amended or otherwise altered document provided the copyright notice and this permission are preserved on all copies and derivative works thereof. In addition, the following notice must be added to this copyright page by each individual or organization that modifies, amends or otherwise alters this document: "This is NOT a verbatim version of the original SSEC document. Portions have been modified, amended or otherwise altered by [name and address of modifying individual or organization]."

SSEC makes no warranty of any kind with regard to the software, hardware or accompanying documentation, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. SSEC does not indemnify any infringement of copyright, patent, or trademark through use or modification of this software.

There is no expressed or implied warranty made to anyone as to the suitability of this software for any purpose. All risk of use is assumed by the user. Users agree not to hold SSEC, the University of Wisconsin-Madison, or any of its employees or assigns liable for any consequences resulting from the use of the McIDAS software.

Mention of any commercial company or product in this document does not constitute an endorsement by SSEC. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this document, and SSEC was aware of the trademark claim, the designations are printed in caps or initial caps.

The information in this document is subject to change without notice. Considerable effort has been expended to make this document accurate and complete, but SSEC cannot assume responsibility for inaccuracies, omissions, manufacturers’ claims or their representations.
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.
# Table of Contents

**Introduction** ........................................ 1-1
  - Overview of McIDAS ................................ 1-3
  - McIDAS programming philosophy .................. 1-7
  - Using this manual .................................. 1-9
  - Getting help ....................................... 1-13
  - Companion documentation ........................... 1-14
  - Using the online man pages ........................ 1-15

**Learning the Basics** ................................. 2-1
  - McIDAS user applications ......................... 2-3
  - McIDAS system overview ............................ 2-21
  - McIDAS developer overview ........................ 2-29

**Getting Started in McIDAS-X** ....................... 3-1
  - Setting up a user account ........................ 3-3
  - Compiling and linking your code ................. 3-9
  - Making program helps ................................ 3-15

**Getting Started in McIDAS-OS2** .................... 4-1
  - Setting up your user environment ............... 4-3
  - Compiling and linking your code ................ 4-7
  - Building dynamic link libraries ............... 4-13
  - Making program helps .............................. 4-14
### McIDAS Utilities

- Overview ........................................ 5-3
- User interface utilities .................... 5-5
- Display utilities ............................. 5-21
- System utilities ............................... 5-56
- Conversion utilities ......................... 5-75
- Scientific utilities ......................... 5-107

### Accessing Data

- McIDAS disk files ............................. 6-3
- Image data ..................................... 6-12
- Grid data ...................................... 6-33
- Point data ..................................... 6-48
- Text data ...................................... 6-61
- McIDAS navigation ......................... 6-67
- McIDAS calibration ........................... 6-96

### Format of the Data Files

- AREAnnnn ..................................... 7-3
- DC* ............................................ 7-43
- *.ET .......................................... 7-46
- FRAMED ....................................... 7-48
- FRAMENH.001 .................................. 7-49
- FRAMEx.y ..................................... 7-51
- GMSCAL ...................................... 7-53
- GRIDnnnn ..................................... 7-54
- *.GRX ........................................ 7-59
- HIRSCRPF .................................... 7-60
- HIRSTAUL .................................... 7-62
- IDMSL ......................................... 7-64
- MDXXnnnn ..................................... 7-66
- MSUSCRPF .................................... 7-69
- OUTL* ........................................ 7-71
- SATANNOT ..................................... 7-73
- SKEDFILE ..................................... 7-74
- UC ............................................. 7-76
- VASTBLS ....................................... 7-84
- VIRTnnnn ..................................... 7-86
# Table of Contents

**Writing ADDE Servers** ............................................. 8-1  
  Overview ......................................................... 8-3  
  Getting started ................................................. 8-4  
  Building your ADDE server ..................................... 8-10  
  Debugging servers .............................................. 8-21  
  Request syntax and data transmission formats .............. 8-23

**Guidelines for Writing Helps** .................................. A-1  
  General guidelines for all helps ............................... A-3  
  Specific guidelines ............................................. A-4  
  Sample helps .................................................... A-6

**Satellite Information** ............................................. B-1  
  Sensor source numbers ......................................... B-3  
  Bands for satellite imagery ................................... B-4

**Valid Parameter Names** .......................................... C-1

**POES AVHRR Calibration Information** ......................... D-1  
  AVHRR Calibration Background ............................... D-2  
  TIRO and AVHR Differences ..................................... D-3

**Glossary**

**Index**
Introduction

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

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

Contents

<table>
<thead>
<tr>
<th>Overview of McIDAS</th>
<th>1-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>McIDAS-X and McIDAS-OS2</td>
<td>1-4</td>
</tr>
<tr>
<td>McIDAS-XSD</td>
<td>1-5</td>
</tr>
<tr>
<td>SDI</td>
<td>1-6</td>
</tr>
<tr>
<td>McIDAS-XCD</td>
<td>1-6</td>
</tr>
<tr>
<td>McIDAS programming philosophy</td>
<td>1-7</td>
</tr>
<tr>
<td>Your responsibilities</td>
<td>1-7</td>
</tr>
<tr>
<td>Planning for McIDAS upgrades</td>
<td>1-8</td>
</tr>
<tr>
<td>Using this manual</td>
<td>1-9</td>
</tr>
<tr>
<td>How this manual is organized</td>
<td>1-10</td>
</tr>
<tr>
<td>Conventions used in this manual</td>
<td>1-11</td>
</tr>
<tr>
<td>Getting help</td>
<td>1-13</td>
</tr>
<tr>
<td>Companion documentation</td>
<td>1-14</td>
</tr>
<tr>
<td>Using the online man pages</td>
<td>1-15</td>
</tr>
</tbody>
</table>
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 currently moving toward a distributed system consisting of the following:

- Unix workstations that receive and process satellite and conventional data, and also run McIDAS-X with the ADDE (Abstract Data Distribution Environment) software
- OS/2 workstations that run McIDAS-OS2 with the ADDE software

Most McIDAS users are moving toward a distributed data environment using McIDAS-X and its subsystems and McIDAS-OS2. The McIDAS distributed system is a computing system 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 -OS2 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.

McIDAS applications, which are McIDAS programs at the command line, are packaged in three major divisions:

- user tools—McIDAS-X and McIDAS-OS2
- satellite data ingest—McIDAS-XSD
- conventional data collection—McIDAS-XCD

These packages are described on the next few pages.
McIDAS-X and McIDAS-OS2

McIDAS-X runs on Unix-based workstations and provides the basic functionality and interfaces of a distributed data system. McIDAS-X is supported on these four workstation platforms under the Unix operating system:

♦ HP/Apollo 9000 series 700
♦ IBM RS/6000
♦ Sun SPARCstation
♦ R4000 CPU-based SGI workstations

McIDAS-OS2 runs on OS/2 workstations and is supported on these three display types under the IBM OS/2 operating system:

♦ Presentation Manager (PM)
♦ VGA
♦ WIDE WORD Workstation (WWW)

McIDAS-X and -OS2 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 and -OS2 packages. These packages provide 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-XSD

The McIDAS-X Satellite Data (XSD) software package on Unix-based workstations receives and processes data from the following satellites:

♦ GOES
♦ Meteosat
♦ POES
♦ GMS
♦ DMSP

A flexible scheduling interface defines areas of interest so that data can be received and processed unattended. The ingest software averages and samples the data, and performs resolution reduction in real time. Because McIDAS-XSD produces images in McIDAS format, any McIDAS function can be applied to them, for example: enhancements, looping, overlaying other data, brightness stretching, and remapping to other projections. McIDAS-XSD uses the McIDAS-X system, but doesn't need an active McIDAS session to run.
SDI

SDI is a hardware and software package that receives and processes satellite data on a PC workstation running a Unix operating system. Currently, SDI receives and processes data from POES and GVAR satellites.

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

The SDI architecture will methodically replace McIDAS-XSD ingestors as new systems are developed. Contact your SSEC program manager or the MUG Help Desk for more information about the SDI development schedule.

McIDAS-XCD

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

The data arrives via satellite broadcast by either an outside vendor or dedicated phone line directly from the circuit source. This transmitted data is converted to McIDAS formatted files. McIDAS users can employ a wide array of McIDAS applications to interpret, change, search, and display this data.
McIDAS programming philosophy

The McIDAS 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 site’s efforts, are made available to the entire user community.

Your responsibilities

McIDAS 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 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 upgrades

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

♦ McIDAS upgrades contain all the code needed for an initial installation of McIDAS, 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 library functions beginning with m0 (Fortran) or M0 (C). These functions are considered private to the McIDAS 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 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 for one year. Functions in this library are no longer referenced by McIDAS 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 Programmer’s Manual is designed to be an instructional guide for new McIDAS programmers and a reference for experienced McIDAS programmers. It assumes that you know the Fortran and C programming languages, and have a basic knowledge of McIDAS and your operating system.

This manual provides the information that you need to:

- write McIDAS applications on Unix- and OS/2-based systems
- understand the McIDAS 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 describes the manual’s organization and its symbol, text and screen conventions.
How this manual is organized

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

<table>
<thead>
<tr>
<th>If you’re interested in</th>
<th>Turn to</th>
</tr>
</thead>
<tbody>
<tr>
<td>A brief history of McIDAS</td>
<td>Preface</td>
</tr>
<tr>
<td>Knowing what’s in this manual and who it’s written for, how to get help, and other McIDAS documentation that you can reference</td>
<td>Chapter 1, Introduction</td>
</tr>
<tr>
<td>The basics about developing applications programs in McIDAS, including the types of data available in McIDAS and the formats and conventions to use when writing online helps</td>
<td>Chapter 2, Learning the Basics</td>
</tr>
<tr>
<td>Setting up the McIDAS-X environment, and compiling, testing and debugging your McIDAS-X code</td>
<td>Chapter 3, Getting Started in McIDAS-X</td>
</tr>
<tr>
<td>Setting up the McIDAS-OS2 environment, and compiling, testing and debugging your McIDAS-OS2 code</td>
<td>Chapter 4, Getting Started in McIDAS-OS2</td>
</tr>
<tr>
<td>A description of the McIDAS library functions that you’ll use to write your applications programs</td>
<td>Chapter 5, McIDAS Utilities</td>
</tr>
<tr>
<td>Learning how to use McIDAS library functions to access data files</td>
<td>Chapter 6, Accessing Data</td>
</tr>
<tr>
<td>The file formats for the data files developed for applications running under McIDAS-X and -OS2</td>
<td>Chapter 7, Format of the Data Files</td>
</tr>
<tr>
<td>Writing and maintaining locally developed servers for ADDE</td>
<td>Chapter 8, Writing ADDE Servers</td>
</tr>
<tr>
<td>SSEC’s guidelines for writing online helps, and information about satellites and data parameters</td>
<td>Appendices</td>
</tr>
<tr>
<td>Definitions of some terms used in this manual</td>
<td>Glossary</td>
</tr>
<tr>
<td>The detailed descriptions for each API function that you need to build applications</td>
<td>Online man pages</td>
</tr>
</tbody>
</table>
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.

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, Mecmd and mcget.

♦ 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 systems. 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.

<table>
<thead>
<tr>
<th>If you want</th>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>clarification of any information contained in this manual</td>
<td>the Help Desk at (608) 262-2455, or send email to <a href="mailto:mug@ssec.wisc.edu">mug@ssec.wisc.edu</a></td>
</tr>
<tr>
<td>to report an error in the manual</td>
<td></td>
</tr>
<tr>
<td>to make a constructive suggestion for improving the manual</td>
<td></td>
</tr>
<tr>
<td>information about software maintenance and upgrades</td>
<td>the Help Desk at (608) 262-2455 or the MUG Web page at http:// <a href="http://www.ssec.wisc.edu/mug.html">www.ssec.wisc.edu/mug.html</a></td>
</tr>
<tr>
<td>help writing specific software or debugging your software</td>
<td>your SSEC Program Manager</td>
</tr>
</tbody>
</table>
The following documentation provides additional information about McIDAS.

**Online man pages**

The online man pages provided with the McIDAS 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 man pages.

**McIDAS-OS2 User's Guide**

This manual provides users with the installation instructions, general information and commands needed to run McIDAS-OS2. Each command includes a description of its usage, definitions for all positional parameters and keywords, remarks and examples. The manual also describes the F Key Menu System, ASK User Interface, Soft Tablet Interface and ADDE (Abstract Data Distribution Environment).

**McIDAS-X 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 F Key Menu System and ADDE.

**McIDAS-X 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 nine lessons: Getting Started, Loop Control System, Satellite Imagery, Graphics and the Cursor, String Tables, MD Files, Grids and Grid Files, Weather Analysis, and Enhancements.
Using the online man pages

The online man pages provided with the McIDAS software contain detailed descriptions of all the API (Application Program Interface) functions that you will need to build McIDAS applications. Typically, man pages are only available on Unix workstations. However, the McIDAS man pages are available for both Unix and some OS/2 workstations. The OS/2 workstations must use HPFS in their development drive to access the man pages.

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

♦ The McIDAS man pages are written for McIDAS software only and tell programmers how to use a particular piece of McIDAS software in their programs.

♦ The McIDAS man pages are references and are not as complete as Unix man pages. For a complete description of each utility, see Chapter 5, Utilities.

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

♦ Setting the environment

♦ Viewing man pages

♦ Directory location of the man page files
Setting the Environment

If you use McIDAS-X, 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 or OS2 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.

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

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.

<table>
<thead>
<tr>
<th>graphic</th>
<th>McTSimTimeToXAxis</th>
<th>Converts a time object to a TV element</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphic</td>
<td>McTSimValueToYAxis</td>
<td>Converts a value to a TV line number</td>
</tr>
<tr>
<td>converter</td>
<td>Mcargparse</td>
<td>Parses text into arg-fetching structure</td>
</tr>
<tr>
<td>day/time</td>
<td>Mcyydmy</td>
<td>Converts ccyyddd to day/month/year</td>
</tr>
<tr>
<td>sys_config</td>
<td>Mcdev2uc</td>
<td>Converts keyword DEV= character value</td>
</tr>
<tr>
<td>user_interface</td>
<td>Mcstrtodbl</td>
<td>Converts token to type double format</td>
</tr>
<tr>
<td>text</td>
<td>cfe</td>
<td>Converts a real character*12</td>
</tr>
</tbody>
</table>

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 `-e` 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 -e converter`. A subset of the output is shown below.

| converter   | mcargparse       | Parse text into arg-fetching structure|
| converter   | mchastostr       | 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 man page files have different directory locations in McIDAS-X and McIDAS-OS2, as described below. The McIDAS 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.

**McIDAS-X**

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.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>~/.mcidas/bin</code></td>
<td>xrefit</td>
</tr>
<tr>
<td><code>~/.mcidas/man</code></td>
<td>xref.tab, xrefawk</td>
</tr>
<tr>
<td><code>~/.mcidas/man/man3</code></td>
<td>sdest.3, Mcgettime.3, all other man page files</td>
</tr>
</tbody>
</table>

**McIDAS-OS2**

The McIDAS-OS2 man page files are placed in the directory `\mcidas\man`, which resides on the development drive. The development drive is chosen during the installation process, for example the d: drive.

In McIDAS-OS2 the man files can have .3 or .4 extensions. Man files that have mixed-case names, for example Mcgettime, are stored in the man4 subdirectory. The files used for searching man files are placed in the `\emx` subdirectory on the development drive. The table below lists the man file directories and their contents.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\mcidas\man</code></td>
<td>xref.tab</td>
</tr>
<tr>
<td><code>\mcidas\man\man3</code></td>
<td>all man page files with the .3 extension</td>
</tr>
<tr>
<td>Directory</td>
<td>Contents</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>\mcidas\man\man4</td>
<td>all man page files with the .4 extension</td>
</tr>
<tr>
<td>\emx\bin</td>
<td>man.cmd</td>
</tr>
<tr>
<td></td>
<td>xrefit.cmd</td>
</tr>
<tr>
<td>\emx\mcidas\tools</td>
<td>xrefawk</td>
</tr>
</tbody>
</table>

*Note:* It is possible that the subdirectory \emx\bin does not exist; therefore, the two .cmd files cannot be stored there. The two files will initially reside in \mcidas\tools. Follow the steps below to move them to a new location.

1. Move the files from \mcidas\tools to a new directory. It can be a directory that already exists or you can create one. SSEC suggests creating a directory named \emx\bin.

2. Verify that this subdirectory is in PATH.
Learning the Basics

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

- the types of data available in McIDAS and how users access them
- the types of commands users can enter on the McIDAS Text and Command Window
- the coordinate systems McIDAS uses for displaying images and graphics on the McIDAS Image Window
- the components of the McIDAS environment and how they’re organized
- the tools available for writing and debugging your code
- the 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
## Contents

**McIDAS user applications** ........................................ 2-3  
- Data types ........................................ 2-4  
- How users access McIDAS data ................................. 2-7  
- The McIDAS windows ........................................ 2-11

**McIDAS system overview** ................................. 2-21  
- Shared memory ........................................ 2-21  
- Resident programs ........................................ 2-25  
- Applications programs ................................... 2-27

**McIDAS developer overview** ................................. 2-29  
- Writing McIDAS applications ................................ 2-30  
- The McIDAS library ........................................ 2-35  
- C and Fortran function interfaces .......................... 2-42  
- Debugging McIDAS applications .............................. 2-48  
- Programming do's and don'ts ............................... 2-52
McIDAS user applications

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

♦ the types of McIDAS data available to users
♦ how users access McIDAS data
♦ the windows displayed when a McIDAS session is started and how to use them

Most of the discussion centers on McIDAS-X, although it applies equally to McIDAS-OS2. Differences are noted as they occur.

For a more complete description of user-level applications in McIDAS, see the McIDAS-X and McIDAS-OS2 User's Guides.
Data types

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

- images
- point observations
- grids
- weather text

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

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 software. Each type of McIDAS data is described briefly below. More information is presented in following chapters.
Images

McIDAS 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. The GOES-8 and GOES-9 satellites view North America; Meteosat views Europe and Africa; GMS views the western Pacific.

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 Text and Command Window in tabular form. If displayed on the Image Window, it is superimposed 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, 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 data easier.

The McIDAS MAP command uses map files to superimpose outlines of political or geographic boundaries on the McIDAS 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 data

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

- ADDE (Abstract Data Distribution Environment)
- file redirection
- MCPATH (McIDAS-X only)

ADDE makes McIDAS image, point, grid and weather text data available to users. During a McIDAS 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 data, see the McIDAS-X or McIDAS-OS2 User's Guide.
ADDE

ADDE is a networked system of clients and servers that makes McIDAS 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.

![Diagram of ADDE network](image)

Each McIDAS 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

McIDAS-OS2 cannot be configured to operate as a remote server, although it can operate as a local 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 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 local 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 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 session is to search for local files in the following directories.

In McIDAS-X:

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

In McIDAS-OS2:

- `c:\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 REDIRECT command. For example, a user who has a McIDAS-OS2 application that requires the file MYFILE in the directory `c:\fred\data`, would enter this REDIRECT command:

```
REDIRECT ADD MYFILE "C:\FRED\DATA"
```

All subsequent commands that access MYFILE would assume it is in `c:\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 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:$HOME/mcidas/help1:~-mcidas/data:~-mcidas/help
```

If a file exists in more than one directory in MCPATH, McIDAS 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 windows

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

- McIDAS Text and Command Window
- McIDAS Image Window

Figure 2-3 below shows a typical start-up screen. The title bar in each window lists the version of McIDAS. In McIDAS-X, it also shows the user's logon and the host name.

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

*Figure 2-3. At start-up, the McIDAS display contains the McIDAS Text and Command Window (in the foreground) and the McIDAS Image Window (in the background).*
McIDAS Text and Command Window

The McIDAS Text and Command Window is used for:

- entering McIDAS commands
- displaying command output
- showing workstation status information

When a session is started, McIDAS 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 2-4 shows a sample McIDAS Text and Command Window.

**Figure 2-4.** The McIDAS Text and Command Window displays text messages.

Users enter two kinds of McIDAS 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 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>advances the current image loop forward by one frame</td>
</tr>
<tr>
<td>B</td>
<td>goes backward in the current image loop by one frame</td>
</tr>
<tr>
<td>C</td>
<td>returns the origin of the current image frame</td>
</tr>
<tr>
<td>D</td>
<td>samples the image value at the cursor position</td>
</tr>
<tr>
<td>E</td>
<td>returns the earth coordinates at the center of the cursor</td>
</tr>
<tr>
<td>F</td>
<td>displays the status of the McIDAS session</td>
</tr>
<tr>
<td>L</td>
<td>stops and starts the currently defined loop</td>
</tr>
<tr>
<td>O</td>
<td>displays the opposite image frame</td>
</tr>
<tr>
<td>P</td>
<td>locks and unlocks the cursor</td>
</tr>
<tr>
<td>Q</td>
<td>exits the command; used with interactive commands</td>
</tr>
<tr>
<td>W</td>
<td>turns the graphics on and off</td>
</tr>
<tr>
<td>Z</td>
<td>zooms the Image Window in and out</td>
</tr>
</tbody>
</table>

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

**Multiple-letter commands**

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

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

A sample McIDAS command line is shown below.

```
MYPROG 72747 72203 FORM=T TD Z "DATA FROM INL TO PBI"
```

```
Command Name  Command Positional Parameters  Keyword  Keyword Positional Parameters  Quoted Text
```
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 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 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 has five global keywords, which are common to all McIDAS commands:

- **DEV=** specifies the destination device of the text output generated by a command.
- **FONT=** specifies the font for drawing text on the McIDAS Image Window.
- **TCOL=** specifies the color for command text output in the text window.
- **TWIN=** specifies the destination text window to route the command output to.
- **VIRT=** specifies a virtual graphics number to write graphical output to.

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-X or McIDAS-OS2 User’s Guide.

---

*For more information about the McIDAS global keywords, see the McIDAS-X or McIDAS-OS2 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 command including the parameters, keywords and remarks. To access the online help, type `HELP` in the McIDAS 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; the full text is displayed on text frame 5.
McIDAS Image Window

The McIDAS Image Window displays frames containing McIDAS generated images and graphics, as shown in Figure 2-5 below.

Figure 2-5. The McIDAS Image Window displays images and graphics.

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

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

These factors are described below.
Image and graphics levels

The McIDAS 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. The default configuration is 128 gray shades and 16 graphics levels with a maximum of 254 total levels for the image and graphics levels. In McIDAS-OS2, these values vary among workstations.

Color enhancements

When McIDAS 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 EU command modifies the color combinations for image data; the GU command modifies the colors of graphics levels.

Frame size

Users can also define the number and size of frames in the McIDAS Image Window. The default is to start the session with four frames that are 480 lines (vertical) by 640 elements (horizontal).

Loop control

McIDAS can display an automatically repeating sequence of frames much like a movie loop. The McIDAS 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 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 Image Window. McIDAS graphics also use a world coordinate system, which is described in the section titled "Advanced McIDAS graphics techniques" in Chapter 5, McIDAS Utilities.
**Image coordinates**

McIDAS receives satellite images from geostationary and polar orbiting satellites. The image coordinates for this data are defined by the sensor source providing the image and form the basis for the other McIDAS coordinate systems.

A *full image* is the entire image transmitted by a satellite. It is a sequence of lines and elements usually arranged from top to bottom, forming a grid for displaying data points on a McIDAS 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 satellites 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, 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, 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 frames are arranged by lines and elements. A frame contains a representation of an image sector displayed on the McIDAS Image Window. The image sector shown in Figure 2-7 on the next page 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 Image Window. The Window Manager then displays the McIDAS Image Window on the workstation monitor or X Terminal.

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 navigation sections in Chapter 6, Accessing Data.
Figure 2-7. Earth and frame coordinates are shown for the image sector.
McIDAS system overview

Now that you’ve seen McIDAS from a user’s perspective, you need to understand how McIDAS operates at the system level. When McIDAS 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 environment and how they fit together.

The McIDAS environment consists of these three components:

- shared memory
- resident programs
- applications programs

Each is described below. The discussion here is written for McIDAS-X, with applicable information for McIDAS-OS2 provided as needed.

Shared memory

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

- User Common
- image frame objects
- redirection tables

In McIDAS-X, the shared memory key associated with the memory is stored in the environment variable MCENV_POSUC.

In McIDAS-OS2, shared memory is created on system start-up. The McIDAS program c:\mcidas\code\malloc.exe is started from CONFIG.SYS.
User Common

User Common (UC) contains information about the McIDAS 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 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 session is started. It is created by `mmap` when a McIDAS 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 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 programmer, you need to understand the concepts of visibility and inheritance when accessing positive and negative UC.

- Visibility describes the portions of the McIDAS 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 command is described below.
**Does positive UC exist?**

- If positive UC does not exist, the McIDAS 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 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 command runs synchronously**, it inherits negative UC from its parent process. This happens, for example, when a command is run from a McIDAS BATCH 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 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 Text and Command Window. The version of negative UC created is visible only while the command that created it is active.

Many user applications take values from UC when they run. For example, most commands that write output to the Image Window use the current frame number as the default output frame. The current frame number is found in both positive and negative UC. When writing output to the Image Window, use negative UC to get the current frame number. If you extract the frame number with positive UC and the frames are looping, the default destination frame number for that command will be unpredictable.

Conversely, if you write an interactive application that uses the cursor position for input, use positive UC. If you use negative UC for the cursor position, the value returned will be the cursor position when negative UC was created (when the command started), not the current position.

For a complete list of the contents of User Common, see *Chapter 7, 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 Image Window. McIDAS 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 is started, the contents of the current redirection table are loaded into shared memory. This is where the McIDAS disk file system retrieves its redirection information.
Resident programs

The second component of the McIDAS environment is resident programs. When a user types the `mcidas` command, the user-specified configuration information is determined. In Unix, this configuration information is in the text file `$HOME/.mcidasrc`. In McIDAS-OS2, it is in the file `\mcidas\data\startup.sys`, which is created by `c:\mcidas\tools\setup`.

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

<table>
<thead>
<tr>
<th>McIDAS-X</th>
<th>McIDAS-OS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcenv</td>
<td></td>
</tr>
<tr>
<td>mctext</td>
<td>KBDCTL and SCRCTL</td>
</tr>
<tr>
<td>mctime</td>
<td>PM, WWW, or VGA</td>
</tr>
</tbody>
</table>

**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.
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 alphanumerical text components needed for the session
- monitors keyboard entry and the placement of the alphanumerical output generated by the commands

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

The commands **serc1.exe** and **kbdctl.exe** perform similar text screen and keyboard tasks for McIDAS-OS2.

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 Image Window
- controls the display of the McIDAS cursor

The commands **pm.exe**, **www.exe** and **vga.exe** perform similar functions for the image display platforms in McIDAS-OS2.

**Applications programs**

The third component of the McIDAS environment is applications programs. A McIDAS applications program is any program that runs from the McIDAS command line. McIDAS provides Application Programming Interface (API) functions that interact with all the McIDAS 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 applications rely on the API functions to read data files for them.*
Figure 2-10 combines the three components of the McIDAS 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 from a user’s perspective and provided an overview of the McIDAS environment. This final section describes how to build applications that will interact with the McIDAS environment. In this section, you will learn how to:

- get started writing McIDAS applications in Fortran and C
- use McIDAS include files
- set status codes in McIDAS
- write online command helps
- name your McIDAS 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 uses to name source files.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.c</td>
<td>C</td>
<td>functions and McIDAS commands</td>
</tr>
<tr>
<td>.for</td>
<td>Fortran</td>
<td>functions and subroutines</td>
</tr>
<tr>
<td>.fp</td>
<td>Fortran</td>
<td>ADDE servers and non-McIDAS applications</td>
</tr>
<tr>
<td>.cp</td>
<td>C</td>
<td>ADDE servers and non-McIDAS applications</td>
</tr>
<tr>
<td>.h</td>
<td>C</td>
<td>include files</td>
</tr>
<tr>
<td>.inc</td>
<td>Fortran</td>
<td>include files</td>
</tr>
<tr>
<td>.mac</td>
<td>Fortran</td>
<td>McIDAS macros</td>
</tr>
<tr>
<td>.dlm</td>
<td>Fortran</td>
<td>dynamic link modules</td>
</tr>
<tr>
<td>.pgm</td>
<td>Fortran</td>
<td>McIDAS commands</td>
</tr>
</tbody>
</table>
Writing McIDAS applications

You can write McIDAS applications in either C or Fortran. This section describes the concepts you need to know to write basic McIDAS 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, you can’t write it as a Fortran MAIN program. Instead, you must place it in the McIDAS 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 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 Text and Command Window.

```fortran
subroutine main0
call sdest('Hello World',0)
call mncodeset (0)
return
end
```
C programs

When writing a C program for McIDAS, the main is provided directly in the C code. The first thing you will do when writing a McIDAS application in C is to initialize the McIDAS 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 applications in C, your source code should also contain a reference to the McIDAS include file mcidas.h.

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

```c
#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 uses include files to hold definitions of constants specific to its software. Four of these include files are discussed below: mcidas.h, fileparm.inc, areaparm.inc and gridparm.inc.

mcidas.h

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

<table>
<thead>
<tr>
<th>Constant value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMISSING</td>
<td>1-byte McIDAS standard missing value code</td>
</tr>
<tr>
<td>MCMISSING4</td>
<td>4-byte McIDAS standard missing value code</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>Defined type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fdouble</td>
<td>equivalent to a double precision declaration type in Fortran</td>
</tr>
<tr>
<td>Fint</td>
<td>equivalent to an integer declaration type in Fortran</td>
</tr>
<tr>
<td>Fint2</td>
<td>equivalent to an integer*2 declaration type in Fortran</td>
</tr>
<tr>
<td>Fint4</td>
<td>equivalent to an integer*4 declaration type in Fortran</td>
</tr>
<tr>
<td>Freal</td>
<td>equivalent to a real declaration type in Fortran</td>
</tr>
<tr>
<td>Freal4</td>
<td>equivalent to a real*4 declaration type in Fortran</td>
</tr>
<tr>
<td>Freal8</td>
<td>equivalent to a real*8 declaration type in Fortran</td>
</tr>
<tr>
<td>FsLen</td>
<td>used for passing string lengths into Fortran routines</td>
</tr>
<tr>
<td>Mcount2</td>
<td>2-byte signed integer</td>
</tr>
<tr>
<td>Mcount4</td>
<td>4-byte signed integer</td>
</tr>
<tr>
<td>Mcount2</td>
<td>2-byte unsigned integer</td>
</tr>
<tr>
<td>Mcount4</td>
<td>4-byte unsigned integer</td>
</tr>
</tbody>
</table>

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.

**fileparm.inc**

The include file **fileparm.inc** contains the maximum file name length allowed, including the path name. The parameter containing this length is MAXPATHLENGTH. This value is usually 512 or 1024 characters.

**areaparm.inc**

The include file **areaparm.inc** contains the maximum number of data elements that one scan line can hold (MAXDEFELEMENTS). It also contains the maximum number of images that can be simultaneously accessed (MAXOPENAREAS).

**gridparm.inc**

The include file **gridparm.inc** contains two constants that you should use when manipulating grid data. MAXGRIDPT contains the maximum number of data points that can be stored in an individual grid. MAXGRIDNUM contains the maximum grid file number for the system.
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 commands are as follows:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>command completed successfully</td>
</tr>
<tr>
<td>1</td>
<td>command contained an unrecoverable error</td>
</tr>
<tr>
<td>2</td>
<td>command contained an error that may be recoverable</td>
</tr>
<tr>
<td>3 - 99</td>
<td>reserved for SSEC</td>
</tr>
<tr>
<td>100 - 127</td>
<td>site-defined return values</td>
</tr>
</tbody>
</table>

This information may be very useful to the calling environment, especially when using McIDAS 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 applications became mandatory, beginning with McIDAS version 7.0 released in May 1996. Most McIDAS applications written before that upgrade do not yet contain the Mccodeset/mccodeset features.
Online command helps

The McIDAS HELP 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 parml parm2 <keywords> "quote
C ? NAME FUNCT2 parml <keywords>
C ? Parameters:
C ? FUNCT1 | describe the purpose of this function option
C ? FUNCT2 | describe the purpose of this function option
C ? parml  | 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 ? 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 parml parm2 <keywords> "quote
* ? NAME FUNCT2 parml <keywords>
* ? Parameters:
* ? FUNCT1 | describe the purpose of this function option
* ? FUNCT2 | describe the purpose of this function option
* ? parml  | 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, and Chapter 4, Getting Started in McIDAS-OS2.
The McIDAS library

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

The name of the McIDAS library is different in McIDAS-X and -OS2:

- libmc.lib in McIDAS-OS2
- libmcidas.a in McIDAS-X

API naming conventions

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

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

The McIDAS 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 library and will have neither its functionality nor calling sequence changed. The M0 or m0 prefix is assigned to:

- functions that are never intended to be called by applications
- functions that will eventually become Mc/mc prefixed

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 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 is as follows:

- All references to the old function will be removed from all applications in the McIDAS packages.
- The old function will be moved to a compatibility library where it will remain for approximately one year.
- When that year expires, the function will be permanently removed from McIDAS.
- If a site needs to use the old function after it is removed from McIDAS, it is the site’s responsibility to maintain support for it.

**Interface documentation block**

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>name of the function</td>
</tr>
<tr>
<td>Interface</td>
<td>details of the function’s calling sequence and return type</td>
</tr>
<tr>
<td>Input</td>
<td>variables in the calling sequence that are input only</td>
</tr>
<tr>
<td>Input/Output</td>
<td>variables in the calling sequence used for both input and output</td>
</tr>
<tr>
<td>Output</td>
<td>variables in the calling sequence that are output only</td>
</tr>
<tr>
<td>Return values</td>
<td>possible function return values and what they mean</td>
</tr>
<tr>
<td>Remarks</td>
<td>information about a function that may be useful to a programmer</td>
</tr>
<tr>
<td>Categories</td>
<td>keywords identifying which subsystem this function operates on</td>
</tr>
</tbody>
</table>
Name
This field is usually a one-line description of what the function does. The example below is taken from the function `mchmstostr`.

<table>
<thead>
<tr>
<th>Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mchmstostr - Converts a time to a character string</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Interface:</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer function</td>
</tr>
<tr>
<td>mchmstostr (integer hms, integer format, character(*) string)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>hms - Time in the form hhmms.</td>
</tr>
<tr>
<td>format - Output format desired for the string.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Input and Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>string - Destination character string.</td>
</tr>
</tbody>
</table>
**Return values**

This field describes the possible values that can be returned by this function. The McIDAS 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.

<table>
<thead>
<tr>
<th>Return values:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success.</td>
</tr>
<tr>
<td>-1</td>
<td>Invalid value for hms.</td>
</tr>
<tr>
<td>-2</td>
<td>Invalid value for format.</td>
</tr>
<tr>
<td>-4</td>
<td>Destination string is not big enough.</td>
</tr>
</tbody>
</table>

**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 *mehmstosstr* is shown below.

<table>
<thead>
<tr>
<th>Remarks:</th>
<th></th>
</tr>
</thead>
</table>
| 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 are:

<table>
<thead>
<tr>
<th>calibration</th>
<th>converter</th>
<th>day/time</th>
<th>display</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>file</td>
<td>graphic</td>
<td>grid</td>
</tr>
<tr>
<td>image</td>
<td>ingest/decode</td>
<td>met/science</td>
<td>navigation</td>
</tr>
<tr>
<td>point</td>
<td>text</td>
<td>sys_config</td>
<td>system</td>
</tr>
<tr>
<td>user_interface</td>
<td>utility</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Be careful not to assign more categories than needed to a function. For example, the McIDAS library function to display a contour graphic of a grid is `mcgridcon`. The logical categories for this routine are `graphic` and `grid`. The function `mcget` 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 `mchmstost`, 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(...)

McIDAS Programmer’s Manual
Issued 8/97
Historically, McIDAS applications were written in Fortran. Recently, C was included as a supported programming language for McIDAS applications. Not all functions in the McIDAS library have both C and Fortran calling interfaces.

While McIDAS attempts to accommodate both languages, C language programs will sometimes need to interface with Fortran-coded functions from the McIDAS 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 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 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.

```fortran
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:

```c
#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 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:

\[
index = ((\text{column} - 1) \times \text{Number Of Rows}) + \text{row}
\]

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

\[
index = (\text{row} \times \text{Number Of Columns}) + \text{column}
\]
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:

```c
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.

```c
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 */
char *c_output_units; /* C string representation */
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 library function _fmalloc converts Fortran strings to C strings. The function _strofs 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 Dynamic Link Library modules**

Certain subsystems in McIDAS, such as navigation and calibration, extensively use Dynamic Link Library modules (DLLs). DLLs allow functions to be linked into an application at runtime as needed, rather than when the application is initially compiled and linked. The advantage of using DLLs in these subsystems is that applications can use new navigation or calibration functions as soon as the DLL is compiled. The application itself doesn’t have to be recompiled before using these functions.

For information on building DLLs, see Chapter 3, *Getting Started in McIDAS-X*, and Chapter 4, *Getting Started in McIDAS-OS2*.

**Writing shell scripts with McIDAS commands**

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

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

Print statements

In McIDAS, the simplest debugging aide is `ddest/Mcddprintf`. 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 `Mcddprintf`, see the section titled `Text messages` in Chapter 5, McIDAS Utilities.

**M0ASSERT**

Another debug facility provided in McIDAS 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.

```c
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 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 file extensions are not the standard file extensions for the language, *dbx* will not recognize the original source file name.

- In general, McIDAS expects the data an application will use to be in the directory `~/mcidas/data`.

- Because McIDAS 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 `~/mcidas/data` and `~/mcidas/data`.

5. Start an environment for McIDAS 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. Continue your debugging session. This will usually begin by setting some breakpoints and then typing `run`.
dbx has many commands for finding bugs. The table below lists some of the most commonly used commands.

<table>
<thead>
<tr>
<th>dbx command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>removes all breakpoints</td>
</tr>
<tr>
<td>clear (n)</td>
<td>removes breakpoints at line (n)</td>
</tr>
<tr>
<td>cont</td>
<td>continues running from the point at which the program stopped</td>
</tr>
<tr>
<td>delete (n)</td>
<td>removes traces, stops and whens that correspond to each command (n) given by status</td>
</tr>
<tr>
<td>display (exp)</td>
<td>displays (exp) every time execution stops; you can have multiple displays active simultaneously</td>
</tr>
<tr>
<td>examine</td>
<td>examines the contents of a memory location</td>
</tr>
<tr>
<td>help</td>
<td>prints helps for all dbx commands</td>
</tr>
<tr>
<td>list (n1, n2)</td>
<td>lists source lines (n1) through (n2)</td>
</tr>
<tr>
<td>list (func)</td>
<td>lists the source lines of a function</td>
</tr>
<tr>
<td>next (n)</td>
<td>runs the next (n) source lines</td>
</tr>
<tr>
<td>print (exp)</td>
<td>prints the values in the current list of expressions; to display an expression in another module, use a syntax like this: print <code>subb.f</code> <code>subb</code> <code>value</code></td>
</tr>
<tr>
<td>quit</td>
<td>exits dbx</td>
</tr>
<tr>
<td>run (args)</td>
<td>runs the command with the argument list</td>
</tr>
<tr>
<td>status</td>
<td>lists the current trace, stop and when commands</td>
</tr>
<tr>
<td>step (n)</td>
<td>runs (n) lines of source code</td>
</tr>
<tr>
<td>stop at (n)</td>
<td>stops execution at line (n)</td>
</tr>
<tr>
<td>stop if (cond)</td>
<td>stops execution if a condition is met</td>
</tr>
<tr>
<td>stop in (func)</td>
<td>stops execution if a specific function is called</td>
</tr>
<tr>
<td>trace in (func)</td>
<td>reports if a specific function is called</td>
</tr>
<tr>
<td>trace (exp) at (n)</td>
<td>prints the value of (exp) each time line (n) is reached</td>
</tr>
<tr>
<td>trace (n)</td>
<td>shows source line (n) before running it</td>
</tr>
<tr>
<td>trace (var)</td>
<td>prints the value of (var) each time it changes</td>
</tr>
<tr>
<td>undisplay</td>
<td>stops displaying expressions set by display</td>
</tr>
<tr>
<td>where</td>
<td>tells where in execution you are in the stack</td>
</tr>
<tr>
<td>whereis (exp)</td>
<td>prints a full path location to an expression</td>
</tr>
</tbody>
</table>

For a complete listing of the options available in dbx, refer to the dbx man page.
Programming do’s and don’ts

This section describes some of the common issues that McIDAS 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 5 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 next century**

Historically, the McIDAS standard for representing days was to use the Julian day with two digits representing the year of the century. For example, 1 January 1997 is 97001. Because this representation will not work starting 1 January 2000, it was changed to include the century. For example, 1 January 1997 is now represented as 1997001. The McIDAS library contains many functions for handling this new Julian day format. These functions are described in the section titled *Conversion utilities* in Chapter 5 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:

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

  Instead, do:

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

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

  ```fortran
  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.

  ```fortran
  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:

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

Instead, do:

```fortran
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:

```fortran
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:

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

Make certain your code passes arguments to foo like this:

```fortran
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:

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

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

```fortran
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:

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

Instead, do:

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

♦ Don’t embed function calls in concatenation of character strings. For example:

```fortran
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:

```fortran
REAL temp
temp = 32.0
write (string, FMT='(a20,14)')'The temperature is ',temp
call sdest(string,0)
```

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

```fortran
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:

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

Instead, do:

```fortran
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:

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

Declare them like this:

```c
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.

```c
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.
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

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up a user account</td>
<td>3-3</td>
</tr>
<tr>
<td>Creating your user account</td>
<td>3-3</td>
</tr>
<tr>
<td>Directories</td>
<td>3-5</td>
</tr>
<tr>
<td>Libraries</td>
<td>3-8</td>
</tr>
<tr>
<td>Compiling and linking your code</td>
<td>3-9</td>
</tr>
<tr>
<td>Compiling source files</td>
<td>3-9</td>
</tr>
<tr>
<td>Linking object files</td>
<td>3-10</td>
</tr>
<tr>
<td>Storing object files in a library</td>
<td>3-11</td>
</tr>
<tr>
<td>Building Dynamic Link Libraries</td>
<td>3-12</td>
</tr>
<tr>
<td>Making program helps</td>
<td>3-15</td>
</tr>
</tbody>
</table>
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 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 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-X User’s Guide for instructions on installing the mcidas account. See Chapter 2 of that manual for additional information on setting up user accounts.
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- 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-specific directories in your PATH in this order:

- $HOME/mcidas/bin, which contains your local applications
- any directory (~mlocal/mcidas/bin for example) containing other site-developed McIDAS 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 section Preparing the mcidas Account in Chapter 1 of the McIDAS-X User's Guide for more information on modifying the PATH environment variable 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**

The package directories contain the files for creating McIDAS-X. The names of the directories depend on the package name and version number. For example, the McIDAS-X version 7.2 package directories and their contents are shown in the table below.

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ mcidas/mcidas7.2/src</td>
<td>McIDAS-X version 7.2 source files, help files and binaries</td>
</tr>
<tr>
<td>~ mcidas/mcidas7.2/data</td>
<td>McIDAS-X version 7.2 data files</td>
</tr>
<tr>
<td>~ mcidas/mcidas7.2/tcl7.5pl</td>
<td>Tcl 7.5 files for GUI</td>
</tr>
<tr>
<td>~ mcidas/7.2/tk4.1pl</td>
<td>Tk 4.1 files for GUI</td>
</tr>
</tbody>
</table>

**Installation directories**

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

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ mcidas/admin</td>
<td>Administrative files</td>
</tr>
<tr>
<td>~ mcidas/bin</td>
<td>Program executables</td>
</tr>
<tr>
<td>~ mcidas/data</td>
<td>Data files</td>
</tr>
<tr>
<td>~ mcidas/help</td>
<td>Help files</td>
</tr>
<tr>
<td>~ mcidas/inc</td>
<td>Include files</td>
</tr>
<tr>
<td>~ mcidas/lib</td>
<td>Libraries</td>
</tr>
<tr>
<td>~ mcidas/man</td>
<td>Man files for subroutines and functions</td>
</tr>
<tr>
<td>~ mcidas/tcl</td>
<td>Tcl and Tk executables and libraries</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$HOME/mcidas</td>
<td>root directory for local McIDAS code</td>
</tr>
<tr>
<td>$HOME/mcidas/bin</td>
<td>local executables</td>
</tr>
<tr>
<td>$HOME/mcidas/data</td>
<td>local data files</td>
</tr>
<tr>
<td>$HOME/mcidas/src</td>
<td>source code for local functions and applications</td>
</tr>
<tr>
<td>$HOME/mcidas/help</td>
<td>helps for local applications</td>
</tr>
<tr>
<td>$HOME/mcidas/lib</td>
<td>local development library</td>
</tr>
</tbody>
</table>
Libraries

When you develop a new function for McIDAS-X, you must either add it to the McIDAS 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 library

The McIDAS library, libmcidas.a, contains all the object code for the functions, subroutines, and Dynamic Link Modules that make up the McIDAS 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 a 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 core functions, making them less susceptible to naming collisions and eliminating the need to regenerate your functions after a McIDAS 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 `mear` script to create and update libraries.

Compiling source files

The script `mccomp` provides a platform-independent compilation and linking environment for McIDAS 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>compiles only</td>
</tr>
<tr>
<td>-Dval</td>
<td>sets preprocessor values</td>
</tr>
<tr>
<td>-g</td>
<td>produces symbol table for debugging</td>
</tr>
<tr>
<td>-Idir</td>
<td>searches a list of directories for include files</td>
</tr>
</tbody>
</table>

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 command `program.k`.

```
mcomp -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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-Ldir</code></td>
<td>links the command with the a directory containing the libraries</td>
</tr>
<tr>
<td><code>-llibrary</code></td>
<td>links the command with a specified library</td>
</tr>
<tr>
<td><code>-obinary</code></td>
<td>specifies a name for the command created</td>
</tr>
</tbody>
</table>

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 -lmidas -o program.k
```

Note that the script contains the object file `main.o`. McIDAS 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 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 mcomp to compile your function and create the object file, foo.o.

   mcomp -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 mcomp script below. Assume the bar command is written in C so that it contains its own main.o.

   mcomp 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 mcomp 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. Unlike McIDAS-OS2, 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 for the navigation and calibration subsystems. For this process to work,

- the API function declarations must be uppercase, for example,
  \begin{verbatim}
  INTEGER FUNCTION NVXINI(...) 
  \end{verbatim}

- the common block names must be uppercase, for example,
  \begin{verbatim}
  COMMON /TANG/ ... 
  \end{verbatim}

- these phrases should not occur anywhere else in the module, including within comments

- the source file names must be prefixed with \texttt{nvx} for navigation modules and \texttt{kbx} for calibration modules

Generating DLL subsystem functions

When adding a new dynamic link library to McIDAS, you must generate a new subsystem initialization function. For navigation, this function is named \texttt{nvprep.for} and for calibration it is named \texttt{kbprep.for}. This is most easily done using the \texttt{nav_init} script and \texttt{cal_init} scripts provided on the MUG Web site.

To download the scripts and make them executable, follow the steps below.

1. Select \textit{MUG Software Exchange} from the MUG Web site.

2. Select \textit{Software currently on-line}. The \texttt{cal_init} and \texttt{nav_init} scripts appear at the bottom of the list.

3. Save these Unix scripts in the same directory as your new navigation and calibration modules.
4. Enter the command below at the Unix prompt to make the scripts executable.

   chmod +x nav_init cal_init

5. Generate a new `nvprep.for` and a new `kbprep.for` so it can use the new navigation and calibration modules as well as the navigation and calibration in McIDAS-X version 7.2. Run the command below at the Unix prompt to generate `nvprep.for`.

   nav_init-mcidas/mcidas7.2/src/nvx*.dlm nvx*.dlm> nvprep.for

6. Run the command below to generate `kbprep.for`.

   cal_init -mcidas/mcidas 7.2/src/kbx*.dlm kbx*.dlm > kbprep.for

7. You must then recompile `nvprep.for` and all applications that use navigation before they can access the new type. Note that `~mcidas/mcidas7.2/src` is the directory where the core McIDAS-X source can be found for version 7.2 (May 1997). Adjust this accordingly for other versions of McIDAS-X.

---

**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`, used on the Unix platforms.

- Write the module in Fortran, since `convdlm` only runs against Fortran. Using a C module in Unix will require a manual modification of the entry point names. C modules work well in OS/2.

- Write the code in uppercase. When `convdlm` was written, all the modules were in uppercase; calibration modules originated from the mainframe and were ported to OS/2. 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:

```fortran
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 .DLMs 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 .DLMs.

To run `convdlm`, use: `convdlm filename`
For example: `convdlm kbxtext.dlm`
Making program helps

The McIDAS command HELP allows users to quickly see the online structure of a command's argument list. When you write a McIDAS 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 McIDAS HELP command. The help files included in the McIDAS 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 McIDAS HELP command, see the McIDAS-X User's Guide. For more information about the HELP command format, see Chapter 2 in this manual.
This chapter provides the information you need to develop applications programs under McIDAS-OS2. You'll learn how to:

- set up your user environment, which consists of directories and libraries
- compile and link your McIDAS-OS2 code
- build dynamic link libraries
- make program helps and HELP files

Contents

<table>
<thead>
<tr>
<th>Setting up your user environment</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory</td>
<td>4-3</td>
</tr>
<tr>
<td>Libraries</td>
<td>4-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compiling and linking your code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development tools</td>
<td>4-7</td>
</tr>
<tr>
<td>Compiling source files</td>
<td>4-10</td>
</tr>
<tr>
<td>Storing object files in a library</td>
<td>4-12</td>
</tr>
</tbody>
</table>

| Building Dynamic Link Libraries        | 4-13 |
|                                        |      |

| Making program helps                   | 4-14 |
|                                        |      |
**Setting up your user environment**

To develop applications programs under McIDAS-OS2, you must establish a proper user environment. This user environment is your development arena for creating and debugging local code and datasets. Your McIDAS-OS2 user environment will contain these two components:

- local directories that you will create to hold locally developed source and executable code
- a local library that you will create to keep local functions separate from McIDAS core functions

Unlike McIDAS-X, McIDAS-OS2 doesn’t need a special user account for running McIDAS or developing applications. Installing McIDAS-OS2 and its companion Development Software on your workstation establishes the directories and settings that you need to develop applications programs.

In this section, you will learn which directories SSEC provides with the McIDAS-OS2 package and which directories you must create, and why you should make your own development library.

> Verify that the McIDAS-OS2 software and the accompanying Development Software are properly installed on your workstation. The installation instructions are in Chapter 1 of the *McIDAS-OS2 User’s Guide.*
Installing the McIDAS-OS2 and Development Software establishes the directories listed below and all developer settings on your workstation.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>\mcidas\tools</td>
<td>libraries, scripts, and editor</td>
</tr>
<tr>
<td>\mcidas\source</td>
<td>McIDAS-OS2 source files</td>
</tr>
<tr>
<td>\mcidas\code</td>
<td>core executable code</td>
</tr>
<tr>
<td>\mcidas\data</td>
<td>data</td>
</tr>
<tr>
<td>\mcidas\help</td>
<td>.HLP files</td>
</tr>
<tr>
<td>\mcidas\man</td>
<td>man pages for McIDAS library APIs</td>
</tr>
</tbody>
</table>

In addition, you should create two local directories for your user environment:

- \mcidas\working for your locally developed source code
- \mcidas\user\code for your locally developed object code

McIDAS upgrade procedures will not affect the contents of these two directories.

Do not change the \mcidas\source and \mcidas\code directories. The files in these directories are deleted during a McIDAS-OS2 upgrade.
When you develop a new function or subroutine for McIDAS-OS2, you must either add it to the McIDAS library or put it in a local library that you've created as part of your user environment. The purpose of these libraries and some of the rules governing them are described below.

The McIDAS library

The McIDAS library, \texttt{libmc.lib} in McIDAS-OS2, is a file containing all the object code for the functions and subroutines that make up the McIDAS Application Program Interface (API).

If you develop applications programs that link to \texttt{libmc.lib}, 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 a list of function changes with each McIDAS-OS2 upgrade. As a developer, you can link to this library if you are only developing commands. If you are developing functions and subroutines, you should create and use a local library for your function and subroutine object code.

Functions removed from the McIDAS library will be placed in a compatibility library for one year. The McIDAS-OS2 compatibility library is currently under development. Functions in this library will not be referenced by any McIDAS core application and will not be tested for upgrades. To continue referencing these functions, you will have to treat them as locally developed code.

Local libraries

When developing code, you will undoubtedly produce your own set of functions to support your applications, and need a local repository 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 core functions. 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 libraries.

- 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. Use the emx command emxomfar to list the functions contained in libraries.

An example of creating and accessing a local library is provided in the next section, *Compiling and linking your code.*
Compiling and linking your code

You will need to compile and link your code after installing each McIDAS-OS2 upgrade, and when developing McIDAS-OS2 commands and functions. In this section, you will learn how to:

- obtain and set up the development tools
- compile and link your code
- build dynamic link libraries
- make program helps

Development tools

McIDAS-OS2 is built using the emx development kit, and the gcc and f2c compilers. This section tells you how to obtain and set up these development tools.

Acquiring the tools and compilers

You can obtain a copy of the emx, gcc, and f2c tools and compilers from SSEC either via anonymous FTP or on diskettes. Instructions for both methods are provided on the next page.

Obtaining the ZIP file and unpacking its contents creates a directory tree under \emx. Be sure to read the information in \emx\doc. The COPYING.TXT and other.TXT files provide background information on the emx development kit and the gcc and f2c compilers. Detailed information is also available in the \emx\book directory; use the VIEW command in OS/2 to read its.INF files.
**Via anonymous FTP**

Use the steps below to access the ZIP file containing the compilers.

1. Change to the drive where you plan to install the compilers. Then change to the *root* of that drive.
   
   Type: `cd \`

2. Enter one of these FTP addresses.
   
   Type: `ftp ftp.ssec.wisc.edu`
   
   -or-
   
   `ftp 144.92.108.61`

3. Login as *anonymous*.

4. Enter the four lines below to get the ZIP file.
   
   Type: `cd pub/os2`
   
   `bin`
   
   `get mcemx32b.zip`
   
   `quit`

**On diskettes**

To request the ZIP file on diskettes, call the McIDAS Help Desk at (608) 262-2455. Use the commands below to restore the file to your hard drive.

1. Change to the drive where you plan to install the compilers. Then change to the *root* of that drive.
   
   Type: `cd \`

2. Restore the file to that drive.
   
   Type: `restore a:\mcemx32b.zip`
Installing the compilers

If you plan to do extensive development, you should put the emx development tools in a partition using OS/2’s HPFS.

Use the steps below to install the compilers.

1. Change to the drive where you plan to install the compilers. Then change to the root of that drive.
   
   Type: `cd `

2. Unpack the ZIP file and create the required directory tree.

   Type: `c:\mcidas\tools\pkunzipf -d medomx32b.zip`

3. Set up the environment.

   Type: `cd \emx`

   The \emx directory contains a script called setemx.cmd. Running this script sets up the emx environment variables so you can compile your code within the currently displayed OS/2 window only.

4. Edit the first line of the setemx.cmd script, replacing the variable named drive with the drive letter where emx is installed. For example, if emx resides on your C: drive the first line should be:

   Type: `@set drive=C`

5. If you do a large amount of local development, you may prefer to make the emx compilers accessible from any OS/2 window. To do this, copy the set command lines from the setemx.cmd script and place them at the end of the \config.sys file, substituting the correct drive letter for each occurrence of %drive%. If you choose this method, be very careful to insert only the set command lines into the \config.sys file.

6. Run the McIDAS-OS2 SETUP program.

7. Exit OS/2 using its Shutdown procedure; then restart the workstation. This step will ensure that the required .DLL files are available.

See Chapter 1 of the McIDAS-OS2 User’s Guide for more information about the SETUP program.
Compiling source files

The steps you take to compile source files vary depending on whether the source file is written in C or Fortran. A source file written in C directly calls the gcc compiler. Since SSEC does not distribute a Fortran compiler, McIDAS-OS2 code written in Fortran must be converted to C using the emx utility f2c. You can then compile the resulting code with the C compiler gcc.

The table below lists three frequently-used options for the f2c utility.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-72</td>
<td>treats text after column 72 as an error</td>
</tr>
<tr>
<td>-Idir</td>
<td>searches alternate directories for include files</td>
</tr>
<tr>
<td>-R</td>
<td>does not promote REAL functions and operations to DOUBLE PRECISION</td>
</tr>
</tbody>
</table>

The table below lists gcc options that are frequently used for compiling and linking code.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>compiles only, no linking</td>
</tr>
<tr>
<td>-Dmacro</td>
<td>defines a preprocessor macro</td>
</tr>
<tr>
<td>-Idir</td>
<td>adds dir to the list of directories to search for the include files</td>
</tr>
<tr>
<td>-o executable</td>
<td>specifies a name for the resulting command executable</td>
</tr>
<tr>
<td>-Ldir</td>
<td>adds dir to the list of directories to search for the library files</td>
</tr>
<tr>
<td>-library</td>
<td>includes specified library in the link step</td>
</tr>
<tr>
<td>-s</td>
<td>strips the symbol table from the object code</td>
</tr>
<tr>
<td>-Zdll</td>
<td>uses the compile option for dynamic link modules</td>
</tr>
<tr>
<td>-Zmtid</td>
<td>uses the code with multi-threaded programs</td>
</tr>
<tr>
<td>-Zomf</td>
<td>uses the emx.dll Dynamic Link Library for making system calls</td>
</tr>
</tbody>
</table>
For example, to build the McIDAS command bar.exe from the source file bar.pgm, you must first convert the Fortran program to the C source file named bar.c., as shown in the example below.

```
copy bar.pgm bar.f
f2c -R -I. -I\mcidas\source bar.f
```

Next, use the C compiler gcc to compile and link the source file bar.c.

```
gcc -O2 -s -Zomf -Zmtd -D_ST_MT_ERRNO -I. -I\mcidas\source
-1\mcidas\tools\libmco.lib -1f2c -1emxio -lwrap -los2
-slpacker /PMTYPE:VIO/EXEPACK:2 \mcidas\tools\main.obj
\mcidas\tools\errno.obj bar.c -O\mcidas\user\code\bar.exe
```

Note that since the bar.exe command was originally a Fortran program, you must link the object file to \mcidas\tools\main.obj. If this command was written in C, you would not have to link the object file. After you compile a Fortran module and use the f2c utility to convert it to C, you must delete the C version of the source file.

If your project contains several source files, you may want to use a project utility. The emx package includes a project management utility called dmake you can use to perform these compilations. If you are unfamiliar with make utilities SSEC recommends the book Managing Projects with make (O'Reilly 1991).

The mcidas\tools directory contains a makefile template called makedev.os2. This template specifies the rules needed to compile and link McIDAS commands, functions, and DLLs.

To use the makefile template, modify the paths to your source code and libraries that appear as macros in the beginning of the file. Name the file makefile and move it to your mcidas\working directory.

To compile and link the makefile, enter the command below.

```
dmake bar.exe
```
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 `emx` command `emxomfar` to put the 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 the `gcc` compiler to compile your function and create the object file `foo.obj`.

   ```
   gcc -O2 -s -Zomf -Zmtd -D_ST_MT_ERRNO -c -I. -I\mcidas\source foo.c
   ```

2. Move the object file to the library `libdev.a`. Use the `-p64` option to set the page size to 64 bytes and the `-r` option to replace any existing version of `foo.obj` that exists in `libdev.lib`.

   ```
   emxomfar -p64 -r libdev.lib foo.obj
   ```

As an alternative, you can use the makefile described in the previous section. Using the makefile, you only need to enter the command below.

   ```
   dmake foo.obj
   ```
Building dynamic link libraries

Unlike McIDAS-X, McIDAS-OS2 supports the use of dynamic link libraries (DLLs). DLLs are used throughout the navigation and calibration subsystems of McIDAS. The gee compiler uses slightly different compile and link arguments for DLLs.

The example below shows the steps for compiling and linking a new calibration type called FRED. The source file kbxfred.dlm contains the calibration code.

1. Convert the Fortran code to C using the f2c utility.

   copy kbxfred.dlm kbxfred.f
   f2c -R -I. -1\mcidas\source kbxfred.f

2. Use the gee compiler to compile the C source file kbxfred.c three times, once for each calibration slot.

   gcc -02 -s -Zomf -Zmtd -D_ST_MT_ERRNO -llibmcl.lib -lf2c -lemxio
      -1\wrap -1\os2 -2\linker /\FMTYPE:VIO/\EXEPACK:2 -Zdill
      \mcidas\tools\dummain.obj \mcidas\tools\errno.obj
      \mcidas\source\callib.def -o \mcidas\code\kbfred.dll

   gcc -02 -s -Zomf -Zmtd -D_ST_MT_ERRNO -llibmcl.lib -lf2c -lemxio
      -1\wrap -1\os2 -2\linker /\FMTYPE:VIO/\EXEPACK:2 -Zdill
      \mcidas\tools\dummain.obj \mcidas\tools\errno.obj
      \mcidas\source\callib.def -o \mcidas\code\kbf2red.dll

   gcc -02 -s -Zomf -Zmtd -D_ST_MT_ERRNO -llibmcl.lib -lf2c -lemxio
      -1\wrap -1\os2 -2\linker /\FMTYPE:VIO/\EXEPACK:2 -Zdill
      \mcidas\tools\dummain.obj \mcidas\tools\errno.obj
      \mcidas\source\callib.def -o \mcidas\code\kbf3red.dll

As an alternative, you can use the makefile described in the previous section. Using the makefile, you only need to enter the command below.

   dmake kbxfred.dll
Making program helps

The McIDAS command HELP allows users to quickly see the online structure of a command's argument list. When you write a McIDAS 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 McIDAS HELP command. The help files included in the McIDAS upgrade are stored in \mcidas\help.

Use the utility mcmkhelp to produce a help file (*.hlp) from a command's source code. For example, the script below creates a help file from the source file bar.pgm then stores the resulting help template in the mcidas\help directory.

```
mcmkhelp < bar.pgm > \mcidas\help\bar.hlp
```

For more information about the McIDAS HELP command, see the McIDAS-OS2 User's Guide. For more information about the HELP command format, see Chapter 2 of this manual.
This chapter describes many of the McIDAS 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 windows
- determining the number of display frames
- converting physical units such as kilometers to nautical miles
- computing meteorological parameters such as potential temperature

Contents

<table>
<thead>
<tr>
<th>Overview</th>
<th>5-3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User interface utilities</strong></td>
<td>5-5</td>
</tr>
<tr>
<td>Command line</td>
<td>5-5</td>
</tr>
<tr>
<td>Pointing device</td>
<td>5-18</td>
</tr>
<tr>
<td><strong>Display utilities</strong></td>
<td>5-21</td>
</tr>
<tr>
<td>Text messages</td>
<td>5-22</td>
</tr>
<tr>
<td>Images</td>
<td>5-25</td>
</tr>
<tr>
<td>Graphics</td>
<td>5-27</td>
</tr>
<tr>
<td>Display characteristics</td>
<td>5-53</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System utilities</strong></td>
<td></td>
</tr>
<tr>
<td>String tables</td>
<td>5-56</td>
</tr>
<tr>
<td>User Common</td>
<td>5-61</td>
</tr>
<tr>
<td>Starting McIDAS commands from applications</td>
<td>5-67</td>
</tr>
<tr>
<td>Missing value codes</td>
<td>5-69</td>
</tr>
<tr>
<td>Error handling</td>
<td>5-70</td>
</tr>
<tr>
<td>Suspending applications</td>
<td>5-72</td>
</tr>
<tr>
<td>File locks</td>
<td>5-73</td>
</tr>
<tr>
<td><strong>Conversion utilities</strong></td>
<td></td>
</tr>
<tr>
<td>Byte- and Word-level manipulation</td>
<td>5-75</td>
</tr>
<tr>
<td>Character-string manipulation</td>
<td>5-85</td>
</tr>
<tr>
<td>Day and time</td>
<td>5-97</td>
</tr>
<tr>
<td>Latitude and longitude</td>
<td>5-104</td>
</tr>
<tr>
<td>Physical units</td>
<td>5-105</td>
</tr>
<tr>
<td><strong>Scientific utilities</strong></td>
<td></td>
</tr>
<tr>
<td>Computing isentropic surfaces</td>
<td>5-109</td>
</tr>
<tr>
<td>Computing heat index</td>
<td>5-110</td>
</tr>
<tr>
<td>Computing relative humidity</td>
<td>5-110</td>
</tr>
<tr>
<td>Computing wind chill</td>
<td>5-111</td>
</tr>
<tr>
<td>Computing mixing ratio</td>
<td>5-111</td>
</tr>
<tr>
<td>Computing stability parameters</td>
<td>5-112</td>
</tr>
</tbody>
</table>
Overview

The McIDAS 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 session, a display similar to the one shown in Figure 5-1 appears on the screen. The window in the foreground is the McIDAS Text and Command Window. The window in the background is the McIDAS Image Window.

Figure 5-1. At start-up, the McIDAS display contains the McIDAS Text and Command Window and the McIDAS 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 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 windows, and obtaining information about the McIDAS display.

- **System utilities**
The system utilities include functions for determining the terminal characteristics and the state of the McIDAS session, such as the number of color levels and frames, and for making applications interact predictably with the operating system, the McIDAS 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 software.

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

---

*McIDAS Utilities 5-4*  
*McIDAS Programmer’s Manual*  
*Issued 8/97*
User interface utilities

Users ultimately interact with McIDAS in one of two ways:

- the McIDAS command line
- the pointing device, or mouse

This section describes each part of the McIDAS 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 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 software.

Command line

McIDAS applications are command-line driven. McIDAS 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 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. In McIDAS-X, the number of characters permitted in a command line is workstation dependent, although there is no practical limit. In McIDAS-OS2, the number of characters permitted in a command line is 512, although the length of the command has no practical limit.
A McIDAS command line may contain one or more of the following:

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

A sample McIDAS command line is shown below.

```
MYPROG PLOT 72747 72203 PARAM= TTD SPD "DATA FROM INL TO PBI"
```

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 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 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 has five global keywords, which are common to all McIDAS commands:

- DEV= specifies the destination device of the text output generated by a command.
- FONT= specifies the font for drawing text on the McIDAS Image Window.
- TCOL= specifies the color for command text output in the text window.
- TWIN= specifies the destination text window to route the command output to.
- VIRT= specifies a virtual graphics number to write graphical output to.

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-X or McIDAS-OS2 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 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.
Functions for extracting values from a command line

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

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mccmddbl</td>
<td>mccmddbl</td>
<td>extracts a value as a double precision number</td>
</tr>
<tr>
<td>Mccmddhr</td>
<td>mccmddhr</td>
<td>extracts a time value as a double precision number in units of hours</td>
</tr>
<tr>
<td>Mccmddll</td>
<td>mccmddll</td>
<td>extracts a latitude or longitude value as a double precision number in units of degrees</td>
</tr>
<tr>
<td>Mccmdihr</td>
<td>mccmdihr</td>
<td>extracts a time value as an integer value in the hhmmss format</td>
</tr>
<tr>
<td>Mccmdiill</td>
<td>mccmdiill</td>
<td>extracts a latitude or longitude value as an integer value in the dddmms format</td>
</tr>
<tr>
<td>Mccmdint</td>
<td>mccmdint</td>
<td>extracts a value as an integer</td>
</tr>
<tr>
<td>Mccmdiyd</td>
<td>mccmdiyd</td>
<td>extracts a day value as an integer value in the Julian day format ccyyddd</td>
</tr>
<tr>
<td>Mccmdquo</td>
<td>mccmdquo</td>
<td>extracts a character string value from the quote field</td>
</tr>
<tr>
<td>Mccmdstr</td>
<td>mccmdstr</td>
<td>extracts a character string value</td>
</tr>
</tbody>
</table>

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:

- a keyword name, which can be a blank string (" ") if you are retrieving a command positional parameter
- a position number, 1-based
- a default value
- a variable to store the result
These functions, except `mcmdstr/mccmdstr`, also expect the following additional information:

- a 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

- a 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 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 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 `FRAME` to denote only the first three characters are mandatory. The table below shows some acceptable and unacceptable forms the user can enter.

<table>
<thead>
<tr>
<th>Acceptable forms</th>
<th>Unacceptable forms</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FRA=</code></td>
<td><code>FR=</code></td>
</tr>
<tr>
<td><code>FRAM=</code></td>
<td><code>FRAEM=</code></td>
</tr>
<tr>
<td><code>FRAMENUM=</code></td>
<td></td>
</tr>
</tbody>
</table>

Keyword names can be letters or numbers, as long as the name is unique; that is, you can’t use `COLOR` and `COLUMN` 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 `ABCDEFGHJKLMN`, 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 --> ABCDEFGHIJKLMNOP
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('HEAD', 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.

```plaintext
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.

```plaintext
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, **mccmdhr** 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 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.

```plaintext
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.

```plaintext
double precision endtime
ok = mccmdhdr('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 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.

```plaintext
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.

```
cchar*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.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mccmd</td>
<td>mccmd</td>
<td>returns the entire user command in one string</td>
</tr>
<tr>
<td>Mccmdkey</td>
<td>mccmdkey</td>
<td>verifies that the keywords entered by the user are valid</td>
</tr>
<tr>
<td>Mccmdnam</td>
<td>mccmdnam</td>
<td>returns a list of the keywords entered by the user</td>
</tr>
<tr>
<td>Mccmdnum</td>
<td>mccmdnum</td>
<td>returns the number of parameters entered for a particular keyword</td>
</tr>
</tbody>
</table>

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 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 **mcmdkey** 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, **mcmdkey** checks your application to verify that it contains no ambiguous keywords; for example, COL.0R and COL.UMN.

The global keywords described earlier in this section are implicitly included in **mcmdkey**, so you don’t need to include them in your list of keywords to be tested. Your cannot use the **mcmdkey** 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 **mcmdkey**.

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

```plaintext
parameter (MAXKW = 5)
character*12 validkw(MAXKW)
data validkw/'END','FILE','PARAMETER','START','TITLE'/
ok = mcmdkey(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 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 **mcmdkey** consistent with the keywords actually used by the application.
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.

```plaintext
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:

```plaintext
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 nParms

nParms = mccmdnum('PAR.AMETER')
```

When the sample TEST command runs, `nParms` will contain the value 2.

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

```
nParms = 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.

<table>
<thead>
<tr>
<th>Status code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-]0bcd</td>
<td>argument comes from the default</td>
</tr>
<tr>
<td>[-]1bcd</td>
<td>argument comes from the command line</td>
</tr>
<tr>
<td>[-]2bcd</td>
<td>argument comes from the McIDAS string table</td>
</tr>
<tr>
<td>[-]a00d</td>
<td>character string argument</td>
</tr>
<tr>
<td>[-]a01d</td>
<td>quote field string argument</td>
</tr>
<tr>
<td>[-]a10d</td>
<td>integer argument</td>
</tr>
<tr>
<td>[-]a11d</td>
<td>integer hexadecimal argument</td>
</tr>
<tr>
<td>[-]a20d</td>
<td>double decimal argument</td>
</tr>
<tr>
<td>[-]a21d</td>
<td>double hexadecimal argument</td>
</tr>
<tr>
<td>[-]a30d</td>
<td>date argument</td>
</tr>
<tr>
<td>[-]a31d</td>
<td>current date argument</td>
</tr>
<tr>
<td>- a32d</td>
<td>year within date argument is invalid</td>
</tr>
<tr>
<td>- a33d</td>
<td><em>mon</em> month within date argument is invalid</td>
</tr>
<tr>
<td>- a34d</td>
<td><em>mm</em> month within date argument is invalid</td>
</tr>
<tr>
<td>- a35d</td>
<td>day of month (<em>dd</em>) within date argument is invalid</td>
</tr>
<tr>
<td>- a36d</td>
<td>day of year (<em>ddd</em>) within date argument is invalid</td>
</tr>
<tr>
<td>[-]a40d</td>
<td>integer time argument</td>
</tr>
<tr>
<td>[-]a41d</td>
<td>current integer time argument</td>
</tr>
<tr>
<td>- a42d</td>
<td>hours within integer time argument are invalid</td>
</tr>
<tr>
<td>- a43d</td>
<td>minutes within integer time argument are invalid</td>
</tr>
<tr>
<td>- a44d</td>
<td>seconds within integer time argument are invalid</td>
</tr>
<tr>
<td>[-]a45d</td>
<td>double time argument</td>
</tr>
<tr>
<td>Status code</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>-a46d</td>
<td>current double time argument</td>
</tr>
<tr>
<td>-a47d</td>
<td>hours within double time argument are invalid</td>
</tr>
<tr>
<td>-a48d</td>
<td>minutes within double time argument are invalid</td>
</tr>
<tr>
<td>-a49d</td>
<td>seconds within double time argument are invalid</td>
</tr>
<tr>
<td>-a50d</td>
<td>integer lat/lon argument</td>
</tr>
<tr>
<td>-a52d</td>
<td>degrees within integer lat/lon argument are invalid</td>
</tr>
<tr>
<td>-a53d</td>
<td>minutes within integer lat/lon argument are invalid</td>
</tr>
<tr>
<td>-a54d</td>
<td>seconds within integer lat/lon argument are invalid</td>
</tr>
<tr>
<td>-a55d</td>
<td>double lat/lon argument</td>
</tr>
<tr>
<td>-a57d</td>
<td>degrees within double lat/lon argument are invalid</td>
</tr>
<tr>
<td>-a58d</td>
<td>minutes within double lat/lon argument are invalid</td>
</tr>
<tr>
<td>-a59d</td>
<td>seconds within double lat/lon argument are invalid</td>
</tr>
<tr>
<td>-90d</td>
<td>keyword status</td>
</tr>
<tr>
<td>abc0</td>
<td>argument is ok</td>
</tr>
<tr>
<td>-abc1</td>
<td>argument has an invalid character</td>
</tr>
<tr>
<td>-abc2</td>
<td>integer argument can't contain a fraction</td>
</tr>
<tr>
<td>-abc3</td>
<td>argument exceeds system limits for desired format</td>
</tr>
<tr>
<td>-abc4</td>
<td>out-of-range argument &lt; given min</td>
</tr>
<tr>
<td>-abc5</td>
<td>out-of-range argument &gt; given max</td>
</tr>
</tbody>
</table>
Pointing device

The standard pointing devices for McIDAS-X and -OS2 are different.

♦ Unix workstations and X Terminals running McIDAS-X usually have a three-button mouse; the leftmost button is used by the window manager and the middle and right buttons are used by the McIDAS mouse interface.

♦ McIDAS-OS2 workstations have a two-button mouse; the mouse buttons are shared between the window manager and the application.

An anomaly may occur if a workstation with a two-button mouse is used as an X server for McIDAS-X. In this case, the left button is used by the window manager and the right button must perform the tasks normally performed by the right button. You must press both buttons to perform the tasks normally performed by the middle mouse button.

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

The McIDAS function for interfacing with the mouse is memoubtn, 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 functions putcur, megetcur, sizcur, colcur and typecur in your applications to define the cursor’s location, size, color and type. The function megetcur 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:

- a mouse button was up, but is now down
- a mouse button is being held down
- a 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 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 ZLM command, monitors the mouse event status.

```plaintext
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 \texttt{mcmbtm} function also indicates the status of the keyboard toggles Alt G and Alt Q, which are monitored by McIDAS similar to the mouse buttons. If you hold down the Alt key on the keyboard and press either the G or Q key, \texttt{mcmbtm} 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 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 cursor.

<table>
<thead>
<tr>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>putcur</td>
<td>moves the cursor to a specified location; in McIDAS-OS2, this locks the cursor, just like entering the Alt P command</td>
</tr>
<tr>
<td>sizcur</td>
<td>changes the cursor size</td>
</tr>
<tr>
<td>colcur</td>
<td>changes the cursor color</td>
</tr>
<tr>
<td>typcur</td>
<td>changes the cursor type; for example, crosshair or box</td>
</tr>
</tbody>
</table>

The values that these functions place into User Common are used by the \texttt{mcimage} program to define the appearance of the McIDAS cursor. The example below illustrates the use of these cursor functions.

```fortran
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 )
...```

McIDAS Utilities
5-20

McIDAS Programmer's Manual
Issued 8/97
Display utilities

The McIDAS library provides many functions for displaying text on the McIDAS Text and Command Window and displaying images and graphics on the McIDAS 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 windows, and to obtain information about the McIDAS display.

For additional information about the functions described in this section, see the online man pages provided with the McIDAS software.
McIDAS 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 SL command in Figure 5-2 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 IGU command in Figure 5-2.

- **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 MSL command in Figure 5-2.

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 Text and Command Window to view that much output and even if the debug messages are suppressed, they will increase CPU usage unnecessarily.

**Figure 5-2.** The McIDAS 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 global keyword, DEV, to specify the destination device of the text output generated by a command.

<table>
<thead>
<tr>
<th>This DEV = option</th>
<th>Sends the output to</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEV = C</td>
<td>McIDAS Text and Command Window (default for standard and error messages)</td>
</tr>
<tr>
<td>DEV = P</td>
<td>printer</td>
</tr>
<tr>
<td>DEV = F</td>
<td>disk file; 80 characters per line</td>
</tr>
<tr>
<td>DEV = N</td>
<td>all output is discarded (default for debug messages)</td>
</tr>
<tr>
<td>DEV = T</td>
<td>text file</td>
</tr>
</tbody>
</table>

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 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 Text and Command Window and the debug messages are suppressed.

For more information about McIDAS global keywords, see the McIDAS-X or McIDAS-OS2 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.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Message type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mcprintf</td>
<td>sdest</td>
<td>standard text</td>
</tr>
<tr>
<td>Mcerror</td>
<td>edest</td>
<td>error</td>
</tr>
<tr>
<td>Mcdebug</td>
<td>ddest</td>
<td>debug</td>
</tr>
</tbody>
</table>
The sample code below illustrates the use of each message type in a Fortran application.

```fortran
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(' ',i,1,1,parameter,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.

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

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.

```fortran
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.

```c
(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 `Medprintf`.
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 Image Window, as shown in Figure 5-3.

Figure 5-3. The McIDAS Image Window displays McIDAS-generated images.

You will use the dfline function to write a line of image data into a frame object for display on the McIDAS 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 shared memory. The number and size of the frame objects allowed per McIDAS session are user-configurable.
The sample code below illustrates how to use **dfl**ine to write a frame object. Note that the **dfl**ine function writes an entire line.

```fortran
integer array(1000)
.
.
C --- get the image frame number
frame = luc(51)

C --- get the size of the frame
status = mcfsizes(frame, num_lines, num elems)
if( status lt 0 ) then
    call epest('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 dfline(frame, line, array)
```

The output from this code displays a series of black-to-white gray shades. It may not map to the McIDAS Image Window quite as you expect, however. McIDAS applications envision the McIDAS 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 applications use the **dfl**ine 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 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 Image Window.

If you are writing an application solely for McIDAS-X in Fortran, use the **mc**line interface.

For more information about McIDAS image display characteristics, see the McIDAS TERM command with the SCALE option in the McIDAS-X or McIDAS-OS2 User's Guide. Also see the section titled Displaying Images and Graphics in the Introduction chapter of the McIDAS-X User's Guide.

For more information about McIDAS shared memory and frame objects, see the section titled Shared memory in Chapter 2, Learning the Basics.
McIDAS graphics appear to the user as text, symbols and line segments drawn in color on the McIDAS 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 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 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 graphics functions used on the following pages. Additional graphics functions for displaying wind barbs and wind vectors on a frame are provided later in this section.

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

This section describes the techniques that you will most often use when working with the McIDAS graphics subsystem. Figure 5-4 below shows a graphic produced by the sample GRAF application when run from a McIDAS session with no arguments and a frame size of 480 lines by 640 elements.

Figure 5-4. This GRAF.PCM output is the result of a McIDAS 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 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
C // Initialize the plot and fetch the color
    frame = luc(-1)
    call initpl(frame, width)
    if( mccmdint('COLOR', 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 5-4 is generated by these lines of code (91 to 132) in GRAF.PGM:

```c
call initpl(frame, width)
...
call plot(w_lrlin, w_ulele, PEN_UP)
call plot(w_lrlin, w_irele, color)
call enpct
```

The second `plot` call draws the line in color level `color` from the previous cursor position to line `w_lrlin` and element `w_irele`, 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_lrlin, 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.

```fortran
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 empty...
```

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 session starts, the dash pattern is preset to 10 pixels on and 10 pixels off. You can change this pattern using the McIDAS GD command, either from the command line before running any graphics applications, or within an application using the `keyin` or `Mckeyin` functions.

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

```fortran
if( border .gt. 0 ) then
    call qgdash( old_dash )
call dshon
call box( u_ullin, u_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 library provides two graphics functions, `wrtext` and `pltfig`, for writing text and numbers on a graphics frame. When using these functions, you must designate the following:

- the height of the characters
- the color level
- the location for the output: upper-left justified for `wrtext` and lower-right justified for `pltfig`

Lines 361 to 386 in GRAF.PGM use `wrtext` to produce the alphanumeric characters shown in Figure 5-4 through its own `plttext` subroutine, as shown below.

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

call qsclip(sclstat)
if(sclstat.eq.1) then
   call sclptnt(clin, cele, txcclin, txtcele)
else
   txcclin = clin
   txtcele = cele
endif

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 = txcclin + 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 qsclip() 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 pltext 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 plttext 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 or 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 Image Window and frees all associated resources. To create more graphics on this or other frames, you will have to call initplt again.
Advanced McIDAS graphics techniques

This section describes some advanced techniques that you can use with the McIDAS graphics subsystem. It builds upon the example and McIDAS-X session started in the previous section, Basic McIDAS 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 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 5-5 below.

Figure 5-5. This GRAF.PGM output is the result of a McIDAS 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:

```c
call mcfsize( frame, f_nlines, 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 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 supports the use of world coordinates with the *page*, *selon*, *seloff*, *selpnt* and *sclhgt* functions.

To use world coordinates properly, you must understand the concepts of *defining* and *activating* worlds. The *selon* function activates world coordinates (scaling) and *seloff* 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*, *ptlbrb*, *pltdig*, *wrtext*) use the current system, whether frame or world. Frame coordinates are determined entirely by the hardware and/or the McIDAS 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 5-5, 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
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 ullam ) .lt. 1100 ) return
    ... three more argfetches deleted...
call page( w ullam, w ulele, w lrlin, w lrele, SET_WORLD )
end if
```

The resulting output is shown in Figure 5-6 below.
Figure 5-6. This GRAF.PGM output uses world coordinates to position the text.

The diagonal now goes from corner to corner as in Figure 5-4 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 \texttt{page} function. If a 480 by 640 world is suitable, you can just enter the calls below to activate \texttt{initpl}'s default world.

\begin{verbatim}
call initpl( frame, width )
call sclo
\end{verbatim}

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 \texttt{wrtex} 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 \texttt{scpnt} and \texttt{sclhgt} functions allow explicit conversion of points and text heights from world to frame coordinates. The \texttt{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:

\begin{verbatim}
C --- Determine the actual (frame coordinates) height of the
C --- text. (The use of local variable 'txtthgt' is needed to keep
C --- plttext() from modifying its inputs.)

txtthgt = hgt
call sclhgt( txtthgt )
\end{verbatim}
The center point is then transformed to frame coordinates:

```c
C --- Determine the center point in frame coordinates. Because
C --- sclipnt() 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 sclipnt( clin, cele, txtclin, txtcele )
    else
      txtclin = clin
      txtcele = cele
    end if
```

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

```c
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 `wrtex` and then be sure to restore the scaling to its original setting, whether off or on, before returning control to the caller:

```c
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 5-7 below, which is identical to Figure 5-6 except the line segment is only drawn when it is more than 130 pixels, in world coordinates, from the edge of the world.

Figure 5-7. This GRAP.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 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 GRAP.PGM define the above viewport and draws its extent, as shown below.

```c
if( mccmdint('BOR.DER', 1, 'Border Width', 0, 0, 
& (w_lrlin-w_uullin)/3, border ).lt. 0 ) return
v_uullin = w_uullin + border
v_uulele = w_uulele + border
v_lrlin = w_lrlin - border
v_lrele = w_lrele - border
call page( v_uullin, v_uulele, v_lrlin, v_lrele, SET_VIEW )
if( border .gt. 0 ) then
call ggdash( old_dash )
call dshon
call box( v_uullin, v_uulele, 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 programming is to generate graphical output in panels, as is done by the McIDAS commands PTDISP 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  C  *** McIDAS Revision History ***
3  C  *** McIDAS Revision History ***
4  C  ? GRAF -- Demo of basic McIDAS graphics
5  C  ?  GRAF <keywords>
6  C  ? Parameters:
7  C  ?  (none)
8  C  ?  Keywords:
9  C  ?  BORDER = | border width, in pixels
10  C  ?  COLOR = | graphics color level
11  C  ?  WORLD = | mn_lin mn_ele mx_lin mx_ele to define and
12  C  ?  activate a world
13  C  Remarks
14  C  ?  This demo nominally draws a diagonal line from
15  C  ?  the upper left to the lower right of the display.
16  C  ?
17  C  ?  If WORLD= arguments are not specified, the initpl()
18  C  ?  default world of 480 lines by 640 elements is defined
19  C  ?  but not activated. The diagonal line will be correct
20  C  ?  only if the display size is 480 by 640. If WORLD= arguments
21  C  ?  are specified the world will be defined and activated and
22  C  ?  the diagonal correct regardless of world size.
23  C  ?
24  C  ?  BORDER= defines a viewport (clipping region) the indicated
25  C  ?  number of pixels (world coordinates) from the edge of the
26  C  ?  world and outlines it with a dashed line.
27  C  ?
28  C  ?
29  C  ?
30  C  subroutine main0
31  C  implicit NONE
32  C
33  C  --- Constants and shared variables
34  integer  PEN_UP  ! move pen without drawing
35  integer  SET_VIEW  ! define a viewport
36  integer  SET_WORLD  ! define a world
37  parameter  ( PEN_UP = 0, SET_VIEW = 0, SET_WORLD = 1 )
38  C  --- local variables
39  character*80 text  ! output text buffer
40  integer  border  ! border (view port) width
41  integer  celie  ! text center element
42  integer  clin  ! text center line
43  integer  color  ! color level
44  integer  frame  ! frame number
45  integer  f_lele  ! frame lower right element
46  integer  f_lrlin  ! frame lower right line
47  integer  f_neles  ! number of elements in frame
48  integer  f_nlines  ! number of lines in frame
49  integer  f_ulele  ! frame upper left element
50  integer  f_ulinn  ! frame upper left line
51  integer  hgt  ! text height
52  integer  linspac  ! spacing between text lines
53  integer  old_dash  ! dashing mode
54  integer  scsstat  ! scaling (world) mode
55  integer  v_lele  ! lower left viewport element
56  integer  v_lrlin  ! lower left viewport line
57  integer  v_ulele  ! upper left viewport element
58  integer  v_ulinn  ! upper left viewport line
59  integer  w_lele  ! lower right world element
60  integer  w_lrlin  ! lower right world line
61  integer  w_ulele  ! upper left world element
62  integer  w_ulinn  ! upper left world line

McIDAS Utilities
5-38

McIDAS Programmer's Manual
Issued 8/97
real elew2f ! element frame:world ratio
real linw2f ! line frame:world ratio
C --- External functions
integer luc ! user common peek
integer mccmdint ! integer argument fetch
integer mccmdnum ! number of args for keyword
C --- Initialized variables
integer width ! line width (session current)
data width / 0/
C -------------------------------------------------------
C INITIALIZE
C -------------------------------------------------------
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
C -------------------------------------------------------
C DEFINE THE WORLD AND DRAW BORDER AND DIAGONAL
C -------------------------------------------------------
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
  if( mccmdint('WORLD',2, 'World upper left element',
  & 1, 1, 0, w ulele ) .lt. 1100 ) return
  if( mccmdint('WORLD',3, 'World lower right line',
  & 1, 1, 0, w lrlin ) .lt. 1100 ) return
  if( mccmdint('WORLD',4, 'World lower right element',
  & 1, 1, 0, w lrele ) .lt. 1100 ) return
  call page( w ullin, w ulele, w lrlin, w lrele, SET_WORLD )
end if
C // get the world definition. Note that there
C // is always a defined world, whether by initpl() or
C // page().
call world( w ullin, w ulele, w lrlin, w lrele )
C // draw the border and diagonal
call qgdash( old dash )
call dshoff
call box( w ullin, w ulele, w lrlin, w lrele, color )
call plot( w ullin, w ulele, PEN UP )
call plot( w lrlin, w lrele, color )
call enpt
if( old dash .ne. 0 ) call dshon
C DEFINE THE VIEWPORT AND DRAW BORDER
C -------------------------------------------------------
C mcidas Programmer's Manual
Issued 8/97
mcidas Utilities
5.39
if ( mcmdint('BORDER', 1, 'Border Width', 0, 0, 
  & (w_lroll-w_uroll)/3, border ).lt. 0 ) return 
v_uroll = w_uroll + border 
v_uroll = w_uroll + border 
v_lroll = w_lroll - border 
v_lroll = w_lroll - border 
call page( v_uroll, v_uroll, v_lroll, v_lroll, SET_VIEW ) 
if( border .gt. 0 ) then 
call qggraph( old_dash ) 
call dshon 
call box( v_uroll, v_uroll, v_lroll, v_lroll, color ) 
if( old_dash .eq. 0 ) call dshoff 
end if 

C GENERATE AND DISPLAY THE TEXT MESSAGES 
C ------------------------------------------------------ 
C // Fetch the text height and use it and the scale factors 
C // to determine the separation between text lines. If the 
C // line scale factor is less than one, the text locations 
C // will be moved closer together by the scaling but the 
C // text height will be unchanged. So increase the separation 
C // between lines (in world coordinates) to prevent overwrite. 
C if( mcmdint('HGT', 1, 'Text Height', 10, 5, 20, 
  & hgt ).lt. 0 ) return 
call qsfrm( scldstat ) 
call scifac( linw2f, elow2f ) 
if( scldstat.eq. 1 .and. linw2f.lt. 1.0 ) then 
  linwmt = 1.5*hgt / linw2f 
else 
  linwmt = 1.5*hgt 
end if 
C // Determine the center of the topmost five lines. 
C // Center the third line in the world; this will illustrate 
C // the main use of world coordinates to preserve centering 
C // even when display sizes differ. 
call world( w_uroll, w_uroll, w_lroll, w_lroll ) 
clin = w_uroll + (w_lroll-w_uroll)/2 - 2*linwmt 
cele = w_uroll + (w_lroll-w_uroll)/2 
C // Display scaling status. 
call qsfrm (scldstat) 
if( scldstat.eq. 1 ) then 
  text = 'world is ACTIVE' 
else 
  text = 'world is INACTIVE' 
end if 
call ptttext( clin, cele, hgt, color, text ) 
C // Get and display frame size. 
call mcfsrsize( frame, f_nlines, f_neles ) 
f_uroll = f_uroll + 1 
f_uroll = f_uroll + 1 
f_lroll = f_nlines + f_uroll - 1 
f_lroll = f_nlines + f_uroll - 1 
write(30,30) f_uroll, f_uroll, f_lroll, f_lroll, f_lroll 
format('DISPLAY ',415) 
clin = clin + linwmt 
call ptttext( clin, cele, hgt, color, text ) 

C // Display world size. 
call world( w_uroll, w_uroll, w_lroll, w_lroll )
write(text,32) w_ullin, w_ulele, w_lrlin, w_lrele
32 format('WORLD ',4I5)
colin = colin + linspac
call pittext( colin, cele, hgt, color, text )

c // Display viewport size.
write(text,34) v_ullin, v_ulele, v_lrlin, v_lrele
34 format('VIEWPORT ',4I5)
colin = colin + linspac
call pittext( colin, cele, hgt, color, text )

C --------------------------------------------
C CLEAN-UP
C --------------------------------------------
call endplt

call sdest('done',0)
** return
**

** Name:
** box - draw a box on the display
**
** Interface:
** subroutine
** box(integer ullin, integer ulele, integer lrlin,
**    integer lrele, integer color)
**
** Input:
** ullin - upper left line
** ulele - upper left element
** lrlin - lower right line
** lrele - lower right element
** color - color of box
**
** Input and Output:
** none
**
** Output:
** none
**
** Return values:
** none
**
** Remarks:
** Input coordinates will be interpreted as either frame or
** world system depending upon whether the caller has activated
** a world or not.
**
** Categories:
** graphic

subroutine box( ullin, ulele, lrlin, lrele, color )

implicit NONE

C --- interface variables
integer ullin
integer ulele
integer lrlin
integer lrele
integer color

C --- symbolic constants and shared values
integer PEN_UP ! move pen without drawing
parameter (PEN_UP = 0)

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 emnt

return
end

** Name:
** pltttest - Center a line of text on the indicated location
**
** Interface:
** subroutine
** pltttest(integer clin, integer cele, integer hgt,
** integer color, character(*) text)
**
** Input:
** clin - center line
** cele - center element
** hgt - text height
** color - text color
** text - text string
**
** Input and Output:
** none
**
** Output:
** none
**
** Return values:
** none
**
** Remarks:
** The text will be centered on (clin,cele) in either world
** or frame coordinates, depending upon whether the caller has
** activated a world or not. If a world is active, the actual
** text will be enlarged if the world is smaller than the frame.
** If the world is larger, the text size will be unchanged.
** In order to keep the text centered, this routine does all
** of its own scaling (using schgt and sclnt) and actually
** writes the text in frame coordinates. The display is then
** returned to the initial state set by the caller.
**
** Categories:
** graphic

subroutine pltttest( clin, cele, hgt, color, text )
implicit NONE

C --- interface variables

integer clin
integer cele
integer hgt
integer color
character(*) text

C --- local variables

integer ullin ! upper left of text (frame)
integer ulele ! upper left of text (frame)
integer txthgt ! text height, frame coords
integer txtcln ! text center, frame coords
integer txtcele ! text center, frame coords
integer sclntat ! scaling status
integer nchar ! number of characters in text
C --- external functions
integer len_trim ! compute length of text
C --- Determine the actual (frame coordinates) height of the
text. (The use of local variable 'txtthgt' is needed to keep
plttext() from modifying its inputs.)
txtthgt = hgt
call sclht( txtthgt )
C --- Determine the center point in frame coordinates. Because
C --- sclpt() 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 sclpt( 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 - txtthgt/2
ulele = txtcele - txtthgt*nchar/2 + txtthgt/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 scoff
call wrtext( ullin, ulele, txtthgt, text, nchar, color )
if( sclstat .eq. 1 ) call sclon
call enpt
return
end

** Name:
** sclfac - return the world to frame scale factors
**
** Interface:
** subroutine
** sclfac(real linw2f, real elev2f)
**
** Input:
** none
**
** Input and Output:
** none
**
** Output:
** linw2f - number of frame lines per world line
** elev2f - number of frame elements per world element
**
** Return values:
** none
**
** Remarks:
** An increment in world coordinates multiplied by the
** appropriate scale factor will become an increment in
** frame coordinates.
** This routine also illustrates a helpful design principle:
** if your application needs access to the inner workings of
** a subsystem (in this case a common block), write an access
** routine to extend the API rather than just including the
** common block in your application.
**
** Categories:
** graphic

subroutine sclfac( linw2f, elew2f )
implicit NONE
C --- interface variables
real linw2f
real elew2f
C --- constants and shared variables
integer ifrm ! frame number for graphics
integer iwidth ! line width
integer mnele ! upper left world element
integer mnlin ! upper left world line
integer mxele ! lower right world element
integer mxlin ! lower right world line
integer sclfg ! scaling on ?
real sele ! frame elems per world elem
real sclin ! frame lines per world line
common /MODFRACOM/ ifrm, iwidth, mnlin, mnele, mxele, mxlin,
& sclin, sele, sclfg
C // Implementation is trivial; just capture the common
C // block values in the formal arguments.
linw2f = sclin
elew2f = sele
return
end
** Name:
** world - return the world coordinates of display corners
**
** Interface:
** subroutine
** world(integer w_ullin, integer w_uilele,
**    integer w_lrlin, integer w_lrele )

** Input:
** none
**
** Input and Output:
** none
**
** Output:
** w_ullin - upper left line of world
** w_uilele - upper left element of world
** w_lrlin - lower right line of world
** w_lrele - lower right element of world

** Return values:
** none
**
** Remarks:
** See sclfac() for a design note.
**
** Categories:
** graphic

```fortran
subroutine world( w_ullin, w_ulele, w_lrlin, w_lrele)
  implicit NONE

C --- interface variables

  integer  w_ullin
  integer  w_ulele
  integer  w_lrlin
  integer  w_lrele

C --- constants and shared variables

  integer  ifrm ! frame number for graphics
  integer  iwidth ! line width
  integer  mnele ! upper left world element
  integer  mlxin ! upper left world line
  integer  mxele ! lower right world element
  integer  mxlin ! lower right world line
  integer  sclfg ! scaling on ?
  real    scale ! frame elems per world elem
  real    sclin ! frame lines per world line

  common /MOPRACOM/ ifrm, iwidth, mlxin, mnele, mxele, mxlin, sclfg
  & sclin, scale, sclfg

C // Implementation is trivial: just capture the common

C // block values in the formal arguments.

  w_ullin = mlxin
  w_ulele = mnele
  w_lrlin = mxlin
  w_lrele = mxele

return
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.

<table>
<thead>
<tr>
<th>integer nr</th>
<th>!number of rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter (nr=22)</td>
<td></td>
</tr>
<tr>
<td>integer nc</td>
<td>!number of columns</td>
</tr>
<tr>
<td>parameter (nc=32)</td>
<td></td>
</tr>
<tr>
<td>integer color</td>
<td>!color of contours</td>
</tr>
<tr>
<td>integer idash</td>
<td>!line dashing control</td>
</tr>
<tr>
<td>integer iwrap</td>
<td>!wrap contours around edges?</td>
</tr>
<tr>
<td>integer lint</td>
<td>!label interval</td>
</tr>
<tr>
<td>integer lsize</td>
<td>!size (in pixels) of labels</td>
</tr>
<tr>
<td>integer smooth</td>
<td>!smoothing factor for contouring</td>
</tr>
<tr>
<td>real cint</td>
<td>!contour interval</td>
</tr>
<tr>
<td>real grid (nr, nc)</td>
<td>!grid to contour</td>
</tr>
<tr>
<td>double precision dint</td>
<td>!contour interval</td>
</tr>
<tr>
<td>character*80 lfmt</td>
<td>!label format</td>
</tr>
<tr>
<td>external ctcvcf</td>
<td>!external function call for strings</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>C---pick up cint from command line</td>
<td></td>
</tr>
<tr>
<td>if (mcxmdbl ('CIN,T',1,'Contour Interval',0,-1.010,1.010,dint) &amp; !if mcxmdbl returns less than</td>
<td></td>
</tr>
<tr>
<td>.1t. 0) return !zero and error has occurred</td>
<td></td>
</tr>
<tr>
<td>cint = sngl (dint)</td>
<td></td>
</tr>
<tr>
<td>[get grid to contour]</td>
<td>[initialize navigation with mapdef call]</td>
</tr>
<tr>
<td>lsize = 6</td>
<td>!labels are 6 pixels high</td>
</tr>
<tr>
<td>color = 4</td>
<td>!use color 4 (green)</td>
</tr>
<tr>
<td>idash = 0</td>
<td>!dash only negative values</td>
</tr>
<tr>
<td>lint = 4</td>
<td>!label very thin contour</td>
</tr>
<tr>
<td>lfmt = ('&quot;Height = ',F6.0,'meters&quot;')&quot; !format statement</td>
<td></td>
</tr>
<tr>
<td>!that controls</td>
<td></td>
</tr>
<tr>
<td>!how the contour</td>
<td></td>
</tr>
<tr>
<td>!labels look</td>
<td></td>
</tr>
<tr>
<td>smooth = 15</td>
<td>!smoothing factor</td>
</tr>
<tr>
<td>iwrap = 0</td>
<td></td>
</tr>
<tr>
<td>iret = mcgrdcon (grid,nr,nc,cint,lsize,color,idash,lint &amp; !smooth, iwrap, ctcvcf)</td>
<td></td>
</tr>
</tbody>
</table>
The `megrdec` 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.

```
C --- a common block that passes the string name for the external
C contour intervals to the external function is defined:
C
external ctcvcf          ! external function call for string
                         ! table entry
                         .

common/extcomm/cint      ! cint is a character*12 and will
                         ! contain the string table entry
                         .
                         ! holding the contour values

C --- when GRDDISP picks up the cint string,
14  if (mccmdstr ('CIN.T',1,'X',cint) .lt 0) goto 2000
15  n = nchars (cint,begin,end)
16  rc = mcsrtodbl (cint (begin:end), dint)
17  C --- if cint is not a number, then rc will be <0 and dint will be
18  C zero.
19  C This will be the case if a string table entry is being used to
20  C set the contour values.
21  if (rc.lt.0 .and. dint.eq.0) then
22    intc = lit ('EXT')
23    xint = slit ('EXT')
24    else
25      xint = snegl (dint)
26      intc = ifix (xint)
27    endif
28  .
29  .
30  C --- The call to mcgrdec is done after the gridded data have been
31  C read in from the server. Grid contains floating point data, not
32  C the scaled integers as stored in the grid array. Because the
33  C data is floating point, an f-type format in the fmt string
34  C generally will not work.
```
lsize = 6 !label size is 6 pixels
icolor = 4 !draw in color 4 (green)
idash = 0 !dash negative contours
lint = 4 !label every 4th contour
lfmt = '("Height = ",F6.0, " meters")' !format statement that
!controls the contour !labels
ismooth =15
iwrap = 0
maxlev = 0
iret = mcgrdcon (grid, nr, nc, xint, lsize, icolor, idash, lint,
& lfmt, ismoth, iwrap, maxlev, ctcvcf)
Wind barbs and wind vectors

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

<table>
<thead>
<tr>
<th>Fortran function</th>
<th>C function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcplotwind</td>
<td>Mcplotwind</td>
<td>plots wind information using barbs and flags at the designated line/element</td>
</tr>
<tr>
<td>mcsplotskywind</td>
<td>McPlotSkyWind</td>
<td>plots wind and sky cover information at a designated line/element</td>
</tr>
<tr>
<td>mcsplotwindnavll</td>
<td>McPlotWindNavLL</td>
<td>plots wind information using flags and barbs at a designated latitude/longitude; wind direction is corrected for projection</td>
</tr>
<tr>
<td>mcsplotskywindnav</td>
<td>McPlotSkyWindNav</td>
<td>plots wind and sky cover information at a designated line/element</td>
</tr>
<tr>
<td>mcsplotskywindnavll</td>
<td>McPlotSkyWindNavLL</td>
<td>plots wind and sky cover information at a designated latitude/longitude; wind direction is corrected for projection</td>
</tr>
<tr>
<td>pltwnv</td>
<td></td>
<td>plots wind information using wind vectors at the designated latitude/longitude</td>
</tr>
</tbody>
</table>

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 \texttt{meplotwind}, \texttt{meplotskywindnavll}, and \texttt{pltwvn} functions.

```fortran
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/
integer color_lev ! vector
integer sky_cover ! graphics color to use
integer frame ! sky cover WMO code value
integer hemisphere ! current frame number
integer ok

data length/40/
data color_lev/3/

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 = meplotwind(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 = meplotskywindnavll(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.

ok = mapdef(latin, tvbounds, 'SAT', navparms)

call pltwvn(nint(speed), nint(direction), 440000, 1400000,
& color_lev, length)
```

\textit{MciDAS Utilities} 5-50
\textit{MciDAS Programmer's Manual} Issued 8/97
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 = mcplothskeywindnavll(frame,latitude,longitude,length,
& color_lev, sky_cover, direction, speed)

call endplt
return
end

Figure 5-8 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

The table below lists the parameters used with the MAP command.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latitude range</td>
<td>20 N to 45 N</td>
</tr>
<tr>
<td>Longitude range</td>
<td>95 W to 165 W</td>
</tr>
<tr>
<td>Standard latitude</td>
<td>60 N</td>
</tr>
<tr>
<td>Standard longitude</td>
<td>115 W</td>
</tr>
</tbody>
</table>

Note that wind flag 1, generated by the mcplothskeywind command, makes no angle
correction based on the navigation for the map projection. However, wind
flag 2, generated by the mcplothskeywindlv command, correctly rotates the wind
direction relative to the earth to compensate for the map direction. Wind
flag 3 was generated by the pltwwv command. Note the preceding mapdef call
in the code. Windflag 4, generated by the mcplothskeywindnavll, plotted
both an angle-corrected wind flag and a sky cover symbol, as if
constructing a station model plot.
Figure 5-8. Three wind barbs and a wind vector are plotted on a conformal projection.
Display characteristics

The McIDAS display is the device used to output image and graphical data. It can be a workstation monitor or an X Terminal. Use the two functions below to inquire about the characteristics of the McIDAS display.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>not available</td>
<td>itrnch</td>
<td>obtains the value of a McIDAS display characteristic</td>
</tr>
<tr>
<td>Mcfsize</td>
<td>mcfsizex</td>
<td>obtains the size of a frame, in lines and elements</td>
</tr>
</tbody>
</table>

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 itrnch and mcfsizex 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 itrnch 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.

<table>
<thead>
<tr>
<th>Valid itrmch strings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMINAL_TYP</td>
<td>terminal display type as an integer literal of four characters, for example: &quot;WWW &quot;</td>
</tr>
<tr>
<td>OPER_SYSTEM</td>
<td>operating system of the current McIDAS session on this workstation as an integer literal of four characters, for example: &quot;UNIX&quot;</td>
</tr>
<tr>
<td>DSP_LINES</td>
<td>number of default display lines</td>
</tr>
<tr>
<td>DSP_ELEMENTS</td>
<td>number of default display elements</td>
</tr>
<tr>
<td>MAX_COLORS</td>
<td>maximum color level</td>
</tr>
<tr>
<td>MIN_COL_LEV</td>
<td>minimum valid color level</td>
</tr>
<tr>
<td>MAX_COL_LEV</td>
<td>maximum valid color level</td>
</tr>
<tr>
<td>MX_CUR_SIZ_H</td>
<td>maximum cursor height</td>
</tr>
<tr>
<td>MX_CUR_SIZ_W</td>
<td>maximum cursor width</td>
</tr>
<tr>
<td>MN_CUR_SIZ_H</td>
<td>minimum cursor height</td>
</tr>
<tr>
<td>MN_CUR_SIZ_W</td>
<td>minimum cursor width</td>
</tr>
<tr>
<td>GRAY_LEVELS</td>
<td>number of gray levels available</td>
</tr>
<tr>
<td>ANNOTAT_SIZE</td>
<td>size of the annotation</td>
</tr>
<tr>
<td>DWELL_RATE_K</td>
<td>dwell rate constant</td>
</tr>
<tr>
<td>BITS_PER_PIX</td>
<td>number of bits per pixel</td>
</tr>
<tr>
<td>IND_GRAPHIC?</td>
<td>Does the display have independent graphics?</td>
</tr>
<tr>
<td>VAR_FR_SIZE?</td>
<td>Does the display have variable frame sizes?</td>
</tr>
<tr>
<td>DISP_TOGGLE?</td>
<td>Is a display toggle required to show a frame?</td>
</tr>
<tr>
<td>ZOOM?</td>
<td>Does the display have zoom capability?</td>
</tr>
<tr>
<td>STRETCHING?</td>
<td>Is contrast stretching supported?</td>
</tr>
<tr>
<td>CUR_BOX?</td>
<td>Is the box cursor type supported?</td>
</tr>
<tr>
<td>CUR_XBOX?</td>
<td>Is the crossbox cursor type supported?</td>
</tr>
<tr>
<td>CUR_XHAIR?</td>
<td>Is the crosshair cursor type supported?</td>
</tr>
<tr>
<td>CUR_BULL?</td>
<td>Is the bullseye cursor type supported?</td>
</tr>
<tr>
<td>CUR_SOLID?</td>
<td>Is the solid box cursor type supported?</td>
</tr>
</tbody>
</table>
The **itrmch** function can be used to validate user-entered command parameters, which may not be valid on certain display types. The example below is from the McIDAS CUR command. It does not allow the user to change to a solid or bullseye type cursor on a VGA workstation.

Notice that the program doesn’t need to know that the display is a VGA; the function will return a value of false for any unsupported display type.

```plaintext
parameter my_terminal=-1
C--check for "SOLID" or "BULL" cursor on VGA --> invalid option
if (itrmch('CUR_SOLID?', my_terminal) .ne. 1) then
  call edest('Solid cursor type not supported for this hardware', 0)
endif
if (itrmch('CUR_BULL?', my_terminal) .ne. 1) then
  call edest('Bullseye cursor type not supported for this hardware', 0)
endif
```

### Obtaining the size of a frame

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

```plaintext
integer frame
integer lines
integer elems
C--- get size of the frame

  frame=4
  call mcsfsize(frame, lines, elems)

C--- upon successful completion, lines and elems will contain
C--- frame 4's screen dimensions
```
System utilities

The McIDAS 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 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 software.

String tables

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

- McIDAS users can use string tables to simplify command entry.
- McIDAS programmers can use string tables to pass information between commands run at different times.

Users can run the McIDAS commands TU, TL, TE, TD and REPEAT to manage string tables and simplify command entry. For information about these McIDAS commands, see the McIDAS-X Learning Guide and the McIDAS-X or McIDAS-OS2 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 another. 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.

<table>
<thead>
<tr>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibget</td>
<td>gets a string from the current string table</td>
</tr>
<tr>
<td>ibput</td>
<td>writes a string to the current string table</td>
</tr>
<tr>
<td>ibputc</td>
<td>forms a character string from a sequence of character tokens and writes it to the string table</td>
</tr>
</tbody>
</table>

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

Note that **ibputc** 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 **ibput** 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 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. Although you must design or choose a file format and write the I/O routines and user interface commands, such as a viewer and editor, if needed, you will gain far more flexibility in the type and volume of information that can be passed.

Examples

The code segments below demonstrate the McIDAS string table API. Sample user-level input, via the McIDAS 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 lget function extracts the contents of a McIDAS string from within an application and places it in a character variable. If the McIDAS 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 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
character*24 out_string   ! extract value from
                          ! 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 string table associated with a specified string name, as shown in the code fragment below.

```
character*12 string_name  ! name of McIDAS string to
character*24 in_string    ! place new value in
                          ! 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 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 `Ibputc` function allows a user to enter a group of character tokens into one McIDAS string. The code fragment below demonstrates this function.

```
parameter (MAXTOK = 3)
c|aracter*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 = Ibputc (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 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
```
McIDAS 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 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 processes including applications; for example, the image frame displayed.

User Common contains the dynamic state of the McIDAS session. All McIDAS 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 ltrmch function, which provides details about the static part of the McIDAS 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 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 McIDAS 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 McIDAS session. At any instant, it is the same for all McIDAS processes. All McIDAS processes, including applications, may change it at any time, and these changes are immediately visible to all other McIDAS 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 McIDAS session state taken at the time a McIDAS process or process chain starts, plus any modifications made to it by the process or chain. A process chain is a series of McIDAS 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 McIDAS 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 EAST/NHV STA=MSN
SF 4
```

The first command sets the current frame to 2; the second command requests that the latest GOES-EAST 4 km visible 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.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>McIuc</td>
<td>luc</td>
<td>returns a value from User Common</td>
</tr>
<tr>
<td>McPuc</td>
<td>puc</td>
<td>changes a value in User Common</td>
</tr>
</tbody>
</table>

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**.

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

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>McGetGraphicsFrameNumber</td>
<td>mcgetgraphicsframenumberi</td>
<td>returns the current graphics frame number for an interactive application</td>
</tr>
<tr>
<td>McGetImageFrameNumber</td>
<td>mcgetimageframenumberi</td>
<td>returns the current image frame number for an interactive application</td>
</tr>
<tr>
<td>McGetGraphicsFrameNumber</td>
<td>mcgetgraphicsframenumber</td>
<td>returns the current graphics frame number</td>
</tr>
<tr>
<td>McSetGraphicsFrameNumber</td>
<td>mcsetgraphicsframenumber</td>
<td>sets the current graphics frame number</td>
</tr>
<tr>
<td>McGetImageFrameNumber</td>
<td>mcgetimageframenumber</td>
<td>returns the current image frame</td>
</tr>
<tr>
<td>McSetImageFrameNumber</td>
<td>mcsetimageframenumber</td>
<td>sets the current image frame number</td>
</tr>
<tr>
<td>McGetMaxImageFrameNumber</td>
<td>mcgetmaximageframenumber</td>
<td>returns the maximum image frame number for the current session</td>
</tr>
<tr>
<td>McGetMaxGraphicsFrameNumber</td>
<td>mcgetmaxgraphicsframenumber</td>
<td>returns the maximum graphics frame number for the current session</td>
</tr>
<tr>
<td>McISImageLooping</td>
<td>mcisimagelooping</td>
<td>returns current image looping state</td>
</tr>
<tr>
<td>McISImageFrameOn</td>
<td>mcisimageframeon</td>
<td>returns whether or not the current image frame is visible</td>
</tr>
<tr>
<td>McSetImageFrameOn</td>
<td>mcsetimageframeon</td>
<td>sets current image frame to visible</td>
</tr>
<tr>
<td>McISImageConnectedToLoop</td>
<td>mcisimageconnectedtoloop</td>
<td>indicates if the image frames are connected to the looping system</td>
</tr>
<tr>
<td>C function</td>
<td>Fortran function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>McGetGraphicsLooping</td>
<td>mcisgraphicslooping</td>
<td>returns current graphics looping state</td>
</tr>
<tr>
<td>McGetGraphicsFrameOn</td>
<td>mcisgraphicsframeon</td>
<td>returns whether the current graphics frame is visible</td>
</tr>
<tr>
<td>McSetGraphicsFrameOn</td>
<td>mcssetgraphicsframeon</td>
<td>sets current graphics frame to visible</td>
</tr>
<tr>
<td>McIsGraphicsConnectedToLoop</td>
<td>mcisgraphicsconnectedtoloop</td>
<td>indicates whether the graphics frames are connected to the looping system</td>
</tr>
<tr>
<td>McGetStdOutputDevice</td>
<td>mcgetstdoutputdevice</td>
<td>returns the destination device for standard output messages</td>
</tr>
<tr>
<td>McGetStdErrorDevice</td>
<td>mcgetstderrordevice</td>
<td>returns the destination device for standard error messages</td>
</tr>
<tr>
<td>McGetStdDebugDevice</td>
<td>mcgetstddebugdevice</td>
<td>returns the destination device for standard debug messages</td>
</tr>
<tr>
<td>McSetStdOutputDevice</td>
<td>mcssetstdoutputdevice</td>
<td>sets the destination device for standard output messages</td>
</tr>
<tr>
<td>McSetStdDebugDevice</td>
<td>mcssetstddebugdevice</td>
<td>sets the destination device for standard debug messages</td>
</tr>
<tr>
<td>McSetStdErrorDevice</td>
<td>mcssetstderrordevice</td>
<td>sets the destination device for standard error messages</td>
</tr>
<tr>
<td>McIsMcIDASRunning</td>
<td>mcismcidasrunning</td>
<td>indicates whether a McIDAS session is active</td>
</tr>
</tbody>
</table>

User Common functions are typically used in interactive McIDAS applications. The two examples below, from the McIDAS IMGPROBE command, illustrate its use.
Determining the current frame within a valid range

The IMGPROBE sample code below performs two tasks:

- determines the currently displayed frame to use as a default
- determines the maximum image frame number to use when range checking user input

```c
--- image frame

current_frame = mcgetimageframenumber()
max_frame = mcgetmaximageframenumber
if ( mccmdint('IMA.GE', 1, 'Image Frame', current_frame),
    &
    1, max_frame, image_frame).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 IMGPROBE 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
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 commands from applications

When developing an applications program, check to see if a McIDAS command performs a needed task. If so, you should start that McIDAS 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 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 command from an application using the functions defined in the table below.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>not available</td>
<td>keyin</td>
<td>starts a command asynchronously with extended format</td>
</tr>
<tr>
<td>not available</td>
<td>skeyin</td>
<td>starts a command synchronously with extended format</td>
</tr>
<tr>
<td>Mckeyin</td>
<td>mckeyin</td>
<td>starts a command asynchronously</td>
</tr>
<tr>
<td>Mcskeyin</td>
<td>mcskeyin</td>
<td>starts a command synchronously</td>
</tr>
</tbody>
</table>

Use the **keyin**, **mckeyin** and **Mckeyin** functions to start commands asynchronously. For example, the McIDAS 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 `Mskeyin` 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 McIDAS EG 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 plotting could occur before the EG command runs, erasing part of the image.

**Starting commands synchronously**

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

Below is a code fragment from an application that uses `Mskeyin`. 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.

```c
/*
 * if there is exactly one match
 * try to put the cursor there
 */
if(founds=1)
{
  char command[100];

  sprintf( command, "PC E %f %f DEV=NNN", slat, slon);

  /*
   * if the cursor positioning was successful
   * output the position
   */
  if(Mskeyin( command )==0) (void)Mskeyin("E");
}
```

**Starting commands asynchronously**

If your application runs a command string containing an extended format, you must use the `keyin` (asynchronous) or `skeyin` (synchronous) function.
Missing Value codes

It is not uncommon to have data reports with missing values. The McIDAS 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 library provides a group of functions for assigning and verifying floating-point missing value codes. The functions are described in the table below.

<table>
<thead>
<tr>
<th>C Function</th>
<th>Fortran Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>McGetMissingDbl</td>
<td>mcgetmissingdbl</td>
<td>returns the missing value code for a 64-bit floating point number</td>
</tr>
<tr>
<td>McGetMissingFlt</td>
<td>mcgetmissingreal</td>
<td>returns the missing value code for a 32-bit floating point number</td>
</tr>
<tr>
<td>McIsMissingDbl</td>
<td>mcismissingdbl</td>
<td>tests whether a 64-bit floating point number contains the missing value code</td>
</tr>
<tr>
<td>McIsMissingFlt</td>
<td>mcismissingreal</td>
<td>tests whether a 32-bit floating point number contains the missing value code</td>
</tr>
</tbody>
</table>
**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 commands, though in a more general way. When run from the command line, error messages generated by the edest or Mcprint function are sufficient. These functions are described in the Text messages section earlier in this chapter.

McIDAS also allows scripts or McIDAS 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.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mccodeset</td>
<td>mccodeset</td>
<td>sets the global status to return upon exiting</td>
</tr>
<tr>
<td>Mccodeget</td>
<td>mcodeget</td>
<td>returns the current value of the global status</td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td>sends an error message and exits</td>
</tr>
<tr>
<td>Mciniterr</td>
<td>not available</td>
<td>returns a string explaining why Mcinit failed</td>
</tr>
</tbody>
</table>

Some older McIDAS 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 abort return a status of 1, the process that started the command won’t know if abort was called, or just return or exit. All new McIDAS applications are coded to call Mccodeset if they encounter a problem when running. Thus, when writing a McIDAS application:

<table>
<thead>
<tr>
<th>Do:</th>
<th>Don’t:</th>
</tr>
</thead>
<tbody>
<tr>
<td>call edest, call mccodeset, and return a status in a function return code</td>
<td>call abort or exit since they are not very informative</td>
</tr>
</tbody>
</table>
Currently, SSEC uses the return codes below for McIDAS applications.

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>command ran successfully</td>
</tr>
<tr>
<td>1</td>
<td>command failed with an unrecoverable error</td>
</tr>
<tr>
<td>2 to 99</td>
<td>command failed with a potentially recoverable error</td>
</tr>
<tr>
<td>100 to 126</td>
<td>reserved for locally developed applications</td>
</tr>
<tr>
<td>127</td>
<td>error of unknown origin</td>
</tr>
</tbody>
</table>

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.

```c
integer ret_val
character*12 option

c--- get command option
    ok = mcmdstr(‘’, 1, ‘’, option)
    if (option ne. ‘LIST’ .and. option ne. ‘COPY’) then
        ret_val = 1
        goto 999
    endif
    ;
    ;

c--- get data
    ok = mOptget (dataset, nsrt, 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 `Mcmoubtfn` 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
  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.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0lock</td>
<td>lock</td>
<td>blocks another application from using a resource</td>
</tr>
<tr>
<td>M0unlock</td>
<td>unlock</td>
<td>releases a resource so another application can use it</td>
</tr>
</tbody>
</table>

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
   format(5A)
c   ... (real processing goes on here)
99   close(21)
c c--- set return code based on whether or not the
c--- name was resolved
   if (valid.eq.0) then
      m0xsresolv=-1
   else
      m0xsresolv=0
   endif

   call unlock(’resolv.srv’)
100  return
end
```
Conversion utilities

The McIDAS 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.

This section describes the functions available in McIDAS 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 software.

Byte- and word-level manipulation

When writing applications for McIDAS, you may find it necessary to manipulate information at the byte level. This section describes the McIDAS 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.

<table>
<thead>
<tr>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alit</td>
<td>copies four bytes of a character string to a single precision floating-point variable</td>
</tr>
<tr>
<td>clit</td>
<td>copies four bytes of an integer value to a 4-character string variable</td>
</tr>
<tr>
<td>crack</td>
<td>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</td>
</tr>
<tr>
<td>dlit</td>
<td>copies eight bytes of a character string to a double precision floating-point variable</td>
</tr>
<tr>
<td>ic</td>
<td>returns the value at a specific byte location (0-based) in a buffer</td>
</tr>
<tr>
<td>lit</td>
<td>copies four bytes of a character string to a 1-word integer variable</td>
</tr>
<tr>
<td>movb</td>
<td>copies a number of bytes from a source buffer to a specified, 0-based offset in a destination buffer</td>
</tr>
<tr>
<td>movblk</td>
<td>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)</td>
</tr>
<tr>
<td>Fortran function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>movc</td>
<td>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)</td>
</tr>
<tr>
<td>movcw</td>
<td>copies the entire contents of a character string buffer to an integer buffer</td>
</tr>
<tr>
<td>movh</td>
<td>copies a number of half-words from a source buffer to a destination buffer with half-word increments</td>
</tr>
<tr>
<td>movpix</td>
<td>copies a number of elements from a source buffer to a destination buffer with incremental offsets for both the source and destination buffers</td>
</tr>
<tr>
<td>movw</td>
<td>copies a number of words from a source buffer to a destination buffer</td>
</tr>
<tr>
<td>movwc</td>
<td>copies the contents of an integer buffer to a character string buffer, copying as many bytes as can be stored in the destination string</td>
</tr>
<tr>
<td>mpixel</td>
<td>copies elements of a specified size in a source buffer to a destination buffer with elements of a specified size</td>
</tr>
<tr>
<td>mpxtb</td>
<td>like mpixel, but additionally converts variable-sized elements in a buffer based on a lookup table</td>
</tr>
<tr>
<td>pack</td>
<td>moves the least significant byte from each element of a word array and compresses it into consecutive bytes in a buffer</td>
</tr>
<tr>
<td>pack2</td>
<td>moves the least significant byte from each element of a half-word array and compresses it into a consecutive byte string</td>
</tr>
<tr>
<td>stc</td>
<td>places a byte value into a specified, 0-based byte location in a buffer</td>
</tr>
<tr>
<td>swbyt2</td>
<td>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</td>
</tr>
<tr>
<td>swbyt4</td>
<td>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</td>
</tr>
<tr>
<td>zeros</td>
<td>sets a number of bytes in an array to zero</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mcinit</td>
<td>converts a value of type double to the nearest integer</td>
</tr>
</tbody>
</table>
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 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
place the result in 'val'
val = ic (oldval, 2)
c--- the resulting value in 'val' will be:
c--- hex
--- 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, 0-based, in 'val'
call stc (src, val, 2)
c--- the resulting value in 'val' will be:
c--- hex
--- 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. Unix machines supported by McIDAS-X are big-endian machines, while PCs running McIDAS-OS2 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
--- src()   input_value input_value output_value
1  abcdef01  01efcdab  abcdef01
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
--- 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
--- src()   input_value input_value output_value
1  abcd     cdab     abcd
2  ef01     01ef     ef01
3  2345     4523     2345
4  6789     8967     6789
```
Copying bytes

The McIDAS 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 move 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 move 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))
    call movb (15, src, dest, 1)
    c---- The resulting value in 'dest' will be:
    c---- dest1   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))
    call movc (14, src, 5, dest, 2)
    c---- The resulting value in 'dest' will be:
    c---- dest1   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)

-- The resulting elements of the array 'dest' will be:

<table>
<thead>
<tr>
<th>dest</th>
<th>hex</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4a756e65</td>
<td>June</td>
</tr>
<tr>
<td>2</td>
<td>20436c65</td>
<td>Cle</td>
</tr>
<tr>
<td>3</td>
<td>61766572</td>
<td>aver</td>
</tr>
</tbody>
</table>
```

**movblk and movpix**

Use the byte-copying functions `movblk` and `movpix` for complicated byte copying operations. These functions let you specify the following:

- the segment sizes to copy
- the number of bytes to skip between source bytes
- the 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
in 'src' into 'dest' beginning at byte 6. The increment
between elements in the source buffer is 2 bytes, the
increment between bytes in the destination buffer is
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
    1    00000000
    2    0000cdef
    3    01000000
    4    00012345
    5    00000000
    6    45678900
    7    00000089
    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'
beginning at byte 6, the increment between elements in
the source buffer is 2 bytes, the increment between bytes
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
    1    00000000
    2    0000cd00
    3    00000000
    4    00010000
    5    00000000
    6    45000000
    7    00000089
    8    00000000
    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 'abcdef01', '23456789', 'fedcba98', '09080706',
     '05040302', '010a0b0c', '0d0e0f11', '22334455',
     '66778899', 'aabcdd', 'eef12113', '14151617',
     '18191a1b', 'icidief', '21232425', '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
value_hex     value_hex
09080706      00010000
c--
05040302      00080009
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, dлит and lit examples
character*4  cvar4
character*8  cvar8
cvar4        = 'FRED'
cvar8        = 'GINGER'
integer      int_dest
double precision  dbl_dest
real         rel_dest
rel_dest     = alit (cvar4)
int_dest     = lit (cvvar4)
```

McIDAS Programmer's Manual

Issued 8/97

McIDAS Utilities

5-83
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.

```c
integer src(2)
integer dest(8)
data src /Z'abc01',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
---  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. 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-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.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsalloc</td>
<td>not available</td>
<td>copies the contents of a Fortran string to a C string, dynamically allocating the memory necessary to store the resulting string</td>
</tr>
<tr>
<td>fslen</td>
<td>not available</td>
<td>returns the logical length of a C string needed to contain a blank-padded Fortran string</td>
</tr>
<tr>
<td>not available</td>
<td>ischar</td>
<td>indicates whether a string of four characters contains all printable characters</td>
</tr>
<tr>
<td>C function</td>
<td>Fortran function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>not available</td>
<td>isdch</td>
<td>determines whether a Fortran string contains all digits, all letters, a mixture of letters and digits, or other characters</td>
</tr>
<tr>
<td>Mclocase</td>
<td>mlocase</td>
<td>converts uppercase characters in a string to lowercase</td>
</tr>
<tr>
<td>Mcstricomp</td>
<td>not available</td>
<td>performs a case-independent comparison of two C character strings</td>
</tr>
<tr>
<td>Mcstrnicmp</td>
<td>not available</td>
<td>performs a case-independent comparison of a specified number of characters in two C strings</td>
</tr>
<tr>
<td>Mcstrtdbl</td>
<td>mcstrtdbl</td>
<td>converts a character string in decimal point format to a double-precision value</td>
</tr>
<tr>
<td>Mcstrtdhr</td>
<td>mcstrtdhr</td>
<td>converts a character string in time format to a double-precision value</td>
</tr>
<tr>
<td>Mcstrtdll</td>
<td>mcstrtdll</td>
<td>converts a character string in lat/lon format to a double-precision value</td>
</tr>
<tr>
<td>Mcstrtohex</td>
<td>mcstrtohex</td>
<td>converts a character string in hexadecimal to an integer value</td>
</tr>
<tr>
<td>Mcstrtohms</td>
<td>mcstrtohms</td>
<td>converts a character string in time format to its components: hours, minutes, seconds</td>
</tr>
<tr>
<td>Mcstrtoihr</td>
<td>mcstrtoihr</td>
<td>converts a character string in time format to an integer value in units of hours/minutes/seconds</td>
</tr>
<tr>
<td>Mcstrtoill</td>
<td>mcstrtoill</td>
<td>converts a character string in lat/lon format to an integer value in units of degrees/minutes/seconds</td>
</tr>
<tr>
<td>Mcstrtoint</td>
<td>mcstrtoint</td>
<td>converts a character string to an integer value</td>
</tr>
<tr>
<td>Mcstrtoiyd</td>
<td>mcstrtoiyd</td>
<td>converts a character string representing a day to an integer representation of Julian day</td>
</tr>
<tr>
<td>Mcstrtofs</td>
<td>not available</td>
<td>copies a C character string to a Fortran character string</td>
</tr>
<tr>
<td>Mcupcase</td>
<td>mcupcase</td>
<td>converts lowercase characters in a string to uppercase</td>
</tr>
<tr>
<td>not available</td>
<td>nchars</td>
<td>indicates the starting and ending character positions (1-based) and length of a Fortran string</td>
</tr>
<tr>
<td>stralloc</td>
<td>not available</td>
<td>concatenates a variable number of C character strings into a single string pointer</td>
</tr>
</tbody>
</table>

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 software.
Converting string formats to integer values

The *Mctrtoint* 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 command line.

To convert a hexadecimal string with *Mctrtoint*, prefix the string with a dollar sign ($). When converting exponential representations with *Mctrtoint*, 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.

```c
{ char src_string[20];
    int val;
    int status;

    /* -- convert the character string '4321' to an integer */
    (void) strcpy (src_string, "4321");
    status = Mctrtoint (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 = Mctrtoint (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 = Mctrtoint (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 = Mctrtoint (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 = Mctrtoint (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 `Mcrestodbl` 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 command line. The sample code below shows how to use `Mcrestodbl`.

```c
{
    char src_string[20];
    double val;
    int status;

    /*
     * -- convert the character string '-4321.1' to a double
     */
    (void) strcpy (src_string, "-4321.1");
    status = Mcrestodbl (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 = Mcrestodbl (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 time and latitude/longitude formats to double-precision and integer values.

- **Mcrestodhr** returns time values in units of hours as a double-precision value.

- **Mcrestoihr** returns an integer representation of time in the format `hhmmss`.

- **Mcrestodll** returns latitude or longitude in double-precision values.

- **Mcrestoiil** returns a scaled integer representation of latitude and longitude in the integer format `ddmmss`.

To get a list of the time formats, enter the command ARGHELP TIME on the McIDAS command line. To get a list of the latitude/longitude formats, enter the command ARGHELP LATLON.
If your input string for Mestrtodhr or Mestrtoihr 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 library function Mcgettime.

The calling sequence and valid string formats for Mestrtodll and Mestrtoll are identical to Mestrtodhr and Mestrtoihr, except the former will not return the current system time if the NULL string or colon string are passed as input.

```c
{
  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 = Mestrtodhr (src_string, &dval);
  status = Mestrtoihr (src_string, &val);

  /*
   * upon exit, 'dval' will contain 421.321 and 'val' will contain
   * 4210736
   */
  /*
   * -- convert the string '21:32:12' to a time
   */
  (void) strcpy (src_string, '21:32:12');
  status = Mestrtodhr (src_string, &dval);
  status = Mestrtoihr (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 = Mestrtodhr (src_string, &dval);
  status = Mestrtoihr (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 **Mestrtohms** 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 command line.

The sample code below shows how to use **Mestrtohms**.

```c
{
    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 = Mestrtohms (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 **Mestrtoiyd** 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 library function **Mcgetday**.
To list the valid date formats, enter the command ARGHELP DATE on the McIDAS command line.

The sample code below shows the use of the `Mestrtoiyd` function.

```c
{  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 = Mestrtoiyd (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 = Mestrtoiyd (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 = Mestrtoiyd (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

`Mostrtofs` 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.

```c
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 = Mostrtofs (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.

```c
character*24 fullname
data fullname/'Joe Dimaggio'/
call printf (fullname)
;

void printf__ (char *fname, FLEN 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 `isdigch` 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.

```c
character*4 cvar4
caller*12 cvar12
integer status

cvar4 = 'AbCd'
status = ischar (cvar4)

---
---
don return, 'status' will contain the value 1 indicating
don that all 4 characters are printable.

cvar12 = '3214as231G'
status = isdigch (cvar12)

---
---
don return, 'status' will contain the value 0 indicating
don 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

Mestricmp performs a case-insensitive string comparison between two entire strings. Mestriicmp 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 = Mestricmp (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 = Mestriicmp (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.

```c
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)
{
    Mprintf ("%s
", string1);
    free (string1);
}

string1 = stralloc ("CO=", "US", ", " , "MK", ", " , "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 MK UK"
  */

if (string1 != (char *) NULL)
{
    Mprintf ("%s
", string1);
    free (string1);
}

return;
```
Converting lowercase and uppercase characters

The `Mlocase` 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.

```c
{ char name[20];
  (void) strcpy (name, "Hank Aaron!!");
  /* convert the contents of the character array ‘name’
   * to lowercase
   */
  Mlocase (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

All McIDAS applications use the Julian day format. To correctly represent both the twentieth and twenty-first centuries, the McIDAS library provides Julian day manipulation routines using the format \texttt{ccyymdd} where \texttt{cc} represents the century, \texttt{yy} is the year of the century and \texttt{mdd} is the day of the year, with January 1 being day one. For example, 17 January 1997 is represented as 1997017.

Some of the functions described in this section still expect the older Julian day format, \texttt{yyymdd}. However, they will be replaced with the new format.

The time utilities described in this section use the integer representation \texttt{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 library, along with a short description.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mcyymdd</td>
<td>mccyymdd</td>
<td>verifies that a Julian day in the form \texttt{ccyymdd} is correct</td>
</tr>
<tr>
<td>Mccyymdd</td>
<td>mccyymdd</td>
<td>converts a Julian day in the form \texttt{ccyymdd} to day, month and year</td>
</tr>
<tr>
<td>Mccyymrostr</td>
<td>mccyymrostr</td>
<td>converts the Julian day in the form \texttt{ccyymdd} to a variety of character string formats</td>
</tr>
<tr>
<td>Mccyymrotoyd</td>
<td>mccyymrotoyd</td>
<td>converts a 7-digit Julian day to 5 digits</td>
</tr>
<tr>
<td>Mccyymrotoymetosec</td>
<td>mccyymrotoymetosec</td>
<td>converts a Julian day in the form \texttt{ccyymdd} and the time of day in the form \texttt{hhmmss} to seconds since 1 January 1970 at 00 UTC</td>
</tr>
<tr>
<td>Mccyymrotohr</td>
<td>mccyymrotohr</td>
<td>converts hours stored in double precision to hours stored in an integer of the form \texttt{hhmmss}</td>
</tr>
<tr>
<td>Mccyymrotoycyd</td>
<td>mccyymrotoycyd</td>
<td>converts day, month and year to a Julian day in the form \texttt{ccyymdd}</td>
</tr>
<tr>
<td>Mccyymrotoymgetday</td>
<td>mccyymrotoymgetday</td>
<td>gets the current system Julian day in the form \texttt{ccyymdd}</td>
</tr>
<tr>
<td>Mccyymrotoymgetdaytime</td>
<td>mccyymrotoymgetdaytime</td>
<td>gets the current Julian day in the form \texttt{ccyymdd} and the current time of day in the form \texttt{hhmmss}</td>
</tr>
<tr>
<td>C function</td>
<td>Fortran function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Mcgettime</td>
<td>mcgettime</td>
<td>gets the current time of day in the form hhmmss</td>
</tr>
<tr>
<td>Mchrmsok</td>
<td>mchrmsok</td>
<td>verifies that a time value in the form hhmmss is correct</td>
</tr>
<tr>
<td>Mchrmsiohr</td>
<td>mchrmsiohr</td>
<td>converts hours, minutes and seconds to hhmmss format</td>
</tr>
<tr>
<td>Mcincday</td>
<td>mcincday</td>
<td>increments/decrements a Julian day value</td>
</tr>
<tr>
<td>Mcinctime</td>
<td>mcinctime</td>
<td>increments/decrements a Julian day and time value</td>
</tr>
<tr>
<td>MCsectodaytime</td>
<td>MCsectodaytime</td>
<td>converts seconds since 1 January 1970 to a Julian day in the form ccyyddd and a time in the form hhmmss</td>
</tr>
<tr>
<td>Mcydtocyd</td>
<td>mcydtocyd</td>
<td>converts a 5-digit Julian day to 7 digits</td>
</tr>
</tbody>
</table>

These functions are further defined below along with examples of sample code. The more commonly used functions are described first.

For more information about these day and time functions, see the online man pages provided with the McIDAS software.
Retrieving the current system day and time

The McIDAS 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.

```c
{
    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 `Mcydtoyd` and `Mcydtoeyd` functions to transition your code from the previous 5-digit Julian day format, `yyddd`, to the new 7-digit format, `ccyddd`, and vice versa.

- `Mcydtoyd` converts the 7-digit Julian day format to the 5-digit representation.
- `Mcydtoeyd` converts the 5-digit Julian day format to the 7-digit version.

The `Mcydtoeyd` 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.

```c
{
    int status;
    int new_day;

    /* convert the day 1996017 to the 5 digit form of the Julian day */
    status = Mcydtoyd (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 = Mcydtoeyd (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 = Mcydtoeyd (11100, &new_day);
    /* if successful, the contents of new_day will be 2011100 */
}
```
Converting a Julian day and time to seconds, and vice versa

`Mcdaytimetosec` and `Mcsectodatime` 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 functions `sksees` and `skhms`, which use 1 January 1972 as the base.

```c
#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 = Mcsectodatime (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

огда converts a Julian day to day, month and year, including the century. The month number returned is 1-based, meaning January is 1. Tody converts a day, month, year combination to a Julian day. Examples of both functions are shown in the code fragment below.

```c
{
    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 = Modmytocyd(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 = Modmytocyd(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 = Mccydtdomy(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

Mcctodtstr converts a Julian day to a variety of character strings representing the date, as shown in the code sample below.

```c
{  int status;
    char *day_string;
    /*
    * convert the Julian day 1996017 to a character string of the
    * form nn mmm, cyy
    */
    status = Mcctodtstr (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
    */
}
```

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 Mcinctday 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.

```c
{  int new_day;
    int new_time;
    int status;
    /* increment the Julian day 1996364 by 3 days */
    status = Mcinctday (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 library contains two utilities for converting latitude and longitude values:

- **flalo** converts an integer representation of latitude or longitude in the format *dddmnss* to a single-precision representation, in degrees.

- **ilalo** converts a single-precision latitude or longitude to a scaled-integer value in the format *dddmnss*.

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
     floating point value

lat = 593000
flat = flalo (lat)

c---- upon successful completion, 'flat' contains the value 59.5

     convert the latitude value 45.75 to a scaled integer value

flat = 45.75
lat = ilalo (flat)

     upon successful completion, 'lat' contains the value 454500
```

For more information about these conversion functions, see the online man pages provided with the McIDAS software.
**Physical units**

The McIDAS library contains two utilities for converting physical units:

- **mcuvtld** 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 an integral part of any user application because data is often stored in units that a user won’t typically display. The **mcuvtld** and **mcucvtr** functions can convert the physical units shown in the table below; all McIDAS core datasets use this standard.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Valid units</th>
<th>Interface representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>meters</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>kilometers</td>
<td>KM</td>
</tr>
<tr>
<td></td>
<td>decameters</td>
<td>DM</td>
</tr>
<tr>
<td></td>
<td>centimeters</td>
<td>CM</td>
</tr>
<tr>
<td></td>
<td>millimeters</td>
<td>MM</td>
</tr>
<tr>
<td></td>
<td>miles</td>
<td>MI</td>
</tr>
<tr>
<td></td>
<td>nautical miles</td>
<td>NMI</td>
</tr>
<tr>
<td></td>
<td>yards</td>
<td>YD</td>
</tr>
<tr>
<td></td>
<td>feet</td>
<td>FT</td>
</tr>
<tr>
<td></td>
<td>inches</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>degrees of latitude</td>
<td>DEGL</td>
</tr>
<tr>
<td>speed</td>
<td>miles per hour</td>
<td>MPH</td>
</tr>
<tr>
<td></td>
<td>knots</td>
<td>KT or KTS</td>
</tr>
<tr>
<td></td>
<td>meters per second</td>
<td>MPS</td>
</tr>
<tr>
<td></td>
<td>feet per second</td>
<td>FPS</td>
</tr>
<tr>
<td></td>
<td>kilometers per hour</td>
<td>KPH</td>
</tr>
<tr>
<td>temperature</td>
<td>Kelvin</td>
<td>K</td>
</tr>
<tr>
<td></td>
<td>Fahrenheit</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Celsius</td>
<td>C</td>
</tr>
<tr>
<td>pressure</td>
<td>millibars</td>
<td>MB</td>
</tr>
<tr>
<td></td>
<td>inches of Mercury</td>
<td>INHG</td>
</tr>
<tr>
<td></td>
<td>pascals</td>
<td>PA</td>
</tr>
<tr>
<td></td>
<td>hectopascals</td>
<td>HPA</td>
</tr>
<tr>
<td>time</td>
<td>hours</td>
<td>HR</td>
</tr>
<tr>
<td></td>
<td>minutes</td>
<td>MIN</td>
</tr>
<tr>
<td></td>
<td>seconds</td>
<td>SEC</td>
</tr>
<tr>
<td></td>
<td>days</td>
<td>DAY</td>
</tr>
<tr>
<td></td>
<td>years</td>
<td>YR</td>
</tr>
<tr>
<td>Attribute</td>
<td>Valid units</td>
<td>Interface representation</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>weight</td>
<td>grams</td>
<td>G</td>
</tr>
<tr>
<td></td>
<td>kilograms</td>
<td>KG</td>
</tr>
<tr>
<td></td>
<td>pounds</td>
<td>LB</td>
</tr>
<tr>
<td></td>
<td>ounces</td>
<td>OZ</td>
</tr>
<tr>
<td></td>
<td>tons</td>
<td>TON</td>
</tr>
</tbody>
</table>

The code segment below demonstrates these unit conversion routines.

```c
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 = mcvrgd (numval, inunit, 50.0, 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 = mcvrgd (numval, inunit, 50.0, 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 software.
Scientific utilities

The McIDAS 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.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>not available</td>
<td>lab</td>
<td>computes potential and equivalent potential temperature and mixing ratio, given the temperature, dew point, pressure, and station elevation</td>
</tr>
<tr>
<td>McBeta</td>
<td>mcbeta</td>
<td>compute beta ((\frac{\partial}{\partial y})) parameter</td>
</tr>
<tr>
<td>McCoriolis</td>
<td>mccorfor</td>
<td>computes the coriolis parameter ((f = 2\Omega))</td>
</tr>
<tr>
<td>McDirec</td>
<td>mcdirec</td>
<td>computes meteorological direction of wind for u- and v-components</td>
</tr>
<tr>
<td>McHypsoP</td>
<td>mchypsop</td>
<td>integrates hyposometric equation to compute a pressure at a given height</td>
</tr>
<tr>
<td>McHypsoZ</td>
<td>mchypsoz</td>
<td>integrates hyposometric equation to compute a new height at a given pressure</td>
</tr>
<tr>
<td>McHeatIndex</td>
<td>mcheatindex</td>
<td>computes the heat index, given the temperature and dew point</td>
</tr>
<tr>
<td>McLatentHeat</td>
<td>mclatvap</td>
<td>latent heat of vaporization</td>
</tr>
<tr>
<td>McLiftClLevel</td>
<td>mclcl</td>
<td>parcel temperature and pressure at LCL</td>
</tr>
<tr>
<td>McMixing</td>
<td>mcrmix</td>
<td>mixing ratio</td>
</tr>
<tr>
<td>McPFromTheta</td>
<td>mcpffromtheta</td>
<td>returns pressure when given theta and temperature</td>
</tr>
<tr>
<td>McRelativeHumidity</td>
<td>mcrelativehumidity</td>
<td>computes the relative humidity, given the temperature and dewpoint</td>
</tr>
<tr>
<td>C function</td>
<td>Fortran function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------</td>
<td>----------------------------------------------------</td>
</tr>
<tr>
<td>McSatVapor</td>
<td>mcsatvap</td>
<td>saturation vapor pressure over water</td>
</tr>
<tr>
<td>McSatVaporIce</td>
<td>mcsatvapi</td>
<td>saturation vapor pressure over ice</td>
</tr>
<tr>
<td>McSpeed</td>
<td>mcspeed</td>
<td>computes wind speed from u- and v-components</td>
</tr>
<tr>
<td>McStationPres</td>
<td>mcstnpres</td>
<td>station pressure, given altimeter and station elevation</td>
</tr>
<tr>
<td>McTDFromMixing</td>
<td>mctamr</td>
<td>computes dew point from mixing ratio</td>
</tr>
<tr>
<td>McTempAtThetae</td>
<td>mctasa</td>
<td>temperature along saturated adiabats</td>
</tr>
<tr>
<td>McTFromTheta</td>
<td>mctfromtheta</td>
<td>returns temperature when given theta and pressure</td>
</tr>
<tr>
<td>McTheta</td>
<td>mcttheta</td>
<td>potential temperature</td>
</tr>
<tr>
<td>McThetae</td>
<td>mctthetae</td>
<td>equivalent potential temperature</td>
</tr>
<tr>
<td>McThetaw</td>
<td>mcthetaw</td>
<td>wet bulb potential temperature</td>
</tr>
<tr>
<td>McUandV</td>
<td>mcuandv</td>
<td>computes wind u- and v-components</td>
</tr>
<tr>
<td>McWetBulb</td>
<td>mcwetbulb</td>
<td>wet bulb temperature</td>
</tr>
<tr>
<td>McWindChill</td>
<td>mcwindchill</td>
<td>computes the wind chill, given the temperature and wind speed</td>
</tr>
<tr>
<td>McVirTemp</td>
<td>mcvrittemp</td>
<td>virtual temperature</td>
</tr>
<tr>
<td>not available</td>
<td>rmix</td>
<td>determines the mixing ratio, given the temperature and pressure</td>
</tr>
<tr>
<td>not available</td>
<td>sndanl</td>
<td>computes various stability parameters, given a sounding</td>
</tr>
</tbody>
</table>

For more information about these scientific functions, see the online man pages provided with the McIDAS 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.

```plaintext
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 = 292.0d0
dewpt= 289.0d0
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`.

```plaintext
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 be 32.5 degrees Celsius
```

Computing Relative Humidity

Given the temperature and dew point, the functions `McRelativeHumidity` and `mrelativehumidity` will compute the relative humidity. The value returned will be given in the dimensionless units of percentage. Below is an example of the function `mrelativehumidity`.

```plaintext
integer ok
double precision temperature
double precision dewpoint
double precision rh

temperature = 30.d0
dewpoint = 20.d0

ok = mrelativehumidity (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 adest('error calculating the wind chill', 0)
    return
endif
---upon successful completion the value of windchill will
---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, $w_v$, 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 **sdanl** function.

```c
parameter (NLEV = 9)
real press(NLEV)
real temp(NLEV)
real dewpt (NLEV)
real dir(NLEV)
real spd (NLEV)
real stabl(12)

C--- get sounding data

: 

C--- assume the arrays: press, temp, dewpt, dir and spd
C--- have been initialized with the following values

<table>
<thead>
<tr>
<th>press(mb)</th>
<th>temp (K)</th>
<th>dewpt (K)</th>
<th>dir (DEG)</th>
<th>spd (MPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>988.0</td>
<td>300.0</td>
<td>299.8</td>
<td>250</td>
<td>1.5</td>
</tr>
<tr>
<td>925.0</td>
<td>301.0</td>
<td>280.0</td>
<td>270</td>
<td>3.8</td>
</tr>
<tr>
<td>850.0</td>
<td>297.7</td>
<td>269.0</td>
<td>285</td>
<td>5.6</td>
</tr>
<tr>
<td>700.0</td>
<td>277.5</td>
<td>276.1</td>
<td>290</td>
<td>12.3</td>
</tr>
<tr>
<td>500.0</td>
<td>262.1</td>
<td>252.1</td>
<td>265</td>
<td>13.3</td>
</tr>
<tr>
<td>400.0</td>
<td>253.6</td>
<td>240.5</td>
<td>280</td>
<td>29.8</td>
</tr>
<tr>
<td>300.0</td>
<td>237.6</td>
<td>226.0</td>
<td>280</td>
<td>33.4</td>
</tr>
<tr>
<td>250.0</td>
<td>229.7</td>
<td>215.7</td>
<td>285</td>
<td>41.6</td>
</tr>
<tr>
<td>200.0</td>
<td>216.3</td>
<td>206.3</td>
<td>285</td>
<td>45.7</td>
</tr>
</tbody>
</table>

C--- perform the sounding analysis for 002

    call sdanl (0, NLEV, press, temp, dewpt, dir, spd, stabl)

C--- upon return, the values stored in stabl will be as follows:

<table>
<thead>
<tr>
<th>stabl(1)</th>
<th>282.4</th>
<th>parcel dewpoint (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>stabl(2)</td>
<td>301.1</td>
<td>potential temperature (k)</td>
</tr>
<tr>
<td>stabl(3)</td>
<td>324.1</td>
<td>equivalent potential temperature (k)</td>
</tr>
<tr>
<td>stabl(4)</td>
<td>8.3</td>
<td>mixing ratio (g/kg)</td>
</tr>
<tr>
<td>stabl(5)</td>
<td>23.7</td>
<td>precipitable water (mm)</td>
</tr>
<tr>
<td>stabl(6)</td>
<td>32.7</td>
<td>convective temperature (c)</td>
</tr>
<tr>
<td>stabl(7)</td>
<td>26.9</td>
<td>forecast maximum temperature (c)</td>
</tr>
<tr>
<td>stabl(8)</td>
<td>2.7</td>
<td>lifted index (k)</td>
</tr>
<tr>
<td>stabl(9)</td>
<td>33.4</td>
<td>total totalis</td>
</tr>
<tr>
<td>stabl(10)</td>
<td>445.5</td>
<td>equivalent pressure (mb)</td>
</tr>
<tr>
<td>stabl(11)</td>
<td>20.6</td>
<td>k index</td>
</tr>
<tr>
<td>stabl(12)</td>
<td>47.6</td>
<td>sweat index</td>
</tr>
</tbody>
</table>
```
6

Accessing Data

This chapter provides the information you will need to access and use McIDAS data. You'll learn:

- the attributes unique to each of the McIDAS data types: disk files, images, grids, point observations and text

- the API functions available for reading, writing and deleting McIDAS 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 McIDAS' ability to integrate and display a variety of datasets is made possible by its navigation and calibration subsystems

Contents

<table>
<thead>
<tr>
<th>McIDAS disk files</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic concepts</td>
<td>6-3</td>
</tr>
<tr>
<td>Disk file APIs</td>
<td>6-5</td>
</tr>
<tr>
<td>Reading and writing disk files</td>
<td>6-6</td>
</tr>
<tr>
<td>Assigning a system pathname to a disk file</td>
<td>6-7</td>
</tr>
<tr>
<td>Determining if a disk file exists</td>
<td>6-8</td>
</tr>
<tr>
<td>Deleting the contents of a disk file</td>
<td>6-8</td>
</tr>
<tr>
<td>Deleting a disk file</td>
<td>6-9</td>
</tr>
<tr>
<td>Locking and unlocking a disk file</td>
<td>6-10</td>
</tr>
<tr>
<td>Copying a disk file</td>
<td>6-11</td>
</tr>
</tbody>
</table>
Contents

Image data ................................................. 6-12
  Basic concepts ........................................... 6-12
  What is an image object? ............................... 6-14
  Reading image objects .................................. 6-15
  Writing image objects to a dataset .................... 6-29
  Deleting image objects ................................. 6-32

Grid data ..................................................... 6-33
  Basic concepts .......................................... 6-34
  What is a grid object? ................................... 6-36
  Reading grid objects ................................... 6-37

Point data ..................................................... 6-48
  Basic concepts .......................................... 6-49
  What is a point object? .................................. 6-51
  Reading point objects ................................... 6-52
  Reading the point-data file header ..................... 6-58

Text data ...................................................... 6-61
  Flat-file text ............................................ 6-61
  General weather text .................................... 6-63

McIDAS navigation .......................................... 6-67
  Basic concepts .......................................... 6-68
  Using the navigation APIs .............................. 6-71
  Implementing an image navigation module ............... 6-76
  Example navigation module code ....................... 6-85

McIDAS calibration ......................................... 6-96
  Basic concepts .......................................... 6-97
  Designing your calibration module ..................... 6-98
  Writing your calibration module ....................... 6-103
**McIDAS disk files**

Reading and writing data to and from disk is fundamental to many applications programs. A McIDAS disk file stores information that applications can randomly access by byte address using standard system library calls.

This section describes:

- the special attributes of McIDAS disk files that distinguish them from other system files
- the API functions you will use to read, write, delete, and copy disk files

---

**Basic concepts**

McIDAS 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 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 command ends, all writable, zero-length files accessed while the program was running are deleted.

♦ **Disk file names are limited to 12 characters.** This restriction is imposed by the limitations of the OS/2 FAT file system, which mandates that file names contain eight characters in the name, one dot, and three characters in the extension. OS/2’s HPFS and Unix do not have this limitation. Although the McIDAS disk file I/O subsystem has no inherent name length limitation, you must abide by the 12-character limit for code that will work on all platforms.

♦ **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 file system.

If the file name passed from the application to the McIDAS disk file API functions does not contain a slash character, the McIDAS 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 library functions that you will use when programming with disk files. Previously in McIDAS, a disk file was called an LW (Large Word) array file. You will notice that many of the Fortran APIs below begin with the letters lw.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mcpathname</td>
<td>volnam</td>
<td>converts a disk file name to a fully qualified pathname/filename</td>
</tr>
<tr>
<td>Mcread</td>
<td>lbi</td>
<td>reads bytes from a disk file into memory</td>
</tr>
<tr>
<td></td>
<td>lwi</td>
<td>reads words (4-byte groups) from a disk file</td>
</tr>
<tr>
<td>Mcwrite</td>
<td>lbo</td>
<td>creates a disk file by writing bytes into it from memory</td>
</tr>
<tr>
<td>not available</td>
<td>lwo</td>
<td>creates a disk file by writing words (4-byte groups) into it</td>
</tr>
<tr>
<td>Mcremove</td>
<td>lwd</td>
<td>deletes a disk file</td>
</tr>
<tr>
<td>Mctruncate</td>
<td>lwtrunc</td>
<td>deletes the contents of a disk file without deleting the file itself</td>
</tr>
<tr>
<td>not available</td>
<td>lwfile</td>
<td>determines if a file exists</td>
</tr>
<tr>
<td>not available</td>
<td>lock</td>
<td>acquires an exclusive lock on a file</td>
</tr>
<tr>
<td>not available</td>
<td>unlock</td>
<td>frees the lock on a file</td>
</tr>
</tbody>
</table>

Each function is described in the sections below, along with sample code illustrating its use.
Reading and writing disk files

The McIDAS 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

--- initialize
first = 0
nwords = 3
do 200 i = 1, nwords
200 array_out(i) = i * 100

--- write the data to disk -- note that there are four bytes
--- 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

--- at this point, the file 'testdata' will consist of:
--- word 0 = integer value 100
--- word 1 = integer value 200
--- word 2 = integer value 300
--- words 3 and beyond are unwritten

--- now, read the data in, skipping the first word:
first = 1
status = lbi('testdata', first*4, nwords*4, array_in)

--- at this point, the array array_in will consist of:
--- word 1 = integer value 200
--- word 2 = integer value 300
--- word 3 = 0x00000000 (the missing value indicating
--- 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 Mpathname function to convert the name of the disk file into a fully qualified, system pathname/file. 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 command that looks like this:
--- REDIRECT ADD MYDATA "'/home/me/mcidas/data"
---
#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 mcccodeset(1)
return
endif
--- At this point, fullname will contain the fully-qualified 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 \texttt{lwtrunc} function to delete the contents of a file.

\begin{verbatim}
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
\end{verbatim}

\section*{Deleting a disk file}

Use the \texttt{lwd} or \texttt{Mcremove} functions to delete a disk file, as shown in the code fragment below.

\begin{verbatim}
... 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 mcodeexit(2)
        goto 999
    else
        goto 999
    endif

C --- do some other processing
\end{verbatim}
Locking and unlocking a disk file

As long as a user has read and write permissions, the McIDAS 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 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')
    rc = lbi('myfile', 0,100,array)
...  
c --- update the values in array
...  
c --- now save the updated info back into 'myfile'
c    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
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 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:

- the attributes that distinguish image data from other types of data
- the blocks of information contained in an image
- the 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 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 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-5 data is 1-byte per data point while GOES-9 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 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 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 library functions for performing these tasks.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcacal</td>
<td>reads the calibration block of an image object</td>
</tr>
<tr>
<td>mcacrd</td>
<td>reads the comment block of an image object</td>
</tr>
<tr>
<td>mcadir</td>
<td>opens a connection to read the directory block from an image object</td>
</tr>
<tr>
<td>mcadrd</td>
<td>reads the directory block from an image object</td>
</tr>
<tr>
<td>mcafree</td>
<td>frees the handle and memory of a connection opened by mcaget</td>
</tr>
<tr>
<td>mcaget</td>
<td>opens a connection to read the data block from an image object</td>
</tr>
<tr>
<td>mcalin</td>
<td>reads the data portion of the current image line</td>
</tr>
<tr>
<td>mcanav</td>
<td>reads the navigation block of an image object</td>
</tr>
<tr>
<td>mcapfx</td>
<td>reads the prefix portion of the current image line</td>
</tr>
<tr>
<td>mcasort</td>
<td>gets the parameters from the command line and adds them to the selection array for a future mcaget call</td>
</tr>
<tr>
<td>mcpcal</td>
<td>parses a list of valid calibration types from the comment cards</td>
</tr>
<tr>
<td>mcpcnav</td>
<td>parses out geographic resolution information from the comment cards</td>
</tr>
</tbody>
</table>

These functions are described below along with sample code.

See the online man pages provided with the McIDAS 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 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.

<table>
<thead>
<tr>
<th>Selection clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUX YES or AUX NO</td>
<td>appends center latitude/longitude, resolution, and calibration types to the comment block (default = YES)</td>
</tr>
<tr>
<td>DAY bday eday</td>
<td>image day range</td>
</tr>
<tr>
<td>SS ss1 ss2</td>
<td>SSEC sensor source number range, from 1 to 99</td>
</tr>
<tr>
<td>SUBSET bpos epos</td>
<td>position range or SUBSET ALL</td>
</tr>
<tr>
<td>TIME btime etime</td>
<td>image time range</td>
</tr>
</tbody>
</table>

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 `mcpsav` and `mcpeal` functions, as shown in the code fragment on the next page.
Reading the directory block

Once the connection is opened with \texttt{mcadir}, the application makes repeated calls to \texttt{mcadrd} until all image object directory blocks are retrieved. The \texttt{mcadir} call must precede the call to \texttt{mcadrd}, as shown below.

```c
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 = mcpocal (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.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>relative position of the image object in the ADDE dataset</td>
</tr>
<tr>
<td>2</td>
<td>version number; currently = 4</td>
</tr>
<tr>
<td>3</td>
<td>SSEC sensor source number; see the Appendices</td>
</tr>
<tr>
<td>4</td>
<td>nominal year and Julian day of the image, yyddd</td>
</tr>
<tr>
<td>5</td>
<td>nominal time of the image, hhmmss</td>
</tr>
<tr>
<td>6</td>
<td>upper-left image line coordinate</td>
</tr>
<tr>
<td>7</td>
<td>upper-left image element coordinate</td>
</tr>
<tr>
<td>8</td>
<td>reserved</td>
</tr>
<tr>
<td>9</td>
<td>number of lines in the image</td>
</tr>
<tr>
<td>10</td>
<td>number of data points per line</td>
</tr>
<tr>
<td>11</td>
<td>number of bytes per data point</td>
</tr>
<tr>
<td>12</td>
<td>line resolution</td>
</tr>
<tr>
<td>13</td>
<td>element resolution</td>
</tr>
<tr>
<td>14</td>
<td>number of spectral bands</td>
</tr>
<tr>
<td>15</td>
<td>length of the line prefix</td>
</tr>
<tr>
<td>16</td>
<td>SSEC project number used when creating the file</td>
</tr>
<tr>
<td>17</td>
<td>year and Julian day the file was created, yyddd</td>
</tr>
<tr>
<td>18</td>
<td>time the file was created, hhmmss</td>
</tr>
<tr>
<td>19</td>
<td>spectral band map</td>
</tr>
<tr>
<td>20</td>
<td>image ID number</td>
</tr>
<tr>
<td>21 - 24</td>
<td>reserved for sensor-specific data</td>
</tr>
<tr>
<td>25 - 32</td>
<td>memo field; 32 ASCII characters</td>
</tr>
<tr>
<td>33</td>
<td>reserved</td>
</tr>
<tr>
<td>34</td>
<td>byte offset to the start of the data block</td>
</tr>
<tr>
<td>35</td>
<td>byte offset to the start of the navigation block</td>
</tr>
<tr>
<td>36</td>
<td>validity code</td>
</tr>
<tr>
<td>37 - 44</td>
<td>PDL (Program Data Load); used for pre-GOES-8 satellites</td>
</tr>
<tr>
<td>45</td>
<td>source of band 8; used for GOES AA processing</td>
</tr>
<tr>
<td>46</td>
<td>actual image start year and Julian day, yyddd</td>
</tr>
<tr>
<td>47</td>
<td>actual image start time, hhmmss; in milliseconds for POES data</td>
</tr>
<tr>
<td>48</td>
<td>actual image start scan</td>
</tr>
<tr>
<td>49</td>
<td>length of the prefix documentation</td>
</tr>
<tr>
<td>50</td>
<td>length of the prefix calibration</td>
</tr>
<tr>
<td>Entry</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>51</td>
<td>length of the prefix band list</td>
</tr>
<tr>
<td>52</td>
<td>source type; satellite specific (ASCII)</td>
</tr>
<tr>
<td>53</td>
<td>calibration type; satellite specific (ASCII)</td>
</tr>
<tr>
<td>54 - 59</td>
<td>reserved</td>
</tr>
<tr>
<td>60</td>
<td>byte offset to the supplemental block</td>
</tr>
<tr>
<td>61</td>
<td>number of bytes in the supplemental block</td>
</tr>
<tr>
<td>62</td>
<td>reserved</td>
</tr>
<tr>
<td>63</td>
<td>byte offset to the start of the calibration block</td>
</tr>
<tr>
<td>64</td>
<td>number of comment cards</td>
</tr>
</tbody>
</table>

For more information about the directory block, see the AREAmn data structure in Chapter 7, 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 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.

<table>
<thead>
<tr>
<th>Selection clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUX YES or AUX NO</td>
<td>inserts the unit and scale factor of the image data into entries 58 and 59 of the directory block</td>
</tr>
<tr>
<td>BAND band</td>
<td>spectral band, if the image has multiple bands</td>
</tr>
<tr>
<td>CAL QTIR</td>
<td>quick calibration switch for POES images</td>
</tr>
<tr>
<td>DAY bday</td>
<td>image Julian day (no default)</td>
</tr>
<tr>
<td>DOC YES or DOC NO</td>
<td>includes the documentation from the line prefix (default = NO)</td>
</tr>
<tr>
<td>LOCATE cor ycor xcor</td>
<td>sets the coordinate type and the coordinate positions relative to the coordinate type (default = AU 0 0)</td>
</tr>
<tr>
<td>MAG imag emag</td>
<td>line and element magnification factor (default = 1 1)</td>
</tr>
<tr>
<td>POS pos</td>
<td>absolute position in the dataset</td>
</tr>
<tr>
<td>SIZE lines elems</td>
<td>number of image lines and data elements (default = 480 x 640)</td>
</tr>
<tr>
<td>SU name</td>
<td>stretching table name (default = no stretch)</td>
</tr>
<tr>
<td>TIME btime etime</td>
<td>image time range (no default)</td>
</tr>
</tbody>
</table>

**AUX YES**—use this clause to insert the unit and scale factor of the image data into entries 58 and 59 of the directory block.

**BAND**—use this clause to identify a spectral band of image data. If you don't specify BAND, the server determines a data-dependent band to use. Specify BAND ALL to return all spectral bands in the source image.

**CAL**—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 (S=satellite, A=array, 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 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 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.
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 11 (1 byte per data point), 12 (2 bytes per data point) and 14 (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.

<table>
<thead>
<tr>
<th>Command line keyword</th>
<th>Equivalent mcaget selection clause</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUX YES</td>
<td>AUX YES</td>
<td>always set by mcasort</td>
</tr>
<tr>
<td>BAND = band</td>
<td>BAND band</td>
<td>only one spectral band in the clause</td>
</tr>
<tr>
<td>DAY = bday</td>
<td>DAY bday</td>
<td></td>
</tr>
<tr>
<td>LATLON = lat lon</td>
<td>LOCATE Eloc lat lon</td>
<td>loc defaults to center (EC) if keyword PLACE is not specified</td>
</tr>
<tr>
<td>LINELE = line ele sys</td>
<td>LOCATE lo loc line ele</td>
<td>loc defaults to center (IC) if keyword PLACE is not specified</td>
</tr>
<tr>
<td>MAG = Imag emag</td>
<td>MAG Imag emag</td>
<td></td>
</tr>
<tr>
<td>PLACE = loc</td>
<td>none</td>
<td>sets loc for the LATLON, LINELE, and STATION keywords</td>
</tr>
<tr>
<td>RTIME = bmin emin</td>
<td>TIME bmin etime</td>
<td>keyword RTIME overrides keyword TIME</td>
</tr>
<tr>
<td>STATION = stn</td>
<td>LOCATE Eloc lat lon</td>
<td>loc defaults to center (EC) if keyword PLACE is not specified</td>
</tr>
<tr>
<td>TIME = btime etime</td>
<td>TIME btime etime</td>
<td></td>
</tr>
</tbody>
</table>
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 mcain 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 mcafreen function readiness 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 6-1 shows the arrangement of data points for an image line of 1-byte data containing three spectral bands.

Figure 6-1. 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 5, MclDAS Utilities.
Below is a code fragment showing the steps for reading an image object data block. Note that the \texttt{meaget} call must precede a call to \texttt{mcalin}.

\begin{verbatim}
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
    end if
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)
    ...
\end{verbatim}
**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 6-2, an image line consists of an optional line prefix and the actual data values.

*Figure 6-2. 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
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
    else if( status.eq.0 ) then
        C --- got a line of data
    endif
C --- read the line prefix
pfstatus = mcapfx(handle, prefix_buffer)
    if( pfstatus.lt.0 ) then
        call edest('Prefix Read failed',0)
        goto 100
    endif
C --- process the data
...
goto 100
endif
...```
**Reading the navigation block**

The `mcnav` 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
...
... read the navigation block
    status = mcnav(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
...
... 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

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( mcacrchandle, 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcaput</td>
<td>opens a connection to write the directory, navigation and calibration blocks of an image object</td>
</tr>
<tr>
<td>mcaout</td>
<td>writes the line prefix and data portions of an image line</td>
</tr>
<tr>
<td>mcacou</td>
<td>writes the comment block of an image object</td>
</tr>
</tbody>
</table>

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 non-zero, the specified number of 80-byte entries is transferred. The `meacou` 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 `meacou` code example, see the function `imgecopy.f`.

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

c --- create a comment card
     call getday( day )
     call gettime( 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
     Note: "navigation_params" is an array of navigation parameters
     that describes the geo-location of the elements of the
     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
     Note: "calibration_params" is an array of calibration parameters
     that transforms the data elements to physical units.
     do 20 i = 1, cal_size
cal_block(i) = calibration_params(i)
continue

--- fill the selection array
nselect = 1
selects(nselect) = 'POS '//cfu(position)

--- 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( mcapout( data_buffer ).ne.0 ) then
    call edest('failed to write image line=',line)
    return
endif

100 continue

c --- write the comment block
if( mcapout( 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 bsque2( selects(nselect) )

--- 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 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.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Grid data</th>
<th>Image data</th>
</tr>
</thead>
<tbody>
<tr>
<td>data volume</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>data value resolution</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>geographic resolution between data points</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>representation to users</td>
<td>graphical contours</td>
<td>gray shading</td>
</tr>
</tbody>
</table>

This section describes:

- the attributes unique to grid data
- the blocks of information contained in a grid object
- the API functions available for reading and writing grid data
- the 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 ETA, NGM or MRF. 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 Image Window.

McIDAS 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 navigation, see the section titled McIDAS Navigation later in this chapter.

---

**Figure 6-3.** Pseudo-Mercator projection.
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 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 library functions for performing these tasks.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcgdrd</td>
<td>reads the grid header from a grid object</td>
</tr>
<tr>
<td>mcgget</td>
<td>opens a connection to read the data block of a grid object</td>
</tr>
<tr>
<td>mcgdir</td>
<td>opens a connection to read the grid header of a grid object</td>
</tr>
<tr>
<td>mcgfdrd</td>
<td>retrieves the grid file header</td>
</tr>
<tr>
<td>mcgridc</td>
<td>reads the grid object and returns the grid in row-major (C) format</td>
</tr>
<tr>
<td>mcgridf</td>
<td>reads the grid object and returns the grid in column-major (Fortran) format</td>
</tr>
<tr>
<td>m0gsort</td>
<td>gets the parameters from the command line and adds them to the selection array for future mcgget call</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Selection clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAY day1 .. dayn</td>
<td>list of primary grid days</td>
</tr>
<tr>
<td>DRANGE bday eday inc</td>
<td>range of primary grid days with a day increment</td>
</tr>
<tr>
<td>FDAY day</td>
<td>forecast day</td>
</tr>
<tr>
<td>FHOUR hour1 .. hourn</td>
<td>list of forecast hours</td>
</tr>
<tr>
<td>FRANGE bvt evt inc</td>
<td>range of forecast hours with an hour increment</td>
</tr>
<tr>
<td>FTIME time</td>
<td>forecast time</td>
</tr>
<tr>
<td>GRID bgrid egrid</td>
<td>specific range of grids in a grid file</td>
</tr>
<tr>
<td>LEV lev1 .. levn</td>
<td>list of data levels</td>
</tr>
<tr>
<td>NUM numgrid</td>
<td>number of grids to find</td>
</tr>
<tr>
<td>OUT option</td>
<td>header output format to return</td>
</tr>
<tr>
<td>PARM p1 .. pn</td>
<td>list of parameters</td>
</tr>
<tr>
<td>POS offset</td>
<td>relative position offset in a dataset</td>
</tr>
<tr>
<td>PRO projection</td>
<td>grid projection type</td>
</tr>
<tr>
<td>SRC src1 .. srcn</td>
<td>list of sources that generated the grid</td>
</tr>
<tr>
<td>TIME time1 .. timen</td>
<td>list of primary grid times</td>
</tr>
<tr>
<td>TRANGE btime etime inc</td>
<td>range of primary grid times with a time increment</td>
</tr>
</tbody>
</table>

**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.

**GRID**—use this clause to access grids based on their position in specific grid files. Since this is an artifact of previous McIDAS 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*. 

---

McIDAS Programmer's Manual
Revised 5/98

Accessing Data
6-39
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.

<table>
<thead>
<tr>
<th>Command line keyword</th>
<th>Equivalent selection clause</th>
<th>m0gsort restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAY = d1 .. dn</td>
<td>DAY</td>
<td>cannot use with DRANGE</td>
</tr>
<tr>
<td>DRANGE = bday eday inc</td>
<td>DRANGE</td>
<td>cannot use with DAY</td>
</tr>
<tr>
<td>FDAY = day</td>
<td>FDAY</td>
<td>cannot use with FHOURL or FRANGE</td>
</tr>
<tr>
<td>FHOURL = h1 .. hn</td>
<td>FHOURL</td>
<td>cannot use with FDAY, FRANGE or FTIME</td>
</tr>
<tr>
<td>FRANGE = bhr ehr inc</td>
<td>FRANGE</td>
<td>cannot use with FDAY, FHOURL or FTIME</td>
</tr>
<tr>
<td>FTIME = time</td>
<td>FTIME</td>
<td>cannot use with FHOURL or FRANGE</td>
</tr>
<tr>
<td>GPRO = g1 .. gn</td>
<td>PRO</td>
<td>validate with projection: MERC, PS, LAMB, EQUI</td>
</tr>
<tr>
<td>GRID = bgrid eggrid</td>
<td>GRID</td>
<td>use LAST to get the last grid in the dataset and position; when specified, all other selection conditions are ignored</td>
</tr>
<tr>
<td>LEV = l1 .. ln</td>
<td>LEV</td>
<td></td>
</tr>
<tr>
<td>PARAM = p1 .. pn</td>
<td>PARM</td>
<td></td>
</tr>
<tr>
<td>SRC = s1 .. sn</td>
<td>SRC</td>
<td></td>
</tr>
<tr>
<td>TIME = t1 .. tn</td>
<td>TIME</td>
<td>cannot use with TRANGE</td>
</tr>
<tr>
<td>TRANGE = btim etim inc</td>
<td>TRANGE</td>
<td>cannot use with TIME</td>
</tr>
</tbody>
</table>
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 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.

```c
character*32 selects(5)
character*32 dataset
integer grid_header(64)
integer file_header(64)
integer nselects
integer error_flag
integer status

--assign the dataset name
dataset = 'RTGRIDS/ALL'

--assign the selection conditions to retrieve six grids from
--the dataset RTGRIDS/ALL that are from the ETA, NGM or
--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

--set an error flag to print a message if an error occurs
error_flag = 1

--open the connection to the server
status = mcgdir(dataset, nselect, selects, error_flag)
if (status .lt. 0) then
  return
endif

--every time statement 100 is reached, try to read a new grid
--file header
100 continue

hread = mcgfdrd(file_header)

--if you have successfully read the grid file header
if (hread .eq. 0) then

-- every time statement 200 is reached, try to read a new grid header
```

McIDAS Programmer's Manual
Issued 8/97

Accessing Data
6-41
200 continue

grread = mcgdrd(grid_header)

--- if you have successfully read the grid header
if (grread .eq. 0) then
  (process grid header here)

--- see if there are any more grid headers to read
  goto 200

--- if you have read the last grid from this file, go see if there
--- are any more grid files to read from
elseif (grread .eq. 1) then
  goto 100

--- if there was a problem reading the grid header
elseif (grread .lt. 0) then
  return
  endif

elseif (hdread .lt. 0) then
  call sdest('Unable to read grid file header', 0)
  endif

--- if you make it to here, hread has returned the value 1,
--- 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.

<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>total size; rows * columns (not to exceed the value of MAXGRIDPT in gridparm.inc)</td>
</tr>
<tr>
<td>2</td>
<td>number of rows</td>
</tr>
<tr>
<td>3</td>
<td>number of columns</td>
</tr>
<tr>
<td>4</td>
<td>Julian date of the data, yyddd</td>
</tr>
<tr>
<td>5</td>
<td>time of the data, hHmmss</td>
</tr>
<tr>
<td>6</td>
<td>forecast time for the grid, if applicable</td>
</tr>
<tr>
<td>7</td>
<td>name of the gridded variable</td>
</tr>
<tr>
<td>8</td>
<td>scale of the gridded variable</td>
</tr>
<tr>
<td>9</td>
<td>units of the gridded variable</td>
</tr>
</tbody>
</table>
| 10          | value of the vertical level
              | 1013 = 'MSL'
              | 999  = ' '     
              | 0    = 'TRO'
              | 1001 = 'SFC'
              (Otherwise, it is displayed as entered.) |
<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>scale of the vertical level</td>
</tr>
<tr>
<td>12</td>
<td>unit of the vertical level</td>
</tr>
</tbody>
</table>
| 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 - 15     | used if the grid parameter is a time difference or time average, hhmms |
| 16 - 32     | reserved |
| 33          | grid origin; identifies the type of program that generated the grid data |
| 34          | projection type:  
|             | 1 = pseudo-Mercator  
|             | 2 = polar stereographic or Lambert conformal  
|             | 3 = equidistant  
|             | 4 = pseudo-Mercator (more general)  
|             | 5 = no navigation  
|             | 6 = Lambert conformal tangent cone |
| 35 - 40     | varies, depending on the grid type; see the GRIDnnnn data file in Chapter 7 for more information |
| 41 - 52     | reserved |
| 53 - 64     | grid directory |

**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 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 mcain to get an entire object, mcgridf and mcgridc return an entire grid object with each call. Below is a sample code fragment demonstrating a mcget/mcgridf calling pair. Note that the mcget call must occur before mcgridf.

```c
#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) = 'PRT=T'
selects(5) = 'LEV=500'
selects(6) = 'FHRG=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 = mcget(dataset, nselect, selects, 'C', 'I4',
                   &maxpts*4, 1, numgrids, totbytes)

    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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcgoutc</td>
<td>writes a grid object stored in row-major format to a file</td>
</tr>
<tr>
<td>mcgoutf</td>
<td>writes a grid object stored in column-major format to a file</td>
</tr>
<tr>
<td>mcgput</td>
<td>opens a connection to write a grid object</td>
</tr>
</tbody>
</table>

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:

- a valid dataset name and position number, which is entered as one of the selection conditions in the mcgput call
- a 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.

<table>
<thead>
<tr>
<th>Selection clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEL = YES or DEL = NO</td>
<td>deletes the destination dataset file before recreating it to write the grid object (default = NO)</td>
</tr>
<tr>
<td>GRID = num</td>
<td>grid number in a dataset location to write the grid object to; the previous grid stored in this location is overwritten</td>
</tr>
<tr>
<td>LABEL =</td>
<td>label to attach to the dataset when the file is created</td>
</tr>
<tr>
<td>MAX = num</td>
<td>maximum number of grid objects that can be stored in the newly created file</td>
</tr>
<tr>
<td>NUM =</td>
<td>number of grid objects to write to the dataset</td>
</tr>
<tr>
<td>POS = pos</td>
<td>position number in the dataset to write the grid object to</td>
</tr>
</tbody>
</table>
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 an error. The mcgoutf and mcgoute functions send the grid objects to the server. mcgoutf assumes the data block is stored in column-major order; mcgoute assumes the data block is stored in row-major order. You must call mcgput before calling mcgoutf or mcgoute, as shown in the sample code fragment below.

```c
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, l, total_bytes)
```
if (ok .lt. 0) then
    return
end if

c---write the first grid object

ok = mcgoutf(grid1, grid_h1)
if (ok .lt. 0) then
    return
end if

c---write the second grid object

ok = mcgoutf(grid2, grid_h2)
if (ok .lt. 0) then
    return
end if

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.

\[
total\ bytes = 8 + 4 \times ((number\ of\ grids \times 64) + (number\ of\ data\ points) + (number\ of\ grids))
\]

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.

\[8 + 4 \times ((2 \times 64) + (200 + 300) + 2) = 2528\]
Point data

McIDAS 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:

- the attributes unique to point data
- the blocks of information contained in a point object
- the API functions available for reading both the point data and the file header
- the selection conditions you can use when requesting point data

Note: at the time this manual was published, the APIs for many of the access routines were not completely stabilized; therefore, they have an M0 prefix.
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

Currently, the point data servers in McIDAS only return data stored in McIDAS MD (Meteorological Data) files. When accessing point data, the MD 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.

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 alphanumerical 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 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 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 command PTLIST is an example of an application that reads point objects. The table below lists alphabetically the McIDAS library functions for acquiring point data.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m0ptget</td>
<td>opens a connection to read a point object</td>
</tr>
<tr>
<td>m0ptparm</td>
<td>extracts the parameter information from the command line</td>
</tr>
<tr>
<td>m0ptrd</td>
<td>reads the point data block from the server</td>
</tr>
<tr>
<td>m0psort</td>
<td>gets the selection parameters from the command line and adds them to a selection array used by m0ptget</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Selection clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>maximum number of point objects to find (default = 1)</td>
</tr>
<tr>
<td>POS</td>
<td>position number in a dataset to retrieve data from</td>
</tr>
<tr>
<td>SELECT s1 .. sn</td>
<td>list of selection conditions</td>
</tr>
</tbody>
</table>
**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.

**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°F and 32°F 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 command line syntax `PARAM=parameters[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 library contains a group of functions for extracting these mixed data type values. You can call the m0ptbufininit 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.

```fortran
subroutine main0

implicit none
include 'ptparm.inc'
integer MAXBYTE ! max bytes the buffer may contain
parameter (MAXBYTE = MAXNUMP * 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(MAXNUMP) ! list of parameters
character*4 units(MAXNUMP) ! list of units
character*4 form(MAXNUMP) ! list of return forms
integer scales(MAXNUMP) ! list of scales
integer buffer(MAXNUMP) ! 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 m0ptbufininit
integer m0ptbufc
integer m0ptbuff
integer m0ptbufi
integer m0ptget
integer m0ptrd
data units/MAXNUMP * "/
quote = char (39)
c--- assign the dataset
dataset = 'RTFTSRC/SFCHOURLY'
```

McIDAS Programmer's Manual
Issued 8/97

Accessing Data
6-55
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 mupsort, 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 96274"/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 96274'

  nselect = 2

c--- make the request to the server

  ok = muptget (dataset, nselect, select, nparams, params,
        & units, form, scales, MAXBYTE, 1)

  if (ok .lt. 0) then
     goto 999
  endif

  c--- upon a successful return from muptget, 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 muptbufc/f/i can be used to
  c--- extract the buffer values

  ok = muptbufinit (nparams, form, scales)
  if (ok .lt. 0) then
     goto 999
  endif

  c--- continuously call muptrd 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 = muptrd (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 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.

<table>
<thead>
<tr>
<th>Selection clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPOS</td>
<td>beginning file in the dataset or ALL</td>
</tr>
<tr>
<td>EPOS</td>
<td>ending file in the dataset</td>
</tr>
</tbody>
</table>

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**.

```c
#include 'ptparm.inc'
character*24 dataset
ingeger   header(HEADSIZE)
character*24 selects(2)
integer   nselects

c--- request file header positions 1 through 5 from the dataset
c--- RTPTSRCSFCHOURLY

dataset = 'RTPTSRCSFCHOURLY'
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
  endif

  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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>schema name</td>
</tr>
<tr>
<td>1</td>
<td>schema version number</td>
</tr>
<tr>
<td>2</td>
<td>schema registration date</td>
</tr>
<tr>
<td>3</td>
<td>default number of rows</td>
</tr>
<tr>
<td>4</td>
<td>default number of columns</td>
</tr>
<tr>
<td>5</td>
<td>total number of keys in the record</td>
</tr>
<tr>
<td>6</td>
<td>number of keys in the row header</td>
</tr>
<tr>
<td>7</td>
<td>number of keys in the column header</td>
</tr>
<tr>
<td>8</td>
<td>number of keys in the data portion</td>
</tr>
<tr>
<td>9</td>
<td>1-based position of the column header</td>
</tr>
<tr>
<td>10</td>
<td>1-based position of the data portion</td>
</tr>
<tr>
<td>11</td>
<td>number of repeat groups</td>
</tr>
<tr>
<td>12</td>
<td>size of the repeat group</td>
</tr>
<tr>
<td>13</td>
<td>starting position of the repeat group</td>
</tr>
<tr>
<td>14</td>
<td>missing data code</td>
</tr>
<tr>
<td>15</td>
<td>integer ID of the file</td>
</tr>
<tr>
<td>16 - 23</td>
<td>text ID of the file</td>
</tr>
<tr>
<td>24</td>
<td>creator’s project number</td>
</tr>
<tr>
<td>25</td>
<td>creation date</td>
</tr>
<tr>
<td>26</td>
<td>creator’s ID</td>
</tr>
<tr>
<td>27</td>
<td>zero-based offset to the row header</td>
</tr>
<tr>
<td>28</td>
<td>zero-based offset to the column header</td>
</tr>
<tr>
<td>29</td>
<td>zero-based offset to the data portion</td>
</tr>
<tr>
<td>30</td>
<td>first unused word in the file</td>
</tr>
<tr>
<td>31</td>
<td>start of the user record</td>
</tr>
<tr>
<td>32</td>
<td>start of the key names</td>
</tr>
<tr>
<td>33</td>
<td>start of the scale factors</td>
</tr>
<tr>
<td>34</td>
<td>start of the units</td>
</tr>
<tr>
<td>35 - 38</td>
<td>reserved</td>
</tr>
<tr>
<td>39</td>
<td>beginning Julian day of the data</td>
</tr>
<tr>
<td>40</td>
<td>beginning time of the data, hhmmss</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>41</td>
<td>ending Julian day of the data</td>
</tr>
<tr>
<td>42</td>
<td>ending time of the data, hhmmss</td>
</tr>
<tr>
<td>43 - 60</td>
<td>reserved</td>
</tr>
<tr>
<td>61 - 62</td>
<td>file name</td>
</tr>
<tr>
<td>63</td>
<td>MD file number</td>
</tr>
<tr>
<td>64 - 463</td>
<td>user record, MD coordinates (0,0); not described by the schema; use for storing arbitrary information</td>
</tr>
<tr>
<td>464 - 863</td>
<td>names of the file keys</td>
</tr>
<tr>
<td>864 - 1264</td>
<td>scale factors for the keys</td>
</tr>
<tr>
<td>1264 - 1663</td>
<td>units of the keys</td>
</tr>
<tr>
<td>1664 - 4095</td>
<td>reserved</td>
</tr>
</tbody>
</table>
Text data

Text data stores a variety of information for the McIDAS user, such as:

- administrative messages
- forecasts, advisories and warnings
- observational data, such as METAR or RAOB reports

McIDAS 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 data
- the blocks of information contained in text data
- the API functions available for reading text data

Note: At the time this manual was published, the APIs for the text data routines were not fully stabilized; therefore, they have an M0 prefix.

Flat-file text

Flat-file text is the simplest text data available in McIDAS. It is used most often on the server machine to convey administrative or configuration information to the user. The McIDAS 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 library functions for flat file text.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0textgt</td>
<td>opens a connection to read a flat file</td>
</tr>
<tr>
<td>M0txtread</td>
<td>reads the text data one line at a time</td>
</tr>
<tr>
<td>M0PrintTextFromServer</td>
<td>opens a connection to read a flat file and prints the file contents to the McIDAS Text Window</td>
</tr>
</tbody>
</table>

**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.

```c
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, 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 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 FMKE 251641
SFPFW
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:

<table>
<thead>
<tr>
<th>WMO product header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>advisory</td>
</tr>
<tr>
<td>C</td>
<td>climatic</td>
</tr>
<tr>
<td>F</td>
<td>forecast</td>
</tr>
<tr>
<td>N</td>
<td>notice</td>
</tr>
<tr>
<td>R</td>
<td>river</td>
</tr>
<tr>
<td>S</td>
<td>surface</td>
</tr>
<tr>
<td>T</td>
<td>satellite</td>
</tr>
<tr>
<td>U</td>
<td>upper air</td>
</tr>
</tbody>
</table>
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 library functions for reading general weather text.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0wtxget</td>
<td>opens a connection to read the text</td>
</tr>
<tr>
<td>M0wtxread</td>
<td>reads the text</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Selection clause</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>APRO= p1 .. pn</td>
<td>list of AWIPS product types to match</td>
<td>three characters; don’t use with WMO=</td>
</tr>
<tr>
<td>ASTN= s1 .. sn</td>
<td>list of AWIPS station IDs to match</td>
<td>two or three characters</td>
</tr>
<tr>
<td>DAY=</td>
<td>most recent day to search</td>
<td></td>
</tr>
<tr>
<td>DTIME=</td>
<td>maximum number of hours of reports to search</td>
<td></td>
</tr>
</tbody>
</table>
### Selection clause

<table>
<thead>
<tr>
<th>Selection clause</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATCH = match1 .. n</td>
<td>list of strings within the body of the text to match</td>
<td>if more than one match string is requested, all match strings requested must be found in each text block for a successful return</td>
</tr>
<tr>
<td>NUM =</td>
<td>maximum number of text blocks to list</td>
<td></td>
</tr>
<tr>
<td>PROD =</td>
<td>predefined product name</td>
<td>to list the available predefined products, use WXTLIST DIR</td>
</tr>
<tr>
<td>SOURCE = s1 .. sn</td>
<td>list of circuit sources</td>
<td>seldom used</td>
</tr>
<tr>
<td>WMO = w1 .. wn</td>
<td>list of WMO headers to match</td>
<td>minimum of two characters; wildcard characters are allowed; see the help for the WXTLIST command; can't be used with APRO =</td>
</tr>
<tr>
<td>WSTN = s1 .. sn</td>
<td>list of WMO station IDs to match</td>
<td>four characters</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 3</td>
<td>circuit source</td>
</tr>
<tr>
<td>4 - 7</td>
<td>number of bytes of data</td>
</tr>
<tr>
<td>12 - 15</td>
<td>time of day that the data was received, hhmmss</td>
</tr>
<tr>
<td>16 - 19</td>
<td>4-character WMO product ID, such as FPUS</td>
</tr>
<tr>
<td>20 - 23</td>
<td>product number</td>
</tr>
<tr>
<td>24 - 27</td>
<td>4-character WMO station ID</td>
</tr>
<tr>
<td>28 - 31</td>
<td>3-character AWIPS product ID</td>
</tr>
<tr>
<td>32 - 35</td>
<td>2- or 3-character AWIPS station ID</td>
</tr>
<tr>
<td>36 - 39</td>
<td>3-character product origin (optional)</td>
</tr>
<tr>
<td>40 - 43</td>
<td>Julian day of the data</td>
</tr>
<tr>
<td>44 - 47</td>
<td>number of bytes per line</td>
</tr>
<tr>
<td>60 - 63</td>
<td>FAA catalog number (optional)</td>
</tr>
</tbody>
</table>
The sample code below demonstrates an ADDE general weather text request.

```c
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, "AFRO=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 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 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’ 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
- the APIs you will use with the navigation subsystems
- how to design a navigation type and implement an image navigation module

Basic concepts

To select and use the best McIDAS 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 data has an associated type and a full set of parameters defining an instance.

Navigation subsystems

McIDAS 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 two types of projections: pseudo-Mercator and Lambert conformal/polar stereographic. Both 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 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 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 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 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 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.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m0distnc</td>
<td>calculates the great-circle distance on a sphere</td>
</tr>
<tr>
<td>geolat</td>
<td>performs geodetic/geocentric latitude conversions</td>
</tr>
<tr>
<td>nlhxyz, nxyzl</td>
<td>perform spherical/Cartesian conversions</td>
</tr>
<tr>
<td>raerac, racrae</td>
<td>perform terrestrial/celestial longitude conversions</td>
</tr>
<tr>
<td>solarp</td>
<td>calculates solar position</td>
</tr>
<tr>
<td>cartll, llcart, llobl, llopt</td>
<td>perform generic spherical/Cartesian conversions</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grddef</td>
<td>initializes the grid transformation</td>
</tr>
<tr>
<td>ijll</td>
<td>converts grid coordinates to earth coordinates</td>
</tr>
<tr>
<td>llij</td>
<td>converts earth coordinates to grid coordinates</td>
</tr>
</tbody>
</table>

No API functions currently exist for creating grid headers for a projection. When creating McIDAS 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 7.
**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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>illtv</td>
<td>converts earth coordinates to image and frame coordinates</td>
</tr>
<tr>
<td>itvll</td>
<td>converts frame coordinates to image and earth coordinates</td>
</tr>
</tbody>
</table>

The only prerequisite for these functions is that the frame has navigation associated with it. The McIDAS IMGDISP command creates frame navigation when it displays an image dataset. The McIDAS 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 PC command.

```c
zlat = 43.13
zlon = 89.35
if(illtv(iframe,gnunm,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 E command.

```c
if( mcmoubtn( 0, left, right, itvlin, itvele ) ) then
  // 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvprep</td>
<td>dynamically loads a navigation module for a specified slot, given a set of parameters</td>
</tr>
<tr>
<td>nvset</td>
<td>initializes navigation for a specified slot from a specified source</td>
</tr>
<tr>
<td>nvxeas</td>
<td>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</td>
</tr>
<tr>
<td>nvxini</td>
<td>initializes the navigation module; due to dynamic linking, this function is called nv1ini, nv2ini or nv3ini from applications, depending on the navigation slot used</td>
</tr>
<tr>
<td>nvxopt</td>
<td>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</td>
</tr>
<tr>
<td>nvxsae</td>
<td>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</td>
</tr>
</tbody>
</table>

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 (dnaw) 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
 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, dnaw(1) ).ne.0 ) then
      navinit = -1
    return
 endif
    init = 1
 endif
 c --- initialize the navigations
 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
 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 = nvlneas( dlat, dlon, dz, sline, selement, dz)
    if( status.ne.0 ) then
      crossnav = -2
    return
 endif
```

When the navigation block source is a local McIDAS 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`.

```c
 nvset ('area', area_num)
 -or-
 nvset ('frame', frame_num)
```
In addition to the commonly used conversions between earth and image coordinates available through *nvxsa* and *nvxra*, 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.

<table>
<thead>
<tr>
<th>Special services</th>
<th>Modules that support them</th>
</tr>
</thead>
<tbody>
<tr>
<td>oblate radius</td>
<td>nvxmerc, nvxps, nvxrect</td>
</tr>
<tr>
<td>orbital period</td>
<td>nvxdmsp</td>
</tr>
<tr>
<td>parallax correction</td>
<td>nvxgoes, nvxgvar, nvxmsat</td>
</tr>
<tr>
<td>satellite location/subpoint</td>
<td>nvxdmsp, nvxgoes, nvxgraf, nvxgvar, nvxlamb, nvxmerc, nvxmsat, nvxps, nvxradr, nvxrect, nvxsin, nvxtiro</td>
</tr>
<tr>
<td>view angles (satellite, solar, etc.)</td>
<td>nvxgoes, nvxgvar</td>
</tr>
</tbody>
</table>

The sample code below shows the use of the **nvxopt**.

```
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 --- fill the input vector, call nvxopt(), and
C --- extract the outputs
    xin(1) = 96158.
    xin(2) = 13.00
    if( nvlopt( lit('SPOS'), xin, xout ) .ne.0 ) then
C    (error handler)
    end if
    geodetic_lat = xout(1)
    lon = xout(2)
    cd = xout(3)        ! distance from earth center
    x = xout(4)
    y = xout(5)
    z = xout(6)
```
Implementing an image navigation module

You can extend image navigation to include a new type and make it available to all McIDAS 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:

- the contents of the navigation block
- the module name
- the names and calling sequences of the functions that it implements
- retention of parameters between calls
- its 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 nvxtype.dlm, where type is the same four characters specified in the first word of the block. The nvpres function uses the type from the block to link the application with the correct module.
Functions

The navigation module must implement the functions \texttt{nvxini}, \texttt{nvxsae}, \texttt{nvxeas} and \texttt{nvxopt}, with calling sequences following the documentation blocks in the tangent cone example. In particular, \texttt{nvxini} must support modes 1 (initialize) and 2 (set earth coordinate mode 'LL' (Latitude-longitude) or 'XYZ' (Cartesian)). The \texttt{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 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 6-6 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 $\psi$ and longitude $\lambda$ corresponds to a point $(R, \theta)$ in polar coordinates on the flattened cone.

*Figure 6-6. Tangent cone projection geometry.*
A published\(^1\) form of the equations defining the location \((R, \theta)\) on the flattened cone in terms of an earth location \((\psi, \lambda)\) is:

\[
R = a \tan \psi_0 \left( \frac{\tan \frac{\psi}{2}}{\left( \tan \frac{\psi_0}{2} \right)^2} \right)^{\cos \psi_0}
\]  

(1)

\[
\theta = \cos \psi_0 (\lambda - \lambda_0)
\]  

(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.

\[
M(\psi) = m \left( \frac{\sin \psi_0}{\sin \psi} \right) \left( \frac{\tan \psi}{\tan \psi_0} \right)^{\cos \psi_0}
\]  

(3)

As you can see from (1) and (2), the planetary radius, \(a\) the standard colatitude \(\psi_0\), and the standard longitude \(\lambda_0\) 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 implementation. The inverse relationship capable of converting earth location to projection coordinate \((R, \theta)\) must be derived, the earth location \((\lambda, \psi)\) related to its equivalent in McIDAS \((\text{LAT}, \text{LON})\), and the projection coordinate \((R, \theta)\) related to McIDAS image coordinates. The inverse is:

\[
\lambda = \lambda_0 + \frac{\theta}{\cos \psi_0}
\]  

(4)

and

\[
\psi = 2 \tan^{-1} \left[ \tan \frac{\psi_0}{2} \left( \frac{R}{a \tan \psi_0} \right)^{\frac{1}{\cos \psi_0}} \right]
\]  

(5)

---

The relationships between McIDAS planetary coordinates and \((\lambda, \psi)\) are:

\[
\psi = \frac{\pi}{2} - \frac{\pi}{180} \text{Lat} \quad \lambda = -\frac{\pi}{180} \text{Lon}
\]  
(6)

and

\[
\text{Lat} = 90 - \frac{180}{\pi} \psi \quad \text{Lon} = -\frac{180}{\pi} \lambda
\]  
(7)

Those between projection and image coordinates are:

\[
L = L_0 + \frac{R \cos \theta}{m} \quad E = E_0 + \frac{R \sin \theta}{m}
\]  
(8)

and

\[
R = m \sqrt{(L-L_0)^2 + (E-E_0)^2}
\]  
(9)

\[
\theta = \tan^{-1} \left(\frac{(E-E_0)}{(L-L_0)}\right)
\]

Adding (6) - (9) to make a complete McIDAS navigation algorithm adds additional parameters to make the complete set \(\psi_0, \lambda_0, m, L_0,\) 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, however
- \(-\pi < \lambda \leq \pi\)  
  all longitudes are navigable

**Image coordinates**

- \(R \geq 0\)
- \(\pi \cos(\psi_0) < \theta \leq \pi \cos(\psi_0)\)
  no valid earth location maps to \(-R\)
  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, 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 to projection form.

\[
\begin{align*}
\psi &= \frac{\pi}{2} - D2R \cdot \text{clat} \\
\psi_0 &= \frac{\pi}{2} - D2R \cdot \text{slat} \\
\lambda &= - D2R \cdot \text{clon} \\
\lambda_0 &= - D2R \cdot \text{slon}
\end{align*}
\]

Next, the radius of the specified image coordinate center point (relative to the projection center or the pole) is computed.

\[
\begin{align*}
\text{radius} &= A \cdot \tan(\psi_0) \\
\theta &= \tan(\psi/2)/\tan(\psi_0/2) \cdot \cos(\psi_0) \\
\theta &= \cos(\psi_0) \cdot (\lambda - \lambda_0)
\end{align*}
\]

Then the pole's image coordinate is computed using simple trigonometry.

\[
\begin{align*}
call \text{ status } &= \text{ mcfsize}() \text{ lin_c, nlins, neles} \\
nlin_c &= \text{ dble}(\text{nlins}) / 2.0 \\
\text{ele}_c &= \text{ dble}(\text{neles}) / 2.0 \\
\text{lin}_0 &= \text{lin}_c - \text{radius} \cdot \cos(\theta)/\text{sscale} \\
\text{ele}_0 &= \text{ele}_c - \text{radius} \cdot \sin(\theta)/\text{sscale}
\end{align*}
\]

Once the parameters are computed, the navigation block can be filled and written either to the image dataset:

\[
\begin{align*}
\text{C} & \quad \text{// Create and insert the navigation block} \\
nvblk(1) &= \text{litr('TANC')} \quad \text{// Tangent Cone nav type} \\
nvblk(2) &= \text{nint(10000*lin_0)} \quad \text{// image line of pole} \\
nvblk(3) &= \text{nint(10000*ele_0)} \quad \text{// image element of pole} \\
nvblk(4) &= \text{nint(10000*sscale)} \quad \text{// scale km/pixel at slat} \\
nvblk(5) &= \text{nint(10000*slat)} \quad \text{// standard latitude} \\
nvblk(6) &= \text{nint(10000*slon)} \quad \text{// standard longitude} \\
\text{status} &= \text{mcaput}() \text{ dataset, nsort, sort, adir, nvblk, calblk)}
\end{align*}
\]

or a McIDAS area format image file:

\[
\begin{align*}
call \text{ araput(aranum,4*DIRSIZ,4*NVDDS,nvblk)}
\end{align*}
\]
Creating the navigation module

If the algorithm and navigation block are complete and unambiguous, implementing a map projection as a McIDAS 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 nxxini 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
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
               ! on projection surface
C     // Common block contents: constants
real  D2R       ! degrees to radians factor
real  Pi        ! returned when navigation
               ! cannot be done
real  Erad      ! Earth radius
logical Init     ! initialized flag
logical Latlon  ! .TRUE. for lat/ion 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 nxxini runs successfully, and Latlon is set if the earth coordinate mode is latitude and longitude rather than Cartesian.

The overall structure of nxxini 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 6-6, are not navigable.
Recognizing such points is possible only after the conversion from
McIDAS image coordinates to projection coordinates is complete. The
code below is from lines 0370 through 0384.

```c
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
clerared 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 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.
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
0021 *$ param(5) = standard latitude *10000
0022 *$ param(6) = standard longitude *10000
0023 *$ for option 2:
0024 *$ param(1) = 'LL' or 'XYZ'
0025 *
0026 *$ Input and Output:
0027 *$ none
0028 *
0029 *$ Output:
0030 *
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
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 'convdim;' 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 implicit none
C // Interface variables (formal arguments)
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 ! codigil 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 //\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\n
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 'convdim'.)
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 ! returned when navigation
0111 real Badreal ! cannot be done
0112 C ! Earth radius
0113 logical Init ! initialized flag
0114 logical Latlon ! .TRUE. for lat/lon I/O
0115
0116 COMMON/TANC/ Lin0, Ele0, Scale, Lon0, Colat0,
0117 & Coscl, Tancl, Tancl2, Mxtheta,
0118 & D2R, Pi, Badreal, Erd, Init, Latlon
0119
0120 C End of common block variables and declaration.
0121
0122 C // Begin initialization process by setting values of constants.
0123
0124 Erd = 6370. ! This value of Erd is ok
0125 C ! for low precision nav
0126 C ! where a spherical Earth is
0127 C ! adequate (Saucier, p. 32)
0128 Pi = acos(-1.)
0129 D2R = Pi / 180.
Badreal = -1.E10 ! obvious unreasonable value
!
for nav transform result

C

// Process initialization options. Only one option, initialize
navigation parameters, is supported in this demo version,
but a 'hook' is left for an additional option to set the
output coordinate to something other than lat/lon

if( option.eq.1 ) then

    call DDEST('nvxini(tanc) option=1',0)
call movc(param(1),nvtyp)
if( nvtyp.eq.'TANC') then

C

    // Unpack tangent cone projection parameters

Lin0 = param(2) / 10000.
Ele0 = param(3) / 10000.
Scale = param(4) / 10000.
lat0 = param(5) / 10000.
Lon0 = param(6) / 10000.

write(cbuf,'(" nvxini: lat0, Lon0 ",2F12.4")
lat0, Lon0
call DDEST(cbuf,0)

C

    // apply range checking

if(Scale.le.0. ) then
    call DDEST('nvxini(tanc) scale is negative',0)
    Init = .FALSE.
    NVXINI= -3
    return
    end if

if(lat0.le.0. .or. lat0.ge.90. ) then
    call DDEST('nvxini(tanc) standard lat out of range',0)
    Init = .FALSE.
    NVXINI= -3
    return
    end if

if(Lon0.le.-180. .or. Lon0.gt.180. ) then
    call DDEST('nvxini(tanc) standard lon out of range',0)
    Init = .FALSE.
    NVXINI= -3
    return
    end if

C

    // convert degrees to radians and latitude to colatitude.

C

    // Account for McIDAS longitude convention

Lon0 = -Lon0 * D2R
Colat0 = Pi/2. - D2R*lat0

write(cbuf,'(" nvxini: Colat0, Lon0 ",2F12.4")
Colat0, Lon0
call DDEST(cbuf,0)

C

    // Compute intermediate quantities

Coscl = cos(Colat0)
Tanc1 = tan(Colat0)
Tanc12 = tan(Colat0/2.)
Mntheta = Pi*Coscl

write(cbuf,'(" nvxini: Coscl, Tanc1", 2F7.4)'

Coscl, Tanc1
call DDEST(cbuf,0)
write(cbuf,'(" nvxini: Tancl2, Mxtheta ", 2F7.4")

  tancl2, Mxtheta
call DDEST(cbuf,0)

  Latlon = .TRUE.

else ! option=1 but type not 'TANC'
call DDEST('nvxini(tanc) parameter type bad',0)
  Init = .FALSE.
  NVXINI = -4
return

end if

else if ( option .eq. 2 ) then
call movw(param(1),outcoord)
if ( outcoord.eq.'LL' ) then
  Latlon = .TRUE.
else if ( outcoord.eq.'XYZ') then
  Latlon = .FALSE.
else
call DDEST('option=2 coord '//outcoord//' not supported',0)
  Init = .FALSE.
  NVXINI = -5
end if

else ! option not 1 or 2
call DDEST('nvxini(tanc) unrecognized output option ',option)
  NVXINI = -4
return

end if

NVXINI = 0
Init = .TRUE.
return

end

*$ Name:
nvxsa - Compute earth coordinates from image coordinates

*$ Interface:
integer function
nvxsa( real lin, real ele, real dummy,
  real e1, real e2, real e3 )

*$ Input:
lin - image line
ele - image element
dummy - (unused)

*$ Input and Output:
  none

*$ Output:
e1 - latitude or x
e2 - longitude or y
e3 - height or z

*$ Return values:
  0 - success
  -1 - input data physically valid but not navigable
given the specified projection
module not initialized

Remarks:
The navigation module must first be initialized with
a call to nvnini(). The output form \((x,y,z)\) or \((x,y,z)\)
depends on the last call to nvnini() with option 2.

Categories:

navigation

INTEGER FUNCTION NVXSAE( lin, ele, dummy, e1, e2, e3 )
implict NONE

C // Interface variables (formal arguments)
real lin ! image line to navigate
real ele ! image element to navigate
real dummy ! (unused argument)
real e1 ! Earth coordinate 1
real e2 ! Earth coordinate 2
real e3 ! Earth coordinate 3

C // Local variables
real lat ! latitude (McIDAS convention)
real lon ! longitude (McIDAS convention)
real hgt ! height
real dx ! zonal displacement from pole
real dy ! meridional displacement from pole
real radius ! distance from pole on projection surface
real theta ! angle from standard longitude on projection surface
real colat ! colatitude of navigated point

C C // Common block variables and declaration.
C C ALL CODE BETWEEN THE '//////' SEPARATORS MUST BE
C C DUPLICATED EXACTLY IN EACH NAVIGATION ROUTINE
C C (A more maintenance-safe version would use ENTRY points
C C rather than separate functions for the navigation API
C C but entry points cannot be processed by 'convdlm',)

C // Common block contents: projection parameters
real Lin0 ! image line of pole
real Ele0 ! image element of pole
real Scale ! km per unit image coordinate
(real ! (pixel)
real Lon0 ! standard longitude
real Colat0 ! standard colatitude

C // Common block contents: pre-computed intermediate values
real Coscl ! cosine(Colat0)
real Tancl ! tangent(Colat0)
real Tancl2 ! tangent(Colat0/2)
real Mtheta ! limit of angle from std. lon

C C // Common block contents: constants
real D2R ! degrees to radians factor
real Pi
real Badreal ! returned when navigation
C real Erad ! cannot be done
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

C End of common block variables and declaration.
C
C // verify initialized module
C
if(.not.Init) then
  NVXSAE = -6
  return
end if

C // Compute radius and bearing from pole
C
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
C
if (theta.le.-Mxtheta .or. theta.gt.Mxtheta ) then
  NVXSAE = -1
  return
end if

C // Forward navigation: compute longitude and colatitude
C // from radius and theta
C
lon = Lon0 + theta/Coscl
if(lon.le.-Pi) lon = lon + 2.d0*Pi
if(lon.gt.Pi) lon = lon - 2.d0*Pi
colat = 2.*atan( Tancl2 * (radius/(Erad*Tancl))**(1./Coscl))

C // Rescale to McIDAS convention (degrees, West positive).
C // Apply conversion to Cartesian coordinates if 'XYZ'. set
C // as output form. Set return code for success.
C
lon = -lon/D2R
lat = 90. - colat/D2R
hgt = 0.

if(.not.Latlon) then
  call nlkxyz(lat,lon,e1,e2,e3)
else
  e1 = lat
e2 = lon
e3 = 0.
end if

NVXSAE = 0
return
end
**Name:**
nvxeas - Compute image coordinates from earth coordinates

**Interface:**
integer function

**Input:**
- real e1 - latitude or x
- real e2 - longitude or y
- real e3 - height or z

**Input and Output:**
- real lin - image line
- real ele - image element
- real dummy - (unused)

**Return values:**
- 0 - success
- -1 - input data physically valid but not navigable given the specified projection
- -2 - input data exceed physical limits
- -6 - module not initialized

**Remarks:**
The navigation module must first be initialized with a call to nvxini(). The input form (lat,lon) or (x,y,z)
depends on the last call to nvxini() with option 2.

Input longitude may be in the range -360 to +360;

values outside this range will not be de-navigated.

Height (hgt) is ignored.

**Categories:**
- navigation

**Interface variables (formal arguments)**

**Local variables**

**Common block variables and declaration.**

// Interface variables (formal arguments)

// Local variables

// Source file for Common block variables and declaration.
A more maintenance-safe version would use ENTRY points rather than separate functions for the navigation API but entry points cannot be processed by 'convdim'."

// Common block contents: projection parameters
real Lin0 ! image line of pole
real Ele0 ! image element of pole
real Scale ! km per unit image coordinate
(real C ! (pixel)
real Lon0 ! standard longitude
real Colat0 ! standard colatitude

// Common block contents: precomputed intermediate values
real Coscl ! cosine(Colat0)
real Tancl ! tangent(Colat0)
real Tancl2 ! tangent(Colat0/2)
real Mtheta ! limit of angle from std. lon
(real C ! on projection surface

// Common block contents: constants
real D2R ! degrees to radians factor
real Pi !
real Badreal ! returned when navigation cannot be done
(real C ! Earth radius
logical Init ! initialized flag
logical Lation ! .TRUE. for lat/lon I/O

COMMON/TANCLin0, Ele0, Scale, Lon0, Colat0,
& Coscl, Tancl, Tancl2, Mtheta,
& D2R, Pi, Badreal, Erad, Init, Lation

End of common block variables and declaration.

/// C // verify that module is initialized
if(.not.init) then
  NVXEAS = -6
  return
end if

// Preprocess input values. If mode is 'XYZ' first convert
// from Cartesian to lat/lon. If mode is 'LL' just transcribe
// from arguments.

if(LatLon) then
  lat = e1
  lon = e2
  hgt = e3
else
  call nxyzll( e1, e2, e3, lat, lon)
  hgt = 0.
end if

// check that input values are physically possible and
// then convert to radians and East positive

if ( lat.lt.-90. .or. lat.gt.90. ) then
  NVXEAS = -2
  return
end if
if (lon.le. -360..or.lon.gt.360.) then
  NVXEAS = -2
  return
end if

if (lat.eq. -90..or.lat.eq.90.) then
  NVXEAS = -1
end if

colat = Pi/2. - D2R*lat
in_lon = -D2R*lon

C // map longitude into range -Pi to Pi

if (in_lon.le.-Pi) in_lon = in_lon + 2.*Pi
if (in_lon.gt.Pi) in_lon = in_lon - 2.*Pi

C // Now trap South Pole. Though a physically possible latitude,
C // tan(colat/2) → infinity there so it is not navigable
if (colat.eq.Pi) then
  NVXEAS = -1
  return
end if

C // Compute radius and theta of point on projection surface.
C // Theta is tricky; you have to compute offset relative
C // to standard longitude, force that into -pi to +pi range,
C // and THEN scale by cos(colat)0

radius = Erad * Tanc1 *( tan(colat/2.)/Tanc12 )** Cosc1
theta = in_lon-Lon0

if(theta.le.-Pi) theta = theta + 2.*Pi
if(theta.gt.Pi) theta = theta - 2.*Pi

theta = Cosc1 * theta

C // Compute line and element
lin = Lin0 + radius*cos(theta)/Scale
ele = Ele0 + radius*sin(theta)/Scale
dummy = 0.

NVXEAS = 0

return
end

*£ Name:
*nvoxopt - Perform supplemental navigation operations
*£ Interface:
*£ integer function
*£ nvoxopt(integer option, real xin(*),
*£ real xout(*)
*£ Input:
*£ option - "SCAL" compute projection scale
*£ xin(1) - latitude
*£ xout(*) - km per pixel at given latitude
*$ 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 *$ Remarks:
0633 *$ The navigation module must first be initialized by
0634 *$ a call to nviini(). Latitude is in degrees, north positive,
0635 *$ and must lie between -90. and +90.
0636 *$ Categories:
0638 *$ navigation
0639
0641 INTEGER FUNCTION NVXOPT( option, xin, xout)
0642 implicit NONE
0644 C // Interface variables (formal arguments)
0646 C integer option    ! special service name (character
0648 C real xin(*)     ! input vector
0649 C real xout(*)    ! output vector
0650 C // Local variables
0654 C character*4 copt    ! special service (character form)
0655 C real colat  ! input colatitude
0656 C //---------------------------------------------
0658 C Common block variables and declaration.
0659 C ALL CODE BETWEEN THE '//' SEPARATORS MUST BE
0661 C DUPLICATED EXACTLY IN EACH NAVIGATION ROUTINE
0662 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 C // Common block contents: projection parameters
0668 C real Lin0         ! image line of pole
0669 C real Ele0        ! image element of pole
0670 C real Scale      ! km per unit image coordinate
0671 C (pixel)
0672 C real Lon0       ! standard longitude
0673 C real Colat0     ! standard colatitude
0674 C // Common block contents: precomputed intermediate values
0675 C real Coscl      ! cosine(Colat0)
0676 C real Tancl      ! tangent(Colat0)
0677 C real Tanc12    ! tangent(Colat0/2)
0678 C real Mxtheta    ! limit of angle from std. lon
0679 C             ! on projection surface
0680 C // Common block contents: constants
0681 C real D2R        ! degrees to radians factor
0682 C real Pi         ! returned when navigation
0683 C             ! cannot be done
0684 C real Erad       ! Earth radius
0685 C logical Init    ! initialized flag
0686 C logical Latlon  ! .TRUE. for lat/lon I/O
COMMON/TANC/ Lin0, Ele0, Scale, Lon0, Colat0, 
& Coscl, Tanc1, Tanc12, Mntheta, 
& D2R, Pi, Badreal, Erad, Init, Lation

C End of common block variables and declaration.

xout(1) = Badreal

// verify initialized module

if(.not.init) then
    NVXOPT = -6
    return
end if

// Extract and interpret the option

call movwc(option,copt)

if(copt.eq.'SCAL') then

    // Compute colatitude and make sure it is
    C // physically possible and navigable

    if ( xin(1).gt.90.0 .or. xin(1).lt.-90.0 ) then
        NVXOPT = -2
        return
    else if ( xin(1).eq.90.0 .or. xin(1).eq.-90.0 ) then
        NVXOPT = -1
        return
    end if

    colat = Pi/2. - D2R*xin(1)

    // Now compute actual scale for this colatitude

    xout(1) = scale
    * * (sin(Colat0) *(tan(colat/2.)/Tanc12)**Coscl)/sin(colat)

    else if(copt.eq.'???')

    // Add code for additional options here

else

    NVXOPT = -5
    return
end if

NVXOPT = 0

return

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 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 applications to view the data that you prescribe.

This section describes:

♦ the calibration API

♦ the design and structure of a calibration module

♦ how to write a calibration module and integrate it into McIDAS

♦ some additional guidelines to use when writing a calibration module
Basic concepts

McIDAS 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 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 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 area format image file, see Chapter 7, 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; the most notable exception is the McIDAS D command. 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 Dynamic Link Library (DLL) to load. For example, the data type for GOES-8 is GVAR. Thus, the name constructed, using slot number 1, is KB1GVAR, which loads the module kb1gvar.dll.

### Designing your calibration module

All calibration modules have the same framework. They must conform to the McIDAS convention for functionality, names of functions, and types of arguments. This standardization allows applications to make use of these modules in a generic, yet powerful way. It is usually not necessary for applications to have any private knowledge of the data 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: **kbxtype.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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kbxini</td>
<td>initializes and verifies the requested calibration</td>
</tr>
<tr>
<td>kbxcal</td>
<td>calibrates the data</td>
</tr>
<tr>
<td>kbxopt</td>
<td>provides additional operations, which are usually queries from the application</td>
</tr>
</tbody>
</table>

The structure of these functions is designed so that any required ancillary
data is passed in as arguments or handled by `kbxopt`. An exception is
the access to the calibration block. Current McIDAS calibration modules
requiring the calibration block read the block from disk from within `kbxcal`.
An alternate memory-based method will be introduced at a later time.

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 calibration modules retrieve necessary parameters from
one of these locations:

- directory block
- calibration block
- line prefix
- McIDAS disk file

Some calibration algorithms, such as VISR (GOES 1-byte data) and WSI
(WSI Radar) 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.
WSI radar images need only 16 values to represent the data, base map and labels. 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, such as Unix or OS/2, 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 *Satellite-specific characteristics* in the area file description in *Chapter 7, 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 7, Format of the Data Files*.

**McIDAS 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 regenerated 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 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 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.

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maaatb</td>
<td>AAA-specific mpixtb</td>
</tr>
<tr>
<td>mavhtb</td>
<td>AVHRR-specific mpixtb</td>
</tr>
<tr>
<td>mvastb</td>
<td>VAS-specific mpixtb</td>
</tr>
<tr>
<td>mgvatb</td>
<td>GVAR-specific mpixtb</td>
</tr>
</tbody>
</table>

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 5, McIDAS 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 command:

Type: CA area TYPE=SIN CTYPE=RAW

After compiling KBXSIN.DLM and any appropriate applications, McIDAS 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 araspec, 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 redare. It takes as input:

- the line prefix
- the area directory
- the number of values to calibrate
- the band, if required
- the 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 D and IMGPROBE commands, which list the stored data values converted to appropriate quantities. 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 CALLTYP 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.

```plaintext
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:
```

Accessing Data

6-104

McIDAS Programmer's Manual

Issued 8/97
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/MOSIN/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: 68: RETURN
69: 70: 900 CONTINUE
71: KBXINI = -1
72: RETURN
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
85: *$ idir - area directory
86: *$ nval - number of values to calibrate
87: *$ iband - band number
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
98: *$
99: *$ Categories:
100: *$ calibration
101: 102: INTEGER PREFIX(*)
103:
INClUDE 'areaparm.inc'

INTEGER JTYPE
INTEGER ISOU
INTEGER IDES
INTEGER JOPT(NUMAREAOPTIONS)
INTEGER ITAB(256)

C--- If the calibration type changes, remake the lookup table
IF( JTYPE .NE. IFLAG ) THEN
    CALL MAKTAB(JTYPE, ITAB)
    IFLAG = JTYPE
ENDIF

C--- Pass the data IBUF through the lookup table ITAB
CALL MPIXTB(NVAL,ISOU,IDES,IBUF,ITAB)

INTEGER FUNCTION KBXOPT(CFUNC,IIN,IOUT)
*$ Name:
  kbxopt - Additional operations
*$ Interface:
      integer function
      kbxopt( character*4 cfunt, integer iin(*), integer iout(*) )
*$ Input:
      cfunt - function ('INFO', 'KEYS')
      iin  - for cfunt 'KEYS', iin contains frame directory block
            for cfunt 'INFO'
      iin(1) - band number
      iin(2) - sensor source number
      iin(3) - calibration type ('GVAR', for example)
*$ Input and Output:
      none
*$ Output:
      iout - for cfunt 'KEYS'
      iout(1) - number of physical quantities,'TEMP' etc.
      iout(2-n) - list of physical quantities
      iout - for cfunt 'INFO'
      iout(1) - number of physical quantities,'TEMP' etc.
      iout(2-n) - list of physical quantities, units, and scale factors
*$ Return values:
      0  - success
      1  - invalid function
*$ Categories:
      calibration
      CHARACTER*4 CFUNC

INTEGER IIN(*)
INTEGER IOUT(*)
INClUDE 'areaparm.inc'
INTEGER JTYPE
INTEGER ISOU
INTEGER IDES
INTEGER JOPT(NUMAREA,OPTIONS)
COMMON/MOSIN/JTYPE,ISOU,IDES,JOPT
C
C--- KEYS option
C
IF ( CFUNC .EQ. 'KEYS') THEN
  IOUT(1) = 3 ! Number of types
  IOUT(2) = LIT('RAW ')
  IOUT(3) = LIT('SIN ')
  IOUT(4) = LIT('BRIT ')
ENDIF

C--- INFO option
C
IF ( CFUNC .EQ. 'INFO') THEN
  IOUT(1) = 3 ! Number of types
  IOUT(2) = LIT('RAW ')
  IOUT(3) = LIT('SIN ')
  IOUT(4) = LIT('BRIT ')
  IOUT(5) = LIT(' ')
  IOUT(6) = LIT('none')
  IOUT(7) = LIT(' ')
  IOUT(8) = 1 ! Scale factors
  IOUT(9) = 1000
  IOUT(10) = 1
ENDIF

RETURN
END

SUBROUTINE MAKTAB(JTYPE, ITAB)

* Name: maktab - Make lookup table for sine modified calibration

* Interface:
  subroutine maktab(integer jtype, integer itab(*))

* Input:
  jtype - calibration type

* Output:
  itab - lookup table of 256 values

* Remarks:
  This routine makes a lookup table by computing the sine
  for all possible values from 0 to 255 (the range of the
  input data). Rather than computing the sine directly on
  the values 0 to 255, it is initially scaled to 0 to 10
  which is approximately 3 sine waves (3 * PI = 10)

* Categories:
  calibration

INTEGER ITAB(*)
REAL SINVAL
REAL X
C
C---
DO 100 I = 1, 256

X = I - 1     ! X goes from 0 to 255
X = 10. * (X / 255.)     ! Normalize and scale to 3 sine waves
SINVAL = SIN(X)

C--- Output sine value

IF (JTYPE .EQ. 1) THEN
  ITAB(I) = NINT(SINVAL * 1000.)     ! Scale sine by 1000
ENDIF

C--- Output grayscale value

ELSE IF (JTYPE .EQ. 2) THEN
  ITAB(I) = NINT(127. + 128 * SINVAL)     ! Scale to 0 to 255
ENDIF

100 CONTINUE

RETURN

END
Format of the Data Files

This chapter describes the formats of the data files developed for applications running under McIDAS-X and -OS2. 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

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREAmmmn</td>
<td>Area (image) files</td>
<td>7-3</td>
</tr>
<tr>
<td>DC*</td>
<td>MD schema definition files</td>
<td>7-43</td>
</tr>
<tr>
<td>*.ET</td>
<td>Enhancement save files</td>
<td>7-46</td>
</tr>
<tr>
<td>FRAMED</td>
<td>Panel configuration file</td>
<td>7-48</td>
</tr>
<tr>
<td>FRAMENH.001</td>
<td>Frame enhancement table</td>
<td>7-49</td>
</tr>
<tr>
<td>FRAMEx,y</td>
<td>Image frame files</td>
<td>7-51</td>
</tr>
<tr>
<td>GMSCAL</td>
<td>GMS calibration file</td>
<td>7-53</td>
</tr>
<tr>
<td>GRIDmnnn</td>
<td>Grid files</td>
<td>7-54</td>
</tr>
<tr>
<td>*.GRX</td>
<td>Graphics save tables for McIDAS-X</td>
<td>7-59</td>
</tr>
<tr>
<td>HIRSCRPF</td>
<td>HIRS calibration reference parameters</td>
<td>7-60</td>
</tr>
<tr>
<td>File</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>HIRSTAUL</td>
<td>HIRS transmittance coefficients</td>
<td>7-62</td>
</tr>
<tr>
<td>IDMSL</td>
<td>Reporting stations list</td>
<td>7-64</td>
</tr>
<tr>
<td>MDXXmmm</td>
<td>MD files</td>
<td>7-66</td>
</tr>
<tr>
<td>MSUSCRPF</td>
<td>MSU calibration reference parameters</td>
<td>7-69</td>
</tr>
<tr>
<td>OUTL*</td>
<td>Base map files</td>
<td>7-71</td>
</tr>
<tr>
<td>SATANNOT</td>
<td>Image annotation description file</td>
<td>7-73</td>
</tr>
<tr>
<td>SKEDFILE</td>
<td>Scheduled McIDAS commands</td>
<td>7-74</td>
</tr>
<tr>
<td>UC</td>
<td>McIDAS User Common</td>
<td>7-76</td>
</tr>
<tr>
<td>VASTBLS</td>
<td>VAS calibration tables</td>
<td>7-84</td>
</tr>
<tr>
<td>VIRTnmm</td>
<td>Virtual graphics files</td>
<td>7-86</td>
</tr>
</tbody>
</table>
AREAAnnnn

Area (image) files, where nnnn is a user-defined number.

In McIDAS, 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 named AREAAnnnn, 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.

Area files consist of these six blocks:

- directory block
- navigation (NAV) block
- calibration (CAL) block
- supplemental block
- data block
- comment (AUDIT) block

Each is described below.

For more information about reading, writing and deleting image data, see the section titled Image data in Chapter 6, 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 Satellite-specific characteristics that follows. All byte offsets and pointers are zero-based. Note that all data shown as yyyyddd are the year and day-of-year. The yyyy values are the actual year modulo 1900.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>relative position of the image object in the ADDE dataset</td>
</tr>
<tr>
<td>2</td>
<td>image type (currently = 4)</td>
</tr>
<tr>
<td>3</td>
<td>SSEC sensor source number; see the Appendices</td>
</tr>
<tr>
<td>4</td>
<td>nominal year and Julian day of the image, yyyyddd</td>
</tr>
<tr>
<td>5</td>
<td>nominal time of the image, hhmmss</td>
</tr>
<tr>
<td>6</td>
<td>upper-left image line coordinate</td>
</tr>
<tr>
<td>7</td>
<td>upper-left image element coordinate</td>
</tr>
<tr>
<td>8</td>
<td>reserved</td>
</tr>
<tr>
<td>9</td>
<td>number of lines in the image</td>
</tr>
<tr>
<td>10</td>
<td>number of data points per line</td>
</tr>
<tr>
<td>11</td>
<td>number of bytes per data point</td>
</tr>
<tr>
<td>12</td>
<td>line resolution</td>
</tr>
<tr>
<td>13</td>
<td>element resolution</td>
</tr>
<tr>
<td>14</td>
<td>number of spectral bands</td>
</tr>
<tr>
<td>15</td>
<td>length of the line prefix</td>
</tr>
<tr>
<td>16</td>
<td>SSEC file creation project number</td>
</tr>
<tr>
<td>17</td>
<td>file creation year and Julian day, yyyyddd</td>
</tr>
<tr>
<td>18</td>
<td>file creation time, hhmmss</td>
</tr>
<tr>
<td>19</td>
<td>spectral band map</td>
</tr>
<tr>
<td>20</td>
<td>image ID number</td>
</tr>
<tr>
<td>21 - 24</td>
<td>reserved for sensor-specific data</td>
</tr>
<tr>
<td>25 - 32</td>
<td>memo field; 32 ASCII characters</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>33</td>
<td>reserved</td>
</tr>
<tr>
<td>34</td>
<td>byte offset to the start of the data block</td>
</tr>
<tr>
<td>35</td>
<td>byte offset to the start of the navigation block</td>
</tr>
<tr>
<td>36</td>
<td>validity code</td>
</tr>
<tr>
<td>37 - 44</td>
<td>PDL (Program Data Load); used for pre-GOES-8 satellites</td>
</tr>
<tr>
<td>45</td>
<td>source of band 8; used for GOES AA processing</td>
</tr>
<tr>
<td>46</td>
<td>actual image start year and Julian day, yyyyddd</td>
</tr>
<tr>
<td>47</td>
<td>actual image start time, hhmmss; in milliseconds for POES data</td>
</tr>
<tr>
<td>48</td>
<td>actual image start scan</td>
</tr>
<tr>
<td>49</td>
<td>length of the prefix documentation</td>
</tr>
<tr>
<td>50</td>
<td>length of the prefix calibration</td>
</tr>
<tr>
<td>51</td>
<td>length of the prefix band list</td>
</tr>
<tr>
<td>52</td>
<td>source type; satellite-specific (ASCII)</td>
</tr>
<tr>
<td>53</td>
<td>calibration type; satellite-specific (ASCII)</td>
</tr>
<tr>
<td>54 - 59</td>
<td>reserved</td>
</tr>
<tr>
<td>60</td>
<td>byte offset to the supplemental block</td>
</tr>
<tr>
<td>61</td>
<td>number of bytes in the supplemental block</td>
</tr>
<tr>
<td>62</td>
<td>reserved</td>
</tr>
<tr>
<td>63</td>
<td>byte offset to the start of the calibration block</td>
</tr>
<tr>
<td>64</td>
<td>number of comment cards</td>
</tr>
</tbody>
</table>

For more information about image data, see the section titled *Image data* in *Chapter 6, Accessing Data*. For more information about coordinate systems, see the section titled *Coordinate systems 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 *Satellite-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 Satellite-specific characteristics that follows.

Supplemental block

The supplemental block may contain information describing or explaining the image data. Word 60 of the directory block contains the byte offset to the start of the supplemental block. Word 61 contains the total number of bytes 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:

\[
\text{Image Line} = \text{UpperLeftLine} + (\text{File Line} \times \text{LineResolution})
\]
\[
\text{Image Element} = \text{UpperLeftEle} + (\text{File Element} \times \text{ElementResolution})
\]

\text{UpperLeftLine} is the line coordinate of the first image line and \text{UpperLeftEle} is the element coordinate of the first image element.
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.

```
<table>
<thead>
<tr>
<th>line prefix 1 line data 1 line prefix 2 line data 2 etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 byte numbers increase &gt;&gt;</td>
</tr>
</tbody>
</table>
```

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.

```
<table>
<thead>
<tr>
<th>validity code documentation calibration band list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 byte numbers increase &gt;&gt;</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Region</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validity code</td>
<td>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.</td>
</tr>
<tr>
<td>documentation</td>
<td>Holds the documentation specific to each satellite. Word 49 of the directory block defines the length of the prefix documentation.</td>
</tr>
<tr>
<td>calibration</td>
<td>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.</td>
</tr>
<tr>
<td>band list</td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

\[
\begin{align*}
\text{line prefix length} &= \text{doc} + \text{cal} + \text{band} + 4 \text{ (if valcode is present)} \\
\text{line data length} &= \text{nbands} \times \text{nele} \times \text{ nbytes} \\
\text{line length} &= \text{line prefix length} + \text{line data length} \\
\text{data block length} &= \text{nlines} \times \text{line length}
\end{align*}
\]

The parameters used in these formulas are defined in the directory block and provided in the table below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Directory block word</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>valcode</td>
<td>36</td>
<td>length of the prefix validity code; if nonzero, the length is four bytes; otherwise it is zero</td>
</tr>
<tr>
<td>doc</td>
<td>49</td>
<td>length of the prefix documentation</td>
</tr>
<tr>
<td>cal</td>
<td>50</td>
<td>length of the prefix calibration</td>
</tr>
<tr>
<td>band</td>
<td>51</td>
<td>length of the prefix band list</td>
</tr>
<tr>
<td>nbands</td>
<td>14</td>
<td>number of bands per line</td>
</tr>
<tr>
<td>nele</td>
<td>10</td>
<td>number of data points per line</td>
</tr>
<tr>
<td>nbytes</td>
<td>11</td>
<td>number of bytes per band</td>
</tr>
<tr>
<td>nlines</td>
<td>9</td>
<td>number of lines in the image</td>
</tr>
</tbody>
</table>

**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.
Satellite-specific characteristics

Some aspects of McIDAS area files are satellite-specific. This section describes characteristics specific to the following satellites:

- Meteosat PDUS
- GVAR Imager
- GVAR Block 11
- GVAR Sounder
- GOES VISSR
- GOES-7
- AVHRR
- TIP
- GMS
- DMSP

Although the descriptions for each satellite 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.

For more information on Meteosat labels and headers, see the EUMETSAT document *Meteosat High Resolution Image Dissemination.*
**Meteosat PDUS directory block**

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1</td>
<td>each band is stored separately</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>for visible image band map</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>(eighth bit from right) for IR band map</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>(tenth bit from right) for WV band map</td>
</tr>
<tr>
<td>22</td>
<td>.xxxx</td>
<td>MIEC absolute calibration band value (IR or WV) from the calibration section of the Meteosat header; stored as scaled integer .xxxx</td>
</tr>
<tr>
<td>23</td>
<td>xx.x</td>
<td>space count corresponding to the calibration value from the calibration section of the Meteosat header; stored as scaled integer xxx</td>
</tr>
<tr>
<td>24</td>
<td>1 or 2</td>
<td>physical sensor number from the Meteosat header</td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>line offset of the southeast corner of the area in image coordinates; 16-bit value from the Meteosat header, right justified plus 1</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>element offset of the southeast corner of the area in image coordinates; 16-bit value from the Meteosat header, right justified plus 1</td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>satellite center longitude of rectification; 16-bit value from the Meteosat header and right justified</td>
</tr>
<tr>
<td>44</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>24</td>
<td>length of the data block line prefix documentation, in bytes</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>length of the data block line prefix calibration, in bytes</td>
</tr>
<tr>
<td>51</td>
<td>0</td>
<td>length of the data block line prefix band list, in bytes</td>
</tr>
<tr>
<td>52</td>
<td>MSAT</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>
<tr>
<td>54</td>
<td>0</td>
<td>data was ingested as sent (full resolution)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>data was sampled down (every other pixel); VIS is sent as resolution 1 in some images and resolution 2 in others</td>
</tr>
<tr>
<td>55</td>
<td></td>
<td>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</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>Region</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validity code</td>
<td>optional but recommended to flag missing data, which must have zeros as placeholder data or a validity code that does not match the value in word 36 of the directory block</td>
</tr>
<tr>
<td>documentation</td>
<td>24 bytes; this is a copy of the label that arrives with each subframe</td>
</tr>
<tr>
<td>calibration</td>
<td>0 bytes (not used)</td>
</tr>
<tr>
<td>band list</td>
<td>0 bytes (not used) since each band is stored in a separate area</td>
</tr>
</tbody>
</table>

**Meteosat PDUS navigation block**

A PDUS navigation block is divided into 256 words.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MSAT</td>
<td>navigation type</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Julian day of the navigation, yyddd</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>time of the navigation, hhmmss</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>reference position for the telescope</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>line number of the telescope reference position</td>
</tr>
<tr>
<td>6</td>
<td>1250</td>
<td>center scan line</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>center longitude of rectification (west positive), ddmmss</td>
</tr>
<tr>
<td>8 - 9</td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Julian day of the navigation, yyddd</td>
</tr>
<tr>
<td>11 - 256</td>
<td>0</td>
<td>reserved</td>
</tr>
</tbody>
</table>

**Meteosat PDUS calibration block**

No calibration block is needed for PDUS; the calibration information is stored in the directory block.
**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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
<td>line resolution</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>element resolution</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>number of bands</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>band map</td>
</tr>
<tr>
<td>25 - 32</td>
<td>RT IMGR DOC</td>
<td>normal entry; ASCII</td>
</tr>
<tr>
<td>49</td>
<td>44</td>
<td>length of the data block line prefix documentation, in bytes</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>length of the data block line prefix calibration, in bytes</td>
</tr>
<tr>
<td>51</td>
<td>0</td>
<td>length of the data block line prefix band list, in bytes</td>
</tr>
<tr>
<td>52</td>
<td>GVAR</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>
</tbody>
</table>
The line prefix’s validity code for Block 0 is four bytes. Its documentation region is 44 bytes, consisting of the following:

<table>
<thead>
<tr>
<th>Documentation region</th>
<th>Bytes</th>
<th>OGE table</th>
</tr>
</thead>
<tbody>
<tr>
<td>block header CRC</td>
<td>2</td>
<td>3-5</td>
</tr>
<tr>
<td>scan status</td>
<td>4</td>
<td>3-6</td>
</tr>
<tr>
<td>year, day and time from Block 0</td>
<td>8</td>
<td>3-6</td>
</tr>
<tr>
<td>block header</td>
<td>30</td>
<td>3-5</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2</td>
<td>number of bytes per band</td>
</tr>
<tr>
<td>19</td>
<td>1 for VIS</td>
<td>band map; one bit should be set for each band in the area</td>
</tr>
<tr>
<td></td>
<td>2, 4, 8 or 16 for IR</td>
<td></td>
</tr>
<tr>
<td>25 - 32</td>
<td>RT IMGR IR</td>
<td>normal entry; ASCII visible band multibanded</td>
</tr>
<tr>
<td></td>
<td>RT IMGR VIS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RT IMGR</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>226</td>
<td>length of the data block line prefix documentation if single band, otherwise a higher value; in bytes</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>length of the data block line prefix calibration, in bytes</td>
</tr>
<tr>
<td>51</td>
<td>0</td>
<td>length of the data block line prefix band list, in bytes</td>
</tr>
<tr>
<td>52</td>
<td>GVAR</td>
<td>image source type; 4 bytes ASCII</td>
</tr>
<tr>
<td>Word</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>53</td>
<td>RAW</td>
<td>calibration type; 4 bytes ASCII</td>
</tr>
<tr>
<td>55</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**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 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.

```
10 | x | x | x | x | x | x | x | x | x | x | x | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1
```

The sensor data’s line prefix contains a 4-byte validity code and a 76-byte documentation region consisting of the following:

<table>
<thead>
<tr>
<th>Documentation region</th>
<th>Bytes</th>
<th>OGE table</th>
</tr>
</thead>
<tbody>
<tr>
<td>block header CRC (the last three bits only); a bit is set if the block header copy is good; this data is usually 00,07</td>
<td>2</td>
<td>2-6</td>
</tr>
<tr>
<td>scan status</td>
<td>4</td>
<td>3-6</td>
</tr>
<tr>
<td>year, day and time from Block 0</td>
<td>8</td>
<td>3-5</td>
</tr>
<tr>
<td>block header</td>
<td>30</td>
<td>3-5</td>
</tr>
<tr>
<td>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</td>
<td>32</td>
<td>3-7</td>
</tr>
<tr>
<td>block zero record, first 150 words</td>
<td>150</td>
<td>3-6</td>
</tr>
</tbody>
</table>

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 two's-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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GVAR</td>
<td>navigation type; 4 bytes ASCII</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>ASCII string, usually a letter followed by three integers</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>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</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>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</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>6 - 62</td>
<td></td>
<td>see OGE Table 3-6, bytes 295 - 522</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>reference longitude, rad*10000000</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>reference distance from nominal, km*100000000</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>reference latitude, rad*10000000</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>reference yaw, rad*10000000</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>reference attitude roll, rad*10000000</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>reference attitude pitch, rad*10000000</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>reference attitude yaw, rad*10000000</td>
</tr>
<tr>
<td>13 - 14</td>
<td></td>
<td>epoch date/time, BCD format</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>delta from epoch time, minutes*100</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>image motion compensation roll, rad*10000000</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>image motion compensation pitch, rad*10000000</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>image motion compensation yaw, rad*10000000</td>
</tr>
<tr>
<td>19 - 31</td>
<td></td>
<td>longitude delta from reference values, rad*10000000</td>
</tr>
<tr>
<td>32 - 42</td>
<td></td>
<td>radial distance delta from reference values, km*100000000</td>
</tr>
<tr>
<td>43 - 51</td>
<td></td>
<td>sine of the geocentric latitude delta values, units*10000000</td>
</tr>
<tr>
<td>52 - 60</td>
<td></td>
<td>sine of the orbit yaw delta values, units*10000000</td>
</tr>
<tr>
<td>61</td>
<td></td>
<td>daily solar rate, rad/min*10000000</td>
</tr>
<tr>
<td>62</td>
<td></td>
<td>exponential start time from epoch, minutes*100</td>
</tr>
<tr>
<td>Word</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>63 - 117</td>
<td>roll attitude angle (OGE Table 3-6, bytes 523-742)</td>
<td>exponential magnitude, rad*10000000</td>
</tr>
<tr>
<td>63</td>
<td>exponential magnitude, rad*10000000</td>
<td>exponential time constant, minutes*100</td>
</tr>
<tr>
<td>65</td>
<td>mean attitude angle, rad*10000000</td>
<td>number of sinusoids/angles, no units</td>
</tr>
<tr>
<td>67</td>
<td>magnitude of first order sinusoid, rad*10000000</td>
<td>phase angle of first order sinusoid, rad*10000000</td>
</tr>
<tr>
<td>95</td>
<td>magnitude of fifteenth sinusoid, rad*10000000</td>
<td>phase angle of fifteenth sinusoid, rad*10000000</td>
</tr>
<tr>
<td>97</td>
<td>number of monomial sinusoid, no units</td>
<td>order of applicable sinusoid, no units</td>
</tr>
<tr>
<td>99</td>
<td>order of first monomial sinusoid, no units</td>
<td>magnitude of monomial sinusoid, rad*10000000</td>
</tr>
<tr>
<td>100</td>
<td>phase angle of monomial sinusoid, rad*10000000</td>
<td>angle from epoch at daily solar rate, rad*10000000</td>
</tr>
<tr>
<td>102</td>
<td>repeat of words 98-102 for second monomial</td>
<td>repeat of words 98-102 for third monomial</td>
</tr>
<tr>
<td>113-117</td>
<td>repeat of words 98-102 for fourth monomial</td>
<td>reserved</td>
</tr>
<tr>
<td>128</td>
<td>MORE</td>
<td>4 bytes ASCII</td>
</tr>
<tr>
<td>129</td>
<td>GVAR</td>
<td>4 bytes ASCII</td>
</tr>
<tr>
<td>130-239</td>
<td>attitude angles</td>
<td>repeat of Words 63-117 for pitch attitude angle; see OGE Table 3-6, bytes 743-962</td>
</tr>
<tr>
<td>185-239</td>
<td>repeat of Words 63-117 for yaw attitude angle; see OGE Table 3-6, bytes 963-1182</td>
<td>reserved</td>
</tr>
<tr>
<td>240-255</td>
<td>MORE</td>
<td>4 bytes ASCII</td>
</tr>
<tr>
<td>256</td>
<td>GVAR</td>
<td>4 bytes ASCII</td>
</tr>
<tr>
<td>258-367</td>
<td>misalignment angles</td>
<td>repeat of Words 63-117 for roll misalignment angle; see OGE Table 3-6, bytes 1183-1402</td>
</tr>
<tr>
<td>313-367</td>
<td>repeat of Words 63-117 for pitch misalignment angle; see OGE Table 3-6, bytes 1403-1622</td>
<td>year and Julian day, yyyddd</td>
</tr>
<tr>
<td>368</td>
<td>nominal start time of the image; comes from Block 0 when navigation data is taken from Block 0, HHMMSSmmm</td>
<td>imager/sounder instrument flag; 1 = imager, 2 = sounder</td>
</tr>
<tr>
<td>370</td>
<td>reserved</td>
<td>instrument nadir, north/south cycles; see OGE Table 3-6, byte 6305</td>
</tr>
<tr>
<td>380</td>
<td>instrument nadir, north/south cycles; see OGE Table 3-6, byte 6305</td>
<td>instrument nadir, east/west cycles; see OGE Table 3-6, byte 6306</td>
</tr>
<tr>
<td>381</td>
<td>instrument nadir, east/west cycles; see OGE Table 3-6, byte 6306</td>
<td>instrument nadir, north/south increments; see OGE Table 3-6, byte 6307-6308</td>
</tr>
<tr>
<td>382</td>
<td>instrument nadir, north/south increments; see OGE Table 3-6, byte 6307-6308</td>
<td>instrument nadir, east/west increments; see OGE Table 3-6, byte 6309-6310</td>
</tr>
<tr>
<td>Word</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>384</td>
<td>MORE</td>
<td>4 bytes ASCII</td>
</tr>
<tr>
<td>385</td>
<td>CVAR</td>
<td>4 bytes ASCII</td>
</tr>
<tr>
<td>386-511</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>512</td>
<td>MORE</td>
<td>4 bytes ASCII</td>
</tr>
<tr>
<td>513</td>
<td>CVAR</td>
<td>4 bytes ASCII</td>
</tr>
<tr>
<td>514-640</td>
<td></td>
<td>reserved</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 8</td>
<td></td>
<td>visible bias coefficients; one per detector (OGE Table 3-6, bytes 6399-6430)</td>
</tr>
<tr>
<td>9 - 16</td>
<td></td>
<td>visible first order gain coefficients; one per detector (OGE Table 3-6, bytes 6431-6462)</td>
</tr>
<tr>
<td>17 - 24</td>
<td></td>
<td>visible second order gain coefficients; one per detector (OGE Table 3-6, bytes 6463-6494)</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>visible radiance to albedo conversion factor (OGE Table 3-6, bytes 6495-6498)</td>
</tr>
<tr>
<td>26 - 29</td>
<td></td>
<td>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)</td>
</tr>
<tr>
<td>30 - 33</td>
<td></td>
<td>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)</td>
</tr>
<tr>
<td>34 - 37</td>
<td></td>
<td>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)</td>
</tr>
<tr>
<td>38 - 41</td>
<td></td>
<td>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)</td>
</tr>
<tr>
<td>42 - 128</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
**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**

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1 or 2</td>
<td>number of bytes per element, depending on the element size of the band; a holding area cannot contain both 1- and 2-byte data</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>number of bands in the image</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>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 0Ce600 hex with bits set in positions 20, 19, 11 and 10</td>
</tr>
<tr>
<td>25 - 32</td>
<td>RT BK11 BYT1</td>
<td>normal entry; ASCII</td>
</tr>
<tr>
<td>49</td>
<td>40</td>
<td>length of the data block line prefix documentation, in bytes</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>length of the data block line prefix calibration, in bytes</td>
</tr>
<tr>
<td>51</td>
<td>0</td>
<td>length of the data block line prefix band list, in bytes</td>
</tr>
<tr>
<td>52</td>
<td>BK11</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>
</tbody>
</table>

**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 |
```
The 8-bit data is formatted as follows:

```
| x | x | x | x | x | x | x |
```

The 6-bit data is formatted as follows:

```
| 0 | 0 | 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:

<table>
<thead>
<tr>
<th>Documentation region</th>
<th>Bytes</th>
<th>OGE table</th>
</tr>
</thead>
<tbody>
<tr>
<td>block header CRC; this field is overwritten in the mainframe by a 2-byte counter and is used to check sequencing of the data flow</td>
<td>2</td>
<td>3-5</td>
</tr>
<tr>
<td>year, day and time from Block 0</td>
<td>8</td>
<td>3-6</td>
</tr>
<tr>
<td>block header</td>
<td>30</td>
<td>3-5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>10</td>
<td>line resolution; if lines are sampled or averaged, the resolution is a multiple of 10</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>element resolution; if pixels are sampled or averaged, the resolution is in multiples of 10</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>number of bands</td>
</tr>
<tr>
<td>19</td>
<td>524287</td>
<td>band filter map; translates to 0007ffff hex; a bit is set for each of bands 1 - 19</td>
</tr>
<tr>
<td>25 - 32</td>
<td>Priority Completed</td>
<td>normal entry; ASCII</td>
</tr>
<tr>
<td>49</td>
<td>36</td>
<td>length of the data block line prefix documentation, in bytes</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>length of the data block line prefix calibration, in bytes</td>
</tr>
<tr>
<td>51</td>
<td>up to 24 in multiples of four</td>
<td>length of the data block line prefix band list, in bytes; see the section below for more information</td>
</tr>
<tr>
<td>52</td>
<td>GVAR</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>
</tbody>
</table>

### 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 bands 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 least 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 sensors are A, B, C and D. Each data block line contains information from only one sensor. The first line contains sensor A information, the second line, sensor B, etc. This pattern is repeated for the entire data block.

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.

<table>
<thead>
<tr>
<th>Documentation region</th>
<th>Bytes</th>
<th>OGE table</th>
</tr>
</thead>
<tbody>
<tr>
<td>first nine words of the SAD Block ID</td>
<td>9</td>
<td>3-8</td>
</tr>
<tr>
<td>year, day and time of the scan line start</td>
<td>8</td>
<td>3-11</td>
</tr>
<tr>
<td>scan status</td>
<td>2</td>
<td>3-11</td>
</tr>
<tr>
<td>number of blocks in the scan</td>
<td>2</td>
<td>3-11</td>
</tr>
<tr>
<td>O&amp;A location (not used)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>detector status</td>
<td>10</td>
<td>3-11</td>
</tr>
<tr>
<td>detector A, B, C or D used in this area line</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>padding</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>band list indicating the band order for this line; see the section below for more information</td>
<td>up to 24 bytes, in multiples of four</td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>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 3-6)</td>
</tr>
<tr>
<td>6 - 62</td>
<td>(OGE Table 3-11, bytes 323 - 550)</td>
<td></td>
</tr>
<tr>
<td>63 - 117</td>
<td>roll attitude angle (OGE Table 3-11, bytes 551-770)</td>
<td></td>
</tr>
<tr>
<td>130-184</td>
<td>repeat of Words 63-117 for pitch attitude angle (OGE Table 3-11, bytes 771-990)</td>
<td></td>
</tr>
<tr>
<td>185-239</td>
<td>repeat of Words 63-117 for yaw attitude angle (OGE Table 3-11, bytes 991-1210)</td>
<td></td>
</tr>
<tr>
<td>258-312</td>
<td>repeat of Words 63-117 for roll misalignment angle (OGE Table 3-11, bytes 1211-1430)</td>
<td></td>
</tr>
<tr>
<td>313-367</td>
<td>repeat of Words 63-117 for pitch misalignment angle (OGE Table 3-11, bytes 1431-1650)</td>
<td></td>
</tr>
<tr>
<td>370</td>
<td>image/sounder instrument flag; 1 = imager, 2 = sounder</td>
<td></td>
</tr>
<tr>
<td>380</td>
<td>instrument nadir, north/south cycles (OGE Table 3-6, byte 3005)</td>
<td></td>
</tr>
<tr>
<td>381</td>
<td>instrument nadir, east/west cycles (OGE Table 3-6, byte 3006)</td>
<td></td>
</tr>
<tr>
<td>382</td>
<td>instrument nadir, north/south increments (OGE Table 3-6, bytes 3007-3008)</td>
<td></td>
</tr>
<tr>
<td>383</td>
<td>instrument nadir, east/west increments (OGE Table 3-6, bytes 3009-3010)</td>
<td></td>
</tr>
</tbody>
</table>

**GVAR Sounder calibration block**

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 4</td>
<td>visible bias coefficients; one per detector (OGE Table 3-11, bytes 3075-3090)</td>
<td></td>
</tr>
<tr>
<td>5 - 8</td>
<td>visible first order gain coefficients; one per detector (OGE Table 3-11, bytes 3091-3106)</td>
<td></td>
</tr>
<tr>
<td>9 - 12</td>
<td>visible second order gain coefficients; one per detector (OGE Table 3-11, bytes 3107-3122)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>visible radiance to albedo conversion factor (OGE Table 3-11, bytes 3123-3126)</td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>visible radiance to albedo conversion factor (OGS Table 3-11, bytes 3123-3126)</td>
</tr>
<tr>
<td>14 - 31</td>
<td></td>
<td>IR bias scaling factors; one per IR channel (OGS Table 3-11, bytes 3991-4278); all channels contain the same values for each detector</td>
</tr>
<tr>
<td>32 - 49</td>
<td></td>
<td>IR gain scaling factors; one per IR channel (OGS Table 3-11, bytes 4279-4566); all channels contain the same values for each detector</td>
</tr>
<tr>
<td>50 - 128</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**GOES VISSR**

The image source type VISSR 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 VISSR 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.

\[
T = 418 - B \quad \text{where } B > 176 \text{ or } B = 176
\]

\[
T = 330 - (B / 2) \quad \text{where } B < 176 \text{ or } B = 176
\]

The line prefix in a VISSR 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

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td></td>
<td>number of spectral bands</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>band filter map; a bit is set to one for each band appearing in the area</td>
</tr>
<tr>
<td>52</td>
<td>AAA</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>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>day, yyddd</td>
</tr>
<tr>
<td>4</td>
<td>time of the scan, hhmmss</td>
</tr>
<tr>
<td>4</td>
<td>scan number (satellite coordinate line number)</td>
</tr>
<tr>
<td>104</td>
<td>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</td>
</tr>
</tbody>
</table>
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 \( j \)th byte of the band list corresponds to the \( j\text{th} \) 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.

<table>
<thead>
<tr>
<th>Channel</th>
<th>Detector</th>
<th>Size</th>
<th>Location</th>
<th>Spectral band</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HGCDET</td>
<td>large</td>
<td>upper</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>HGCDET</td>
<td>large</td>
<td>upper</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>HGCDET</td>
<td>large</td>
<td>upper</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>HGCDET</td>
<td>large</td>
<td>upper</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>HGCDET</td>
<td>large</td>
<td>upper</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>INSB</td>
<td>large</td>
<td>upper</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>HGCDET</td>
<td>large</td>
<td>upper</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>HGCDET</td>
<td>large</td>
<td>upper</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>HGCDET</td>
<td>large</td>
<td>upper</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>HGCDET</td>
<td>large</td>
<td>upper</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>INSB</td>
<td>large</td>
<td>upper</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>INSB</td>
<td>large</td>
<td>upper</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>HGCDET</td>
<td>large</td>
<td>lower</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>HGCDET</td>
<td>large</td>
<td>lower</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>HGCDET</td>
<td>large</td>
<td>lower</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>HGCDET</td>
<td>large</td>
<td>lower</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>HGCDET</td>
<td>large</td>
<td>lower</td>
<td>5</td>
</tr>
<tr>
<td>Channel</td>
<td>Detector</td>
<td>Size</td>
<td>Location</td>
<td>Spectral band</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>------</td>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>18</td>
<td>INSB</td>
<td>large</td>
<td>lower</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td>HGCDE</td>
<td>large</td>
<td>lower</td>
<td>7</td>
</tr>
<tr>
<td>20</td>
<td>HGCDE</td>
<td>large</td>
<td>lower</td>
<td>8</td>
</tr>
<tr>
<td>21</td>
<td>HGCDE</td>
<td>large</td>
<td>lower</td>
<td>9</td>
</tr>
<tr>
<td>22</td>
<td>HGCDE</td>
<td>large</td>
<td>lower</td>
<td>10</td>
</tr>
<tr>
<td>23</td>
<td>INSB</td>
<td>large</td>
<td>lower</td>
<td>11</td>
</tr>
<tr>
<td>24</td>
<td>INSB</td>
<td>large</td>
<td>lower</td>
<td>12</td>
</tr>
<tr>
<td>25</td>
<td>HGCDE</td>
<td>small</td>
<td>upper</td>
<td>3</td>
</tr>
<tr>
<td>26</td>
<td>HGCDE</td>
<td>small</td>
<td>upper</td>
<td>4</td>
</tr>
<tr>
<td>27</td>
<td>HGCDE</td>
<td>small</td>
<td>upper</td>
<td>5</td>
</tr>
<tr>
<td>28</td>
<td>HGCDE</td>
<td>small</td>
<td>upper</td>
<td>7</td>
</tr>
<tr>
<td>29</td>
<td>HGCDE</td>
<td>small</td>
<td>upper</td>
<td>8</td>
</tr>
<tr>
<td>30</td>
<td>HGCDE</td>
<td>small</td>
<td>upper</td>
<td>9</td>
</tr>
<tr>
<td>31</td>
<td>HGCDE</td>
<td>small</td>
<td>upper</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>HGCDE</td>
<td>small</td>
<td>lower</td>
<td>3</td>
</tr>
<tr>
<td>33</td>
<td>HGCDE</td>
<td>small</td>
<td>lower</td>
<td>4</td>
</tr>
<tr>
<td>34</td>
<td>HGCDE</td>
<td>small</td>
<td>lower</td>
<td>5</td>
</tr>
<tr>
<td>35</td>
<td>HGCDE</td>
<td>small</td>
<td>lower</td>
<td>7</td>
</tr>
<tr>
<td>36</td>
<td>HGCDE</td>
<td>small</td>
<td>lower</td>
<td>8</td>
</tr>
<tr>
<td>37</td>
<td>HGCDE</td>
<td>small</td>
<td>lower</td>
<td>9</td>
</tr>
<tr>
<td>38</td>
<td>HGCDE</td>
<td>small</td>
<td>lower</td>
<td>10</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GOES</td>
<td>navigation type (ASCII characters)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>satellite ID, year, and Julian day, ssyyddd</td>
</tr>
<tr>
<td>Word</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>nominal start time of the image, hhmmss</td>
</tr>
</tbody>
</table>
| 4 - 12 | 1 | orbit parameters  
|       | | orbit type  
| 4    |       | epoch date (ETIMY), yyyyddd  
| 5    |       | epoch time (ETIMH), hhmmss  
| 6    |       | semimajor axis (SEMIMA), km * 100  
| 7    |       | orbital eccentricity (ECCEN) * 1000000 (unitless)  
| 8    |       | orbital inclination (ORBINC), deg * 1000  
| 9    |       | mean anomaly (MEANA), deg * 1000  
| 10   |       | argument of perigee (PERIGEE), deg * 1000  
| 11   |       | right ascension of ascending node (ASNODE), deg * 1000 |
| 13 - 15 | | attitude parameters  
| 13   |       | declination of satellite axis (DECLIN), dddmmss (+ = north)  
| 14   |       | right ascension of satellite axis (RASCN), dddmmss  
| 15   |       | picture center line number (PICLIN) |
| 16   |       | spin period (SPINP); the satellite period, in microseconds, or the spin rate in revolutions/minute |
| 17 - 20 | | frame geometry  
| 17   |       | total sweep angle, line direction (DEGLN), dddmmss  
| 18   |       | number of scan lines (LINTOT), nnnnnnn where nn is the number of sensors and nnnn is the number of scans; total number of lines is nn * nnnn  
| 19   |       | total sweep angle, element direction (DGELE), dddmmss  
| 20   |       | number of elements in a scan line (ELETOT) |
| 21 - 30 | | camera geometry  
| 21   |       | forward-leaning (PITCH), dddmmss  
| 22   |       | sideways-leaning (YAW), dddmmss  
| 23   |       | rotation (ROLL), dddmmss  
| 24   |       | reserved  
| 25   |       | east/west adjustment (IAJUST) in visible elements (+ or -)  
| 26   |       | time computed by IAJUST from the first valid landmark of the day (IAJTIM), hhmmss  
| 27   |       | reserved  
| 28   |       | angle between VISSR and sun sensor (ISEANG), dddmmss  
| 29   |       | reserved for later implementation of *SKEW*  
| 30   | 0     | reserved |
| 31 - 38 | | betas for this area  
| 31   |       | scan line of the first beta  
| 32   |       | time of the first beta scan line (beginning), hhmmss  
| 33   |       | time of the first beta scan line (continued), milliseconds*10  
| 34   |       | beta count 1  
| 35   |       | scan line of the second beta  
| 36   |       | time of the second beta scan line (beginning), hhmmss  
| 37   |       | time of the second beta scan line (cont.), milliseconds*10  
<p>| 38   |       | beta count 2 |</p>
<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>39-128</td>
<td>39</td>
<td>gammas for this area</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gamma, element offset * 100; this is the nominal offset at time zero of this day</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>gamma-dot, element drift per hour * 100</td>
</tr>
<tr>
<td>41-120</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>121-128</td>
<td></td>
<td>memo; up to 32 ASCII characters of comments</td>
</tr>
</tbody>
</table>

**GOES-7 AAA calibration block**

The calibration block is composed of the following data.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>sensor source number</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>date, yyddd</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>time, hhmmss</td>
</tr>
<tr>
<td>4 - 9</td>
<td></td>
<td>radiance equation coefficients, array IAB(2,38)</td>
</tr>
<tr>
<td>80 - 117</td>
<td></td>
<td>radiance equation coefficients scale factors, IFAB(38)</td>
</tr>
<tr>
<td>118-128</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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:

\[
\begin{align*}
AB1 &= IAB(1, ICHAN) \\
AB2 &= IAB(2, ICHAN) \\
FAB &= 2.***(15 - IFAB(ICHAN)) \\
R &= (AB2 * P/32. - AB1) / FAB
\end{align*}
\]

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 users 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

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>5</td>
<td>all five bands are normally stored in one file</td>
</tr>
<tr>
<td>15</td>
<td>244 or 284</td>
<td>line prefix length in bytes. (TIRO = 244, AVHR = 284)</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>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</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>POES signal type (HRPT, LAC, GAC)</td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>line prefix validity (VAL) code. (length = 4)</td>
</tr>
<tr>
<td>47</td>
<td></td>
<td>time in milliseconds</td>
</tr>
<tr>
<td>49</td>
<td>192</td>
<td>length of the line prefix documentation</td>
</tr>
<tr>
<td>50</td>
<td>40 or 80</td>
<td>line prefix calibration length (40 for TIRO, 80 for AVHR)</td>
</tr>
<tr>
<td>51</td>
<td>8</td>
<td>length of the level (LEV) map section</td>
</tr>
<tr>
<td>52</td>
<td>TIRO/AVHRR</td>
<td>image source type</td>
</tr>
<tr>
<td>53</td>
<td>RAW</td>
<td>pixel type stored in the area file</td>
</tr>
<tr>
<td>54</td>
<td>0, 1</td>
<td>sampling/averaging indicator (0 = average, 1 = sample)</td>
</tr>
<tr>
<td>55</td>
<td>1, 2, 3</td>
<td>signal type indicator (1 = LAC, 2 = GAC, 3 = HRPT)</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td>orbit position (ascending node, descending node, equatorial pass)</td>
</tr>
</tbody>
</table>
**AVHRR data block**

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 |
```

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.

The line prefix for the data block contains the information below.

<table>
<thead>
<tr>
<th>Region</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validity code</td>
<td>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</td>
</tr>
<tr>
<td>documentation</td>
<td>192-byte DOC section from the signal transmission</td>
</tr>
<tr>
<td>calibration</td>
<td>40 bytes of zeros; filled during post-processing in McIDAS</td>
</tr>
<tr>
<td>band list</td>
<td>8 bytes; values 1 through 5 (left to right), with three pad zeros in successive bytes, indicate the order the bands will appear in the subsequent data section</td>
</tr>
</tbody>
</table>

For a complete description of the line documentation fields, see NASA Technical Memorandum NESS 107, 1988.

**AVHRR navigation block**

The AVHRR navigation block is divided into 128 words.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TIRO</td>
<td>navigation type; 4 bytes ASCII</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>sensor source, year and Julian day of the navigation, ssyymdd</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>time of the navigation, hhmms</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>orbit type</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>epoch date, yymmd</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>epoch time, hhmms</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>semi-major axis, km * 100</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>orbital eccentricity, *1000000</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>orbital inclination, degrees * 1000</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>mean anomaly, degrees * 1000</td>
</tr>
<tr>
<td>Word</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>argument of perigee, degrees * 1000</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>right ascension of the ascending node, degrees * 1000</td>
</tr>
<tr>
<td>13</td>
<td>2048</td>
<td>number of samples per line</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>angular increment between samples, degrees * 1000</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>fraction of a second in epoch time</td>
</tr>
<tr>
<td>16-45</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>46</td>
<td>-1</td>
<td>satellite is in a descending pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>satellite is in an ascending pass</td>
</tr>
<tr>
<td>47</td>
<td></td>
<td>image coordinates of the first line to navigate</td>
</tr>
<tr>
<td>48</td>
<td></td>
<td>time at the start of the first line, milliseconds from start-of-day</td>
</tr>
<tr>
<td>49</td>
<td></td>
<td>time interval between lines, in milliseconds</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>image is displayed normally</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>image is inverted</td>
</tr>
<tr>
<td>51</td>
<td></td>
<td>number of lines in the inverted image</td>
</tr>
<tr>
<td>52</td>
<td></td>
<td>number of elements in the inverted image</td>
</tr>
<tr>
<td>53</td>
<td></td>
<td>time interval between lines, in microseconds (preferred over word 49)</td>
</tr>
<tr>
<td>54</td>
<td></td>
<td>time interval between individual data points * 100000000</td>
</tr>
<tr>
<td>55-120</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>121-128</td>
<td></td>
<td>comments; up to 32 characters</td>
</tr>
</tbody>
</table>

**AVHRR Calibration**

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 McIDAS area structure between the NOAA-14 areas and the NOAA-15 areas. Refer to Appendix D, *POES AVHRR Calibration Information* for additional information.

The calibration block is composed of the following data:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>VAL</strong></td>
<td>Validity code</td>
</tr>
<tr>
<td>4</td>
<td><strong>DOC</strong></td>
<td>Words 7-102 of HRPT minor frame (all left shifted 5 bits into a 2-byte sample)</td>
</tr>
<tr>
<td>196</td>
<td><strong>CAL</strong></td>
<td>Band 1 Slope/Gain 1 (all slopes and intercepts in the CAL block are scaled by 1000)</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>Band 1 Intercept/Offset 1</td>
</tr>
<tr>
<td>204</td>
<td></td>
<td>Band 1 Slope/Gain 2</td>
</tr>
<tr>
<td>Byte</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>208</td>
<td></td>
<td>Band 1 Intercept/Offset 2</td>
</tr>
<tr>
<td>212</td>
<td></td>
<td>Band 2 Slope/Gain 1</td>
</tr>
<tr>
<td>216</td>
<td></td>
<td>Band 2 Intercept/Offset 1</td>
</tr>
<tr>
<td>220</td>
<td></td>
<td>Band 2 Slope/Gain 2</td>
</tr>
<tr>
<td>224</td>
<td></td>
<td>Band 2 Intercept/Offset 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...or...</td>
</tr>
<tr>
<td>228</td>
<td></td>
<td>Band 3 Slope/Gain 1</td>
</tr>
<tr>
<td>232</td>
<td></td>
<td>Band 3 Intercept/Offset 1</td>
</tr>
<tr>
<td>236</td>
<td></td>
<td>Band 3 Space scan five sample average (rounded and left shifted 5 bits)</td>
</tr>
<tr>
<td>240</td>
<td></td>
<td>Band 3 Internal target temperature (scaled by 100)</td>
</tr>
<tr>
<td>244</td>
<td></td>
<td>Band 4 Slope/Gain 1</td>
</tr>
<tr>
<td>248</td>
<td></td>
<td>Band 4 Intercept/Offset 1</td>
</tr>
<tr>
<td>252</td>
<td></td>
<td>Band 4 Space scan five sample average (rounded and left shifted 5 bits)</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>Band 4 Internal target temperature (scaled by 100)</td>
</tr>
<tr>
<td>260</td>
<td></td>
<td>Band 5 Slope/Gain 1</td>
</tr>
<tr>
<td>264</td>
<td></td>
<td>Band 5 Intercept/Offset 1</td>
</tr>
<tr>
<td>268</td>
<td></td>
<td>Band 5 Space scan five sample average (rounded and left shifted 5 bits)</td>
</tr>
<tr>
<td>272</td>
<td></td>
<td>Band 5 Internal target temperature (scaled by 100)</td>
</tr>
<tr>
<td>276</td>
<td>LEV</td>
<td>Band number: 01</td>
</tr>
<tr>
<td>277</td>
<td></td>
<td>Band number: 02</td>
</tr>
<tr>
<td>278</td>
<td></td>
<td>Band number: 03 or 06</td>
</tr>
<tr>
<td>279</td>
<td></td>
<td>Band number: 04</td>
</tr>
<tr>
<td>280</td>
<td></td>
<td>Band number: 05</td>
</tr>
<tr>
<td>281</td>
<td></td>
<td>Unused Band numbers: 00 00 00 (three bytes)</td>
</tr>
</tbody>
</table>
**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 HIIRS (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**

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1</td>
<td>band map value</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>length of the line prefix documentation</td>
</tr>
<tr>
<td>49</td>
<td>196</td>
<td>length of the line prefix calibration</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>line prefix band list</td>
</tr>
<tr>
<td>51</td>
<td>0</td>
<td>image source type; 4 bytes ASCII</td>
</tr>
<tr>
<td>52</td>
<td>TIRO</td>
<td>calibration type; 4 bytes ASCII</td>
</tr>
</tbody>
</table>

**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 | x | 0 | 0 | 0 | 0

The line prefix for the data block contains the information below.

<table>
<thead>
<tr>
<th>Region</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validity code</td>
<td>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</td>
</tr>
<tr>
<td>documentation</td>
<td>196-byte DOC section from the signal transmission</td>
</tr>
<tr>
<td>calibration</td>
<td>0 bytes; not needed for TIP data</td>
</tr>
<tr>
<td>band list</td>
<td>0 bytes; not needed for TIP data</td>
</tr>
</tbody>
</table>

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 on page 7-26 describes the GOES-7 navigation block.

<table>
<thead>
<tr>
<th>Word</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>R*6.8</td>
<td>observation start time (MJD)</td>
</tr>
<tr>
<td>7-10</td>
<td>R*4.8</td>
<td>VIS channel stepping angle along line (rad)</td>
</tr>
<tr>
<td>11-14</td>
<td>R*4.8</td>
<td>IR channel stepping angle along line (rad)</td>
</tr>
<tr>
<td>15-18</td>
<td>R*4.10</td>
<td>VIS channel sampling angle along pixel (rad)</td>
</tr>
<tr>
<td>19-22</td>
<td>R*4.10</td>
<td>IR channel sampling angle along pixel (rad)</td>
</tr>
<tr>
<td>23-26</td>
<td>R*4.4</td>
<td>VIS channel center line number of VISSR frame</td>
</tr>
<tr>
<td>27-30</td>
<td>R*4.4</td>
<td>IR1 channel center line number of VISSR frame</td>
</tr>
<tr>
<td>31-34</td>
<td>R*4.4</td>
<td>VIS channel center pixel number of VISSR frame</td>
</tr>
<tr>
<td>35-38</td>
<td>R*4.4</td>
<td>IR1 channel center pixel number of VISSR frame</td>
</tr>
<tr>
<td>39-42</td>
<td>R*4.0</td>
<td>number of sensors of VIS channel</td>
</tr>
<tr>
<td>43-46</td>
<td>R*4.0</td>
<td>number of sensors of IR channel</td>
</tr>
<tr>
<td>47-50</td>
<td>R*4.0</td>
<td>VIS total line number of VISSR frame</td>
</tr>
<tr>
<td>51-54</td>
<td>R*4.0</td>
<td>IR total line number of VISSR frame</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>55-58</td>
<td>R*4.0</td>
<td>VIS pixel number of one line</td>
</tr>
<tr>
<td>59-62</td>
<td>R*4.0</td>
<td>IR pixel number of one line</td>
</tr>
<tr>
<td>63-66</td>
<td>R*4.10</td>
<td>VISSR misalignment angle around x-axis (rad)</td>
</tr>
<tr>
<td>67-70</td>
<td>R*4.10</td>
<td>VISSR misalignment angle around y-axis (rad)</td>
</tr>
<tr>
<td>71-74</td>
<td>R*4.10</td>
<td>VISSR misalignment angle around z-axis (rad)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>These words are modified as a function of the horizon points. See words 11-14 in Table A.1 in the document <em>Revision of CMS Stretched-VISSR Data Format</em> 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</td>
</tr>
<tr>
<td>75-78</td>
<td>R*4.7</td>
<td>IR2 channel center line of VISSR frame</td>
</tr>
<tr>
<td>79-82</td>
<td>R*4.10</td>
<td>IR3 channel center line number of VISSR frame</td>
</tr>
<tr>
<td>83-86</td>
<td>R*4.10</td>
<td>IR2 channel center pixel number of VISSR frame</td>
</tr>
<tr>
<td>87-90</td>
<td>R*4.10</td>
<td>IR3 channel center pixel number of VISSR frame</td>
</tr>
<tr>
<td>91-94</td>
<td>R*4.7</td>
<td>IR2 channel center line of VISSR frame</td>
</tr>
<tr>
<td>95-98</td>
<td>R*4.10</td>
<td>IR3 channel center line number of VISSR frame</td>
</tr>
<tr>
<td>99-102</td>
<td>R*4.10</td>
<td>IR2 channel center pixel number of VISSR frame</td>
</tr>
<tr>
<td>103-106</td>
<td>R*4.10</td>
<td>IR3 channel center pixel number of VISSR frame</td>
</tr>
<tr>
<td>107-110</td>
<td>R*4.7</td>
<td>IR2 channel center line of VISSR frame</td>
</tr>
<tr>
<td>111-114</td>
<td>R*4.4</td>
<td>IR2 channel center line of VISSR frame</td>
</tr>
<tr>
<td>115-118</td>
<td>R*4.4</td>
<td>IR3 channel center line number of VISSR frame</td>
</tr>
<tr>
<td>119-122</td>
<td>R*4.4</td>
<td>IR2 channel center pixel number of VISSR frame</td>
</tr>
<tr>
<td>123-126</td>
<td>R*4.4</td>
<td>IR3 channel center pixel number of VISSR frame</td>
</tr>
<tr>
<td>127-240</td>
<td>R*6.8</td>
<td>daily mean of Satellite Spin Rate (rpm)</td>
</tr>
<tr>
<td>241-246</td>
<td>R*6.8</td>
<td>daily mean of Satellite Spin Rate (rpm)</td>
</tr>
<tr>
<td>247-256</td>
<td>R*6.8</td>
<td>daily mean of Satellite Spin Rate (rpm)</td>
</tr>
<tr>
<td>257-617</td>
<td>R*6.8</td>
<td>daily mean of Satellite Spin Rate (rpm)</td>
</tr>
<tr>
<td>618-2352</td>
<td>R*6.8</td>
<td>daily mean of Satellite Spin Rate (rpm)</td>
</tr>
</tbody>
</table>

McIDAS Programmer's Manual
Revised 11/98

Format of the Data Files
7-35
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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>R*6.8</td>
<td>prediction time (UTC represented in MJD)</td>
</tr>
<tr>
<td>7-12</td>
<td>BCD*6</td>
<td>anno domini represented by BCD (YYMMDDHHmmSS: year, month, day, hour, minute, second)</td>
</tr>
<tr>
<td>13-18</td>
<td>R*6.8</td>
<td>angle between z-axis and satellite spin axis projected on yz-plane in mean of 1950.0 coordinates (rad)</td>
</tr>
<tr>
<td>19-24</td>
<td>R*6.11</td>
<td>angle between satellite spin axis and yz-plane in mean of 1950.0 coordinates (rad)</td>
</tr>
<tr>
<td>25-30</td>
<td>R*6.8</td>
<td>dihedral Angle between the Sun and the Earth measured clockwise seeing from North (rad)</td>
</tr>
<tr>
<td>31-36</td>
<td>R*6.8</td>
<td>spin Rate: spin speed of satellite (rpm)</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>R*6.8</td>
<td>prediction time (UTC represented in MJD)</td>
</tr>
<tr>
<td>7-48</td>
<td></td>
<td>not used</td>
</tr>
<tr>
<td>49-54</td>
<td>R*6.6</td>
<td>x component of satellite position in Earth-fixed coordinates (m)</td>
</tr>
<tr>
<td>55-60</td>
<td>R*6.6</td>
<td>y component of satellite position in Earth-fixed coordinates (m)</td>
</tr>
<tr>
<td>61-66</td>
<td></td>
<td>not used</td>
</tr>
<tr>
<td>67-84</td>
<td></td>
<td>not used</td>
</tr>
<tr>
<td>85-90</td>
<td>R*6.8</td>
<td>Greenwich sidereal time in true of data coordinates (deg)</td>
</tr>
<tr>
<td>91-102</td>
<td></td>
<td>not used</td>
</tr>
<tr>
<td>103-108</td>
<td>R*6.8</td>
<td>right ascension from the satellite to the sun in the Earth-fixed coordinates (deg)</td>
</tr>
<tr>
<td>109-114</td>
<td>R*6.8</td>
<td>declination from the satellite to the sun in the Earth-fixed coordinates (deg)</td>
</tr>
<tr>
<td>115-128</td>
<td></td>
<td>not used</td>
</tr>
<tr>
<td>Word</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>129-134</td>
<td>R*6.12</td>
<td>Element of nutation and precession matrix</td>
</tr>
<tr>
<td>135-140</td>
<td>R*6.14</td>
<td>row 1 and column 1</td>
</tr>
<tr>
<td>141-146</td>
<td>R*6.14</td>
<td>row 2 and column 1</td>
</tr>
<tr>
<td>147-152</td>
<td>R*6.14</td>
<td>row 3 and column 1</td>
</tr>
<tr>
<td>153-158</td>
<td>R*6.12</td>
<td>row 1 and column 2</td>
</tr>
<tr>
<td>159-164</td>
<td>R*6.16</td>
<td>row 2 and column 2</td>
</tr>
<tr>
<td>165-170</td>
<td>R*6.12</td>
<td>row 3 and column 2</td>
</tr>
<tr>
<td>171-176</td>
<td>R*6.16</td>
<td>row 1 and column 3</td>
</tr>
<tr>
<td>177-182</td>
<td>R*6.12</td>
<td>row 2 and column 3</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GMS5</td>
<td>4-byte ASCII identifier</td>
</tr>
<tr>
<td>1</td>
<td>0x58</td>
<td>directory block length in bytes</td>
</tr>
<tr>
<td>2</td>
<td>COEF</td>
<td>4-byte ASCII identifier for sub-block 1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>starting byte offset from word 0</td>
</tr>
<tr>
<td>4</td>
<td>4VIS or 5VIS</td>
<td>identifies sub-block 2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>starting byte offset from word 0</td>
</tr>
<tr>
<td>6</td>
<td>4IR or 5IR1</td>
<td>identifies sub-block 3</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>starting byte offset from word 0</td>
</tr>
<tr>
<td>8</td>
<td>5IR2</td>
<td>identifies sub-block 4</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>starting byte offset from word 0</td>
</tr>
<tr>
<td>10</td>
<td>5IR3</td>
<td>identifies sub-block 5</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>starting byte offset from word 0</td>
</tr>
<tr>
<td>12</td>
<td>SPAR</td>
<td>identifies spares block, sub block 6</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>starting byte offset from word 0</td>
</tr>
<tr>
<td>14-21</td>
<td></td>
<td>not used</td>
</tr>
<tr>
<td>22-85</td>
<td></td>
<td>Words 1 to 256 of GMS 4-or -5 calibration block; see the table below</td>
</tr>
<tr>
<td>86-127</td>
<td></td>
<td>reserved for future tables (GMS-6, etc..)</td>
</tr>
</tbody>
</table>
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-VISSL data format* document for a complete description of the data block.

<table>
<thead>
<tr>
<th>Byte</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>calibration information ID</td>
</tr>
<tr>
<td>5-10</td>
<td>data generated date (YYYYMMDDHHmm)</td>
</tr>
<tr>
<td>11</td>
<td>sensor selection</td>
</tr>
<tr>
<td>12-256</td>
<td>sub-block 1; infrared radiance calculations</td>
</tr>
<tr>
<td>257-1280</td>
<td>sub-block 2; visible level-to-albedo conversion tables; four 64-level tables for VIS1 through VIS4 detectors</td>
</tr>
<tr>
<td>1281-2304</td>
<td>sub-block 3; IR1 level-to-temperature conversion table</td>
</tr>
<tr>
<td>2305-3328</td>
<td>sub-block 4; IR2 level-to-temperature conversion table</td>
</tr>
<tr>
<td>3329-4352</td>
<td>sub-block 5; IR3 level-to-temperature conversion table</td>
</tr>
<tr>
<td>4353-6400</td>
<td>sub-block 6; spares</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>not a visible band</td>
</tr>
<tr>
<td>6C6C0000</td>
<td>detector 1 of the visible band</td>
</tr>
<tr>
<td>B4B40000</td>
<td>detector 2 of the visible band</td>
</tr>
<tr>
<td>D8D80000</td>
<td>detector 3 of the visible band</td>
</tr>
<tr>
<td>FCFC0000</td>
<td>detector 4 of the visible band</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DMSP</td>
<td>navigation type; indicates the signal type being navigated</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>sensor source, year and date of the signal; ssyyddd</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>nominal time of image; hhmmss</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>orbit type; set to one</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>epoch date from the ASCII record below; yyddd</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>epoch time from the ASCII record below; days*1.e9</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>mean motion from the ASCII record below; deriv*1.e9</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>mean motion from ASCII record below; accel and mantissa</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>mean motion from the ASCII record below; accel and expon</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>bstart and mantissa; usually not used</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>b-start and expon; usually not used</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>inclination from the ASCII record below</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>right ascension of ascending node from the ASCII record below; deg*1.e6</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>eccentricity from the ASCII record below; ecc*1.e6</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>argument of perigee from the ASCII record below; per,deg*1.e7</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>mean anomaly from the ASCII record below; anomaly<em>deg</em>1.e6</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>mean motion; revs/day*1.e7</td>
</tr>
<tr>
<td>17-43</td>
<td></td>
<td>unused</td>
</tr>
<tr>
<td>44</td>
<td></td>
<td>data type; OLS, MT, MT2 or MI</td>
</tr>
<tr>
<td>45</td>
<td></td>
<td>ascending LT; 0 = morning, 1 = evening</td>
</tr>
<tr>
<td>46</td>
<td></td>
<td>number of first scan to navigate</td>
</tr>
<tr>
<td>47</td>
<td></td>
<td>time of first scan; sec*1.e3</td>
</tr>
<tr>
<td>48</td>
<td></td>
<td>scan flipped flag; 0 = no 1 = yes</td>
</tr>
<tr>
<td>49</td>
<td></td>
<td>element flipped flag; 0 = no 1 = yes</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>unused</td>
</tr>
<tr>
<td>51</td>
<td></td>
<td>number of elements per scan</td>
</tr>
<tr>
<td>52-119</td>
<td></td>
<td>unused</td>
</tr>
<tr>
<td>120-127</td>
<td></td>
<td>32-character ASCII record</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Character numbers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-4</td>
<td>spacecraft ID</td>
</tr>
<tr>
<td>6-17</td>
<td>epoch date; a decimal point indicates a fraction of a day</td>
</tr>
<tr>
<td>18-25</td>
<td>mean motion derivative</td>
</tr>
<tr>
<td>26-28</td>
<td>mean motion acc mantissa</td>
</tr>
<tr>
<td>28-30</td>
<td>mean motion acc exponent</td>
</tr>
<tr>
<td>32-39</td>
<td>inclination</td>
</tr>
<tr>
<td>41-48</td>
<td>RA of ascending mode</td>
</tr>
<tr>
<td>50-56</td>
<td>eccentricity</td>
</tr>
<tr>
<td>58-65</td>
<td>perigee</td>
</tr>
<tr>
<td>67-74</td>
<td>mean anomaly</td>
</tr>
<tr>
<td>76-85</td>
<td>mean motion</td>
</tr>
</tbody>
</table>

**DMSP Calibration**

The OLS and microwave sensors use different calibration. The OLS calibration uses only one pair 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 0.4726. The calibration module *kboxls.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.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>validity code</td>
</tr>
<tr>
<td>2-4</td>
<td>hot load temperatures; two measurements in byte 2, one measurement in byte 3</td>
</tr>
<tr>
<td>5-6</td>
<td>radiometer temperatures; forward radiometer temperature</td>
</tr>
<tr>
<td>Bytes</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>7-13</td>
<td>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)</td>
</tr>
<tr>
<td>14-15</td>
<td>reference voltages; reference voltage and reference return both from block 6</td>
</tr>
<tr>
<td>16-20</td>
<td>cold voltages band 1</td>
</tr>
<tr>
<td>21-25</td>
<td>hot voltage band 1</td>
</tr>
<tr>
<td>26-30</td>
<td>cold voltages band 2</td>
</tr>
<tr>
<td>31-35</td>
<td>hot voltages band 2</td>
</tr>
<tr>
<td>36-40</td>
<td>cold voltages band 3</td>
</tr>
<tr>
<td>41-45</td>
<td>hot voltages band 3</td>
</tr>
<tr>
<td>46-50</td>
<td>cold voltages band 4</td>
</tr>
<tr>
<td>51-55</td>
<td>hot voltages band 4</td>
</tr>
<tr>
<td>56-60</td>
<td>cold voltages band 5</td>
</tr>
<tr>
<td>61-65</td>
<td>hot voltages band 5</td>
</tr>
<tr>
<td>66-70</td>
<td>cold voltages band 6</td>
</tr>
<tr>
<td>71-75</td>
<td>hot voltages band 6</td>
</tr>
<tr>
<td>76-80</td>
<td>cold voltages band 7</td>
</tr>
<tr>
<td>81-85</td>
<td>hot voltages band 7</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>gain, band 1</td>
</tr>
<tr>
<td>2-3</td>
<td>offset, band 1</td>
</tr>
<tr>
<td>4-5</td>
<td>gain, band 2</td>
</tr>
<tr>
<td>6-7</td>
<td>offset, band 2</td>
</tr>
<tr>
<td>8-9</td>
<td>gain, band 3</td>
</tr>
<tr>
<td>10-11</td>
<td>offset, band 3</td>
</tr>
<tr>
<td>12-13</td>
<td>gain, band 4</td>
</tr>
<tr>
<td>14-15</td>
<td>offset, band 4</td>
</tr>
<tr>
<td>Bytes</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------</td>
</tr>
<tr>
<td>16-17</td>
<td>gain, band 5</td>
</tr>
<tr>
<td>18-19</td>
<td>offset, band 6</td>
</tr>
<tr>
<td>20-21</td>
<td>offset, band 6</td>
</tr>
<tr>
<td>22-23</td>
<td>gain, band 7</td>
</tr>
<tr>
<td>24-25</td>
<td>offset, band 7</td>
</tr>
</tbody>
</table>

**API functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcadir</td>
<td>opens a connection to read the directory block from an image object</td>
</tr>
<tr>
<td>mcadrd</td>
<td>reads the directory block from an image object</td>
</tr>
<tr>
<td>mcaget</td>
<td>opens a connection to read the data block from an image object</td>
</tr>
<tr>
<td>mcalin</td>
<td>reads the data portion of the current image line</td>
</tr>
<tr>
<td>mcapfx</td>
<td>reads the prefix portion of the current image line</td>
</tr>
<tr>
<td>mcnav</td>
<td>reads the navigation block of an image object</td>
</tr>
<tr>
<td>mcacal</td>
<td>reads the calibration block of an image object</td>
</tr>
<tr>
<td>mcacrd</td>
<td>reads the comment block of an image object</td>
</tr>
<tr>
<td>mcafree</td>
<td>frees the handle and memory of a connection opened by <code>mcaget</code></td>
</tr>
<tr>
<td>mcaput</td>
<td>opens a connection to write the directory, navigation and calibration blocks of an image object</td>
</tr>
<tr>
<td>mcaout</td>
<td>writes the line prefix and data portions of an image line</td>
</tr>
<tr>
<td>mcacou</td>
<td>writes the comment block to an image object</td>
</tr>
<tr>
<td>mcadel</td>
<td>deletes image objects from a dataset</td>
</tr>
<tr>
<td>mcasort</td>
<td>gets the parameters from the command line and adds them to the selection array for a future <code>mcaget</code> call</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Schema file</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCF014</td>
<td>NGM MOS forecasts</td>
</tr>
<tr>
<td>DCIRAB</td>
<td>international radiosonde observations</td>
</tr>
<tr>
<td>DCISEN</td>
<td>isentropic data</td>
</tr>
<tr>
<td>DCISFC</td>
<td>international surface hourly observations</td>
</tr>
<tr>
<td>DCISHP</td>
<td>international ship and buoy observations</td>
</tr>
<tr>
<td>DCIPRP</td>
<td>PIREP, AIREP and ACARS data</td>
</tr>
<tr>
<td>DCSYN</td>
<td>surface synoptic observations</td>
</tr>
</tbody>
</table>

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"
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>four-character schema name</td>
</tr>
<tr>
<td>version</td>
<td>schema version number</td>
</tr>
<tr>
<td>date</td>
<td>Julian day the schema definition file was created</td>
</tr>
<tr>
<td>id</td>
<td>schema identification number</td>
</tr>
<tr>
<td>&quot;description&quot;</td>
<td>comments</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>default number of rows to make for this MD file</td>
</tr>
<tr>
<td>rkeyname1 ... N</td>
<td>name of the first and Nth row header keys</td>
</tr>
<tr>
<td>rscale1 ... N</td>
<td>scale factor of the first and Nth row header keys</td>
</tr>
<tr>
<td>runits1 ... N</td>
<td>physical units of the first and Nth row header keys</td>
</tr>
<tr>
<td>&quot;description</td>
<td>comments</td>
</tr>
</tbody>
</table>

COLUMNS ncols "description
 ckeyname1 cscale1 cunits1 "description
 ... 
ckeynameN cscaleN cunitsN "description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ncols</td>
<td>default number of columns to make for this MD file</td>
</tr>
<tr>
<td>ckeyname1 ... N</td>
<td>name of the first and Nth column header keys</td>
</tr>
<tr>
<td>cscale1 ... N</td>
<td>scale factor of the first and Nth column header keys</td>
</tr>
<tr>
<td>cunits1 ... N</td>
<td>physical units of the first and Nth column header keys</td>
</tr>
<tr>
<td>&quot;description</td>
<td>comments</td>
</tr>
</tbody>
</table>

Data lines

Use the format below for the data lines.

DATA
dkeyname1 dscale1 dunits1 "description
 ... 
dkeynameN dscaleN dunitsN "description
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>dkeyname1 ... N</td>
<td>name of the first and Nth data keys</td>
</tr>
<tr>
<td>dscale1 ... N</td>
<td>scale factor of the first and Nth data keys</td>
</tr>
<tr>
<td>dunits1 ... N</td>
<td>physical units of the first and Nth data keys</td>
</tr>
<tr>
<td>&quot;description&quot;</td>
<td>comments</td>
</tr>
</tbody>
</table>

**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"
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrepeats</td>
<td>number of repeat groups in the data portion of the schema definition</td>
</tr>
<tr>
<td>rkeynames</td>
<td>names of the repeated data keys</td>
</tr>
<tr>
<td>rscales</td>
<td>scale factors of the repeated data keys</td>
</tr>
<tr>
<td>runits</td>
<td>physical units of the repeated data keys</td>
</tr>
<tr>
<td>&quot;description&quot;</td>
<td>comments</td>
</tr>
</tbody>
</table>

**End-of-schema and comment formats**

To end the schema definition file, use ENDSHEMA. 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 *MDXXnmm* data structure in this chapter.
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 2 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. The next 48 words contain the red, green and blue intensities for the 16 graphic color levels on a VGA display. These intensities range from 0 to 15.

**Word allocation**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reserved for system use</td>
</tr>
<tr>
<td>1</td>
<td>red color intensity for an image brightness of 0</td>
</tr>
<tr>
<td>2</td>
<td>red color intensity for an image brightness of 1</td>
</tr>
<tr>
<td>3</td>
<td>red color intensity for an image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>256</td>
<td>red color intensity for an image brightness of 255</td>
</tr>
<tr>
<td>257</td>
<td>green color intensity for an image brightness of 0</td>
</tr>
<tr>
<td>258</td>
<td>green color intensity for an image brightness of 1</td>
</tr>
<tr>
<td>259</td>
<td>green color intensity for an image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>512</td>
<td>green color intensity for an image brightness of 255</td>
</tr>
<tr>
<td>513</td>
<td>blue color intensity for an image brightness of 0</td>
</tr>
<tr>
<td>514</td>
<td>blue color intensity for an image brightness of 1</td>
</tr>
<tr>
<td>515</td>
<td>blue color intensity for an image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>768</td>
<td>blue color intensity for an image brightness of 255</td>
</tr>
<tr>
<td>769</td>
<td>red color intensity for VGA image brightness of 0</td>
</tr>
<tr>
<td>770</td>
<td>red color intensity for VGA image brightness of 1</td>
</tr>
<tr>
<td>771</td>
<td>red color intensity for VGA image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>784</td>
<td>red color intensity for VGA image brightness of 15</td>
</tr>
<tr>
<td>785</td>
<td>green color intensity for VGA image brightness of 0</td>
</tr>
<tr>
<td>786</td>
<td>green color intensity for VGA image brightness of 1</td>
</tr>
<tr>
<td>787</td>
<td>green color intensity for VGA image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>800</td>
<td>green color intensity for VGA image brightness of 15</td>
</tr>
<tr>
<td>801</td>
<td>blue color intensity for VGA image brightness of 0</td>
</tr>
<tr>
<td>802</td>
<td>blue color intensity for VGA image brightness of 1</td>
</tr>
<tr>
<td>803</td>
<td>blue color intensity for VGA image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>816</td>
<td>blue color intensity for VGA image brightness of 15</td>
</tr>
</tbody>
</table>

**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 unpaneled. 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, its two bytes contain HEX 8080. If the frame was explicitly set to a 1x1 geometry, the bytes contain HEX 0101. The software considers them both to represent an unpaneled frame.

**Byte allocation**

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>number of panels in the x-direction for frame 1</td>
</tr>
<tr>
<td>1</td>
<td>number of panels in the y-direction for frame 1</td>
</tr>
<tr>
<td>2</td>
<td>number of panels in the x-direction for frame 2</td>
</tr>
<tr>
<td>3</td>
<td>number of panels in the y-direction for frame 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Format of the Data Files
7-48

McIDAS Programmer's Manual
Revised 11/98
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.

\[
\text{position} = (\text{frame number} \times 816) + 1
\]

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. The remaining 48 words contain the red, green and blue intensities for the 16 graphic color levels on a VGA display. These intensities range from 0 to 15.

### Word allocation for each file

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 815</td>
<td>reserved for system use</td>
</tr>
<tr>
<td>816 - 1631</td>
<td>image frame 1 enhancement table</td>
</tr>
<tr>
<td>1632 - 2447</td>
<td>image frame 2 enhancement table</td>
</tr>
<tr>
<td>2448 - 3263</td>
<td>image frame 3 enhancement table</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Word allocation for each table

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>red color intensity for an image brightness of 0</td>
</tr>
<tr>
<td>1</td>
<td>red color intensity for an image brightness of 1</td>
</tr>
<tr>
<td>2</td>
<td>red color intensity for an image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>255</td>
<td>red color intensity for an image brightness of 255</td>
</tr>
<tr>
<td>256</td>
<td>green color intensity for an image brightness of 0</td>
</tr>
<tr>
<td>257</td>
<td>green color intensity for an image brightness of 1</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>258</td>
<td>green color intensity for an image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>511</td>
<td>green color intensity for an image brightness of 255</td>
</tr>
<tr>
<td>512</td>
<td>blue color intensity for an image brightness of 0</td>
</tr>
<tr>
<td>513</td>
<td>blue color intensity for an image brightness of 1</td>
</tr>
<tr>
<td>514</td>
<td>blue color intensity for an image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>767</td>
<td>blue color intensity for an image brightness of 255</td>
</tr>
<tr>
<td>768</td>
<td>red color intensity for VGA image brightness of 0</td>
</tr>
<tr>
<td>769</td>
<td>red color intensity for VGA image brightness of 1</td>
</tr>
<tr>
<td>770</td>
<td>red color intensity for VGA image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>783</td>
<td>red color intensity for VGA image brightness of 15</td>
</tr>
<tr>
<td>784</td>
<td>green color intensity for VGA image brightness of 0</td>
</tr>
<tr>
<td>785</td>
<td>green color intensity for VGA image brightness of 1</td>
</tr>
<tr>
<td>786</td>
<td>green color intensity for VGA image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>799</td>
<td>green color intensity for VGA image brightness of 15</td>
</tr>
<tr>
<td>800</td>
<td>blue color intensity for VGA image brightness of 0</td>
</tr>
<tr>
<td>801</td>
<td>blue color intensity for VGA image brightness of 1</td>
</tr>
<tr>
<td>802</td>
<td>blue color intensity for VGA image brightness of 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>815</td>
<td>blue color intensity for VGA image brightness of 15</td>
</tr>
</tbody>
</table>

**API functions**

Currently, no API library functions exist for reading and writing frame enhancement files.
FRAMEEx.y

Image frame files, where x is the frame number and y is the panel number.

The image frame file is a binary file that describes the contents of an image frame. For unpaneled frames, y is zero. An unpaneled frame 1 uses file FRAME1.0; a 4-panel frame 2 uses files FRAME2.1 through FRAME2.4.

The file has three components:

- a 64-word frame directory block
- a user-defined extension block
- a 640-word navigation block

The default size of the user extension is zero. The size can be modified by editing the file m0panel.h and rebuilding McIDAS. 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 Satellite-specific characteristics for image files (AREAnnnn).

Frame directory

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>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</td>
</tr>
<tr>
<td>1</td>
<td>year and Julian day of the image, ccyyddd</td>
</tr>
<tr>
<td>2</td>
<td>time of the image, hHmmss</td>
</tr>
<tr>
<td>3</td>
<td>spectral band</td>
</tr>
<tr>
<td>4</td>
<td>upper-left image line in satellite coordinates</td>
</tr>
<tr>
<td>5</td>
<td>upper-left image element in satellite coordinates</td>
</tr>
<tr>
<td>6</td>
<td>reserved for system use</td>
</tr>
<tr>
<td>7</td>
<td>image frame line containing the upper-left image line</td>
</tr>
<tr>
<td>8</td>
<td>image frame element containing the upper-left image element</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>9</td>
<td>line blowup; also see word 32</td>
</tr>
<tr>
<td>10</td>
<td>line blowdown</td>
</tr>
<tr>
<td>11</td>
<td>element blowdown; also see word 33</td>
</tr>
<tr>
<td>12</td>
<td>year and Julian day the image was created, ccyyddd</td>
</tr>
<tr>
<td>13</td>
<td>time the image was created, hhmmss</td>
</tr>
<tr>
<td>14</td>
<td>year and Julian day the image frame was created, ccyyddd</td>
</tr>
<tr>
<td>15</td>
<td>time the image frame was created, hhmmss</td>
</tr>
<tr>
<td>16</td>
<td>number of the area that the frame was loaded from; 0 if loaded with IMGDISP</td>
</tr>
<tr>
<td>17 - 31</td>
<td>identification block (optional); CHARACTER</td>
</tr>
<tr>
<td>32</td>
<td>if positive, element blowup; if negative, same as word 11</td>
</tr>
<tr>
<td>33</td>
<td>digital data byte offset in the associated area</td>
</tr>
<tr>
<td>34</td>
<td>navigation block byte offset in the associated area</td>
</tr>
<tr>
<td>35</td>
<td>number of bytes per data point</td>
</tr>
<tr>
<td>36</td>
<td>original source type if changed by IMGCOPY, AAMAP, etc.; CHARACTER</td>
</tr>
<tr>
<td>37 - 38</td>
<td>file name used by the SU application; CHARACTER</td>
</tr>
<tr>
<td>39 - 54</td>
<td>breakpoints for VGA; gray levels 0 through 15; -1 indicates graphics level usage</td>
</tr>
<tr>
<td>55 - 60</td>
<td>ASCII ADDE dataset name from IMGDISP</td>
</tr>
<tr>
<td>61</td>
<td>binary position number within the ADDE dataset</td>
</tr>
<tr>
<td>62 - 63</td>
<td>reserved for system use</td>
</tr>
</tbody>
</table>

**API functions**

<table>
<thead>
<tr>
<th>Fortran Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>erafrm</td>
<td>flags a frame directory or navigation block as unused (erased)</td>
</tr>
<tr>
<td>frtonv</td>
<td>reads a navigation block</td>
</tr>
<tr>
<td>getfrm</td>
<td>reads a frame directory block</td>
</tr>
<tr>
<td>m0getusr</td>
<td>reads a user extension block</td>
</tr>
<tr>
<td>m0putusr</td>
<td>writes a user extension block</td>
</tr>
<tr>
<td>nvtofr</td>
<td>writes a navigation block</td>
</tr>
<tr>
<td>putfrm</td>
<td>writes a frame directory block</td>
</tr>
</tbody>
</table>
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 and -OS2.

Word allocation

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>number of IR calibration tables in the file</td>
</tr>
<tr>
<td>1</td>
<td>identification number of the first IR table in the file</td>
</tr>
<tr>
<td>2</td>
<td>identification number of the second IR table in the file</td>
</tr>
<tr>
<td>511</td>
<td>identification number of the 511th IR table, if present</td>
</tr>
<tr>
<td>512 - 767</td>
<td>interpolated VIS calibration table; albedos are multiplied by 10**6 and stored as integers</td>
</tr>
<tr>
<td>1024 - 1279</td>
<td>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</td>
</tr>
<tr>
<td>1280 - 1535</td>
<td>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</td>
</tr>
</tbody>
</table>

API functions

Currently, no API library functions exist for reading and writing GMSCAL.
GRIDnnnn

Grid files, where nnnn is a user-defined number.

A grid has two components:

- a 64-word header describing the grid projection, grid directory, gridded variable, level parameter, time of the data, etc.
- the actual grid data, which is an \( n \times n \)-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.

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.

**Grid file directory**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 7</td>
<td>32 characters of label information</td>
</tr>
<tr>
<td>8</td>
<td>project number used to create the grid file</td>
</tr>
<tr>
<td>9</td>
<td>date the file was created, yyyydd</td>
</tr>
<tr>
<td>10</td>
<td>maximum number of grids (( n )) in the grid file</td>
</tr>
<tr>
<td>10 + 1</td>
<td>word offset, from the beginning of the grid file, where grid 1 starts; if the offset is -1, no grids exist</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10 + n</td>
<td>word offset for grid ( n )</td>
</tr>
<tr>
<td>10 + n + 1</td>
<td>next available address to start writing the next grid</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>total size; rows * columns (not to exceed 65525 or contouring routines will not work)</td>
</tr>
<tr>
<td>1</td>
<td>number of rows</td>
</tr>
<tr>
<td>2</td>
<td>number of columns</td>
</tr>
<tr>
<td>3</td>
<td>Julian date of the data, yyyyddd</td>
</tr>
<tr>
<td>4</td>
<td>time of the data, hhmmss</td>
</tr>
<tr>
<td>5</td>
<td>valid time for the grid, if applicable</td>
</tr>
<tr>
<td>6</td>
<td>name of the gridded variable, MD file terms</td>
</tr>
<tr>
<td>7</td>
<td>scale of the gridded variable, MD file terms</td>
</tr>
<tr>
<td>8</td>
<td>units of the gridded variable, MD file terms</td>
</tr>
<tr>
<td>9</td>
<td>value of the vertical level</td>
</tr>
<tr>
<td>10</td>
<td>scale of the vertical level</td>
</tr>
<tr>
<td>11</td>
<td>unit of the vertical level</td>
</tr>
<tr>
<td>12</td>
<td>grid variable type:</td>
</tr>
<tr>
<td></td>
<td>1 = time difference</td>
</tr>
<tr>
<td></td>
<td>2 = time average</td>
</tr>
<tr>
<td></td>
<td>4 = level difference</td>
</tr>
<tr>
<td></td>
<td>8 = level average (or any sum of 1, 2, 4 and 8)</td>
</tr>
<tr>
<td>13</td>
<td>used if the grid parameter is a time difference or time average, hhmmss</td>
</tr>
<tr>
<td>14</td>
<td>used if the grid parameter is a level difference or level average; values are the same as Word 9</td>
</tr>
<tr>
<td>15 - 31</td>
<td>reserved</td>
</tr>
<tr>
<td>32</td>
<td>grid origin; identifies the type of program that generated the grid data</td>
</tr>
<tr>
<td>33</td>
<td>grid type:</td>
</tr>
<tr>
<td></td>
<td>1 = pseudo-Mercator</td>
</tr>
<tr>
<td></td>
<td>2 = polar stereographic or Lambert conformal</td>
</tr>
<tr>
<td></td>
<td>3 = equidistant</td>
</tr>
<tr>
<td></td>
<td>4 = pseudo-Mercator (more general)</td>
</tr>
<tr>
<td></td>
<td>5 = no navigation</td>
</tr>
<tr>
<td>Header Word</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>40 - 45</td>
<td>reserved</td>
</tr>
<tr>
<td>46 - 51</td>
<td>reserved; filled only if the grid was created by the McIDAS-XCD GRIB decoder</td>
</tr>
<tr>
<td>52 - 63</td>
<td>grid description</td>
</tr>
</tbody>
</table>

**Remaining header words if grid type is pseudo-Mercator**

<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>maximum latitude of the grid, degrees*10000</td>
</tr>
<tr>
<td>35</td>
<td>maximum longitude of the grid, degrees*10000</td>
</tr>
<tr>
<td>36</td>
<td>minimum latitude of the grid, degrees*10000</td>
</tr>
<tr>
<td>37</td>
<td>minimum longitude of the grid, degrees*10000</td>
</tr>
</tbody>
</table>

If TYPE = 1:

<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>increment between grid points; same in x/y directions</td>
</tr>
<tr>
<td>39</td>
<td>reserved</td>
</tr>
</tbody>
</table>

If TYPE = 4:

<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>increment between the grid points (latitude)</td>
</tr>
<tr>
<td>39</td>
<td>increment between the grid points (longitude)</td>
</tr>
</tbody>
</table>

**Remaining header words if grid type is polar stereographic or Lambert conformal**

<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>row number of the North Pole*10000</td>
</tr>
<tr>
<td>35</td>
<td>column number of the North Pole*10000</td>
</tr>
<tr>
<td>36</td>
<td>column spacing at standard latitude, meters</td>
</tr>
<tr>
<td>37</td>
<td>longitude parallel to columns, degrees*10000</td>
</tr>
<tr>
<td>38 - 39</td>
<td>standard latitudes, degrees*10000; set these two equal for polar stereographic</td>
</tr>
</tbody>
</table>
### Remaining header words if grid type is equidistant

<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>latitude of (1,1), degrees*10000</td>
</tr>
<tr>
<td>35</td>
<td>longitude of (1,1), degrees*10000</td>
</tr>
<tr>
<td>36</td>
<td>clockwise rotation of column 1 relative to north, degrees*10000</td>
</tr>
<tr>
<td>37</td>
<td>column spacing, in meters</td>
</tr>
<tr>
<td>38</td>
<td>row spacing, in meters</td>
</tr>
</tbody>
</table>

### Remaining header words if grid type is no navigation

<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34 - 39</td>
<td>reserved</td>
</tr>
<tr>
<td>40</td>
<td>user's initials</td>
</tr>
<tr>
<td>41</td>
<td>project number under which the grid was created</td>
</tr>
<tr>
<td>42</td>
<td>arbitrary character ID supplied by the program</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Header Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>date grid was filed, yyyddd</td>
</tr>
<tr>
<td>47</td>
<td>time grid was filed, hhmmss</td>
</tr>
<tr>
<td>48</td>
<td>original geographic ID; GRIB projection number (PDS octet 7)</td>
</tr>
<tr>
<td>49</td>
<td>original parameter ID; GRIB parameter number (PDS octet 9)</td>
</tr>
<tr>
<td>50</td>
<td>original process ID or model number (PDS octet 6)</td>
</tr>
<tr>
<td>51</td>
<td>original level type (PDS octet 10)</td>
</tr>
</tbody>
</table>
Grid data

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset + 64</td>
<td>beginning of the grid data</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>offset + 64 + (rows*columns) - 1</td>
<td>end of the grid data</td>
</tr>
</tbody>
</table>

API functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcgdir</td>
<td>opens a connection to read a grid file directory</td>
</tr>
<tr>
<td>mcgdrd</td>
<td>reads a grid file directory</td>
</tr>
<tr>
<td>mcgfdrd</td>
<td>reads a grid file header</td>
</tr>
<tr>
<td>mcgget</td>
<td>opens a connection to read a grid</td>
</tr>
<tr>
<td>mcgridf</td>
<td>reads a grid in Fortran (column-major) order</td>
</tr>
<tr>
<td>mcgridc</td>
<td>reads a grid in C (row-major) order</td>
</tr>
<tr>
<td>mcgput</td>
<td>opens a connection to write a grid</td>
</tr>
<tr>
<td>mcgoutf</td>
<td>writes a grid in Fortran (column-major) order</td>
</tr>
<tr>
<td>mcgoutc</td>
<td>writes a grid in C (row-major) order</td>
</tr>
<tr>
<td>igquit</td>
<td>deletes a grid file</td>
</tr>
<tr>
<td>m0gsort</td>
<td>gets grid selection parameters from the command line</td>
</tr>
</tbody>
</table>
*.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.

**Word allocation**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 3</td>
<td>reserved</td>
</tr>
<tr>
<td>4</td>
<td>red color intensity for graphics level 1</td>
</tr>
<tr>
<td>5</td>
<td>green color intensity for graphics level 1</td>
</tr>
<tr>
<td>6</td>
<td>blue color intensity for graphics level 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3 * (n-1) + 1</td>
<td>red color intensity for graphics level n</td>
</tr>
<tr>
<td>3 * (n-1) + 2</td>
<td>green color intensity for graphics level n</td>
</tr>
<tr>
<td>3 * (n-1) + 3</td>
<td>blue color intensity for graphics level n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3 * (max-1) + 1</td>
<td>red color intensity for the maximum graphics level</td>
</tr>
<tr>
<td>3 * (max-1) + 2</td>
<td>green color intensity for the maximum graphics level</td>
</tr>
<tr>
<td>3 * (max-1) + 3</td>
<td>blue color intensity for the maximum graphics level</td>
</tr>
</tbody>
</table>

**API functions**

<table>
<thead>
<tr>
<th>Fortran Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delgra</td>
<td>deletes a saved graphics table</td>
</tr>
<tr>
<td>resgra</td>
<td>restores a saved graphics table</td>
</tr>
<tr>
<td>savgra</td>
<td>writes a graphics save table</td>
</tr>
</tbody>
</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 and -OS2.

**Word allocation for record 1**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>total number of records in the file</td>
</tr>
<tr>
<td>1 - 47</td>
<td>unused</td>
</tr>
</tbody>
</table>

**Word allocation for records 2 through n**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 7</td>
<td>array of thermistor counts-to-temperature coefficients, zeroth order</td>
</tr>
<tr>
<td>8 - 15</td>
<td>array of thermistor counts-to-temperature coefficients, first order</td>
</tr>
<tr>
<td>16 - 23</td>
<td>array of thermistor counts-to-temperature coefficients, second order</td>
</tr>
<tr>
<td>24 - 31</td>
<td>array of thermistor counts-to-temperature coefficients, third order</td>
</tr>
<tr>
<td>32 - 39</td>
<td>array of thermistor counts-to-temperature coefficients, fourth order</td>
</tr>
<tr>
<td>40 - 43</td>
<td>array of weights for blackbody target temperature</td>
</tr>
<tr>
<td>44</td>
<td>coefficient for calibrating the HIRS visible channel</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>---------------</td>
</tr>
<tr>
<td>45</td>
<td>reserved</td>
</tr>
<tr>
<td>46</td>
<td>NOAA satellite number</td>
</tr>
<tr>
<td>47</td>
<td>date, yyyddd</td>
</tr>
</tbody>
</table>

**API functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hircon</td>
<td>initializes the HIRS constants</td>
</tr>
</tbody>
</table>
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 and -OS2.

**Word allocation for records 1-19**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 199</td>
<td>carbon dioxide transmittance coefficients</td>
</tr>
<tr>
<td>200 - 319</td>
<td>carbon dioxide angular correction coefficients</td>
</tr>
<tr>
<td>320 - 325</td>
<td>continuum transmittance coefficients</td>
</tr>
<tr>
<td>326 - 339</td>
<td>water vapor transmittance coefficients</td>
</tr>
<tr>
<td>340 - 419</td>
<td>ozone transmittance coefficients</td>
</tr>
<tr>
<td>420 - 499</td>
<td>ozone angular correction coefficients</td>
</tr>
<tr>
<td>500 - 505</td>
<td>flags indicating whether to use coefficients</td>
</tr>
</tbody>
</table>

**Word allocation for record 20**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 19</td>
<td>central wave numbers of the channels</td>
</tr>
<tr>
<td>20 - 59</td>
<td>coefficients for the Planck function</td>
</tr>
<tr>
<td>60 - 99</td>
<td>band correction coefficients</td>
</tr>
<tr>
<td>100 - 139</td>
<td>reserved</td>
</tr>
<tr>
<td>140 - 144</td>
<td>solar coefficients</td>
</tr>
<tr>
<td>145 - 157</td>
<td>flux coefficients</td>
</tr>
<tr>
<td>158 - 179</td>
<td>reserved</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>180 - 187</td>
<td>synthetic coefficients for the surface temperature</td>
</tr>
<tr>
<td>188 - 189</td>
<td>reserved</td>
</tr>
<tr>
<td>190 - 197</td>
<td>empirical coefficients for the surface temperature</td>
</tr>
<tr>
<td>198 - 505</td>
<td>reserved</td>
</tr>
</tbody>
</table>

**API functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pcoef</td>
<td>retrieves HIRS Planck function, solar, flux and surface temperature coefficients</td>
</tr>
</tbody>
</table>
IDMSL

Reporting station list.

IDMSL is a variable-length, binary file that lists all reporting stations and location information. It is supplied with McIDAS-X and -OS2.

**Word allocation**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( n ), number of stations in the list</td>
</tr>
<tr>
<td>1 to 2n</td>
<td>8-character station ICAO identifier, left justified (ASCII)</td>
</tr>
<tr>
<td>2n + 1 to 3n</td>
<td>6-digit station identifier</td>
</tr>
<tr>
<td>3n + 1 to 4n</td>
<td>latitude, degrees*10000</td>
</tr>
<tr>
<td>4n + 1 to 5n</td>
<td>longitude, degrees*10000</td>
</tr>
<tr>
<td>5n + 1 to 6n</td>
<td>elevation, meters</td>
</tr>
<tr>
<td>6n + 1 to 7n</td>
<td>2-character state code</td>
</tr>
<tr>
<td>7n + 1 to 8n</td>
<td>2-character country code</td>
</tr>
<tr>
<td>8n + 1 to 9n</td>
<td>reported data types; packed integer representation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Reporting type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3-hourly synoptic</td>
</tr>
<tr>
<td>1</td>
<td>6-hourly synoptic</td>
</tr>
<tr>
<td>2</td>
<td>off-hourly synoptic</td>
</tr>
<tr>
<td>3</td>
<td>AERO</td>
</tr>
<tr>
<td>4</td>
<td>airways</td>
</tr>
<tr>
<td>5</td>
<td>METAR</td>
</tr>
<tr>
<td>6</td>
<td>RAOB</td>
</tr>
<tr>
<td>7</td>
<td>PIBAL</td>
</tr>
<tr>
<td>8</td>
<td>TAF forecast</td>
</tr>
<tr>
<td>9</td>
<td>PLATF forecast</td>
</tr>
<tr>
<td>10</td>
<td>radar</td>
</tr>
<tr>
<td>11</td>
<td>coastal/SMARS</td>
</tr>
<tr>
<td>12</td>
<td>short-range terminal forecast</td>
</tr>
<tr>
<td>13</td>
<td>hourly (SA)</td>
</tr>
<tr>
<td>14</td>
<td>forecast (FT)</td>
</tr>
<tr>
<td>15</td>
<td>radar (SD)</td>
</tr>
<tr>
<td>16</td>
<td>ASOS/AWOS</td>
</tr>
</tbody>
</table>

| 9n + 1 to 9n + 5n | 20-character station name          |
| 14n + 1 to 15n    | surface station reporting characteristics |
### API functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idinfo</td>
<td>provides information about a reporting station</td>
</tr>
</tbody>
</table>
MDXXnnnn

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. 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 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.

**Word allocation**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>schema name</td>
</tr>
<tr>
<td>1</td>
<td>schema version number</td>
</tr>
<tr>
<td>2</td>
<td>schema registration date, yyyydd</td>
</tr>
<tr>
<td>3</td>
<td>default number of rows</td>
</tr>
<tr>
<td>4</td>
<td>default number of columns</td>
</tr>
<tr>
<td>5</td>
<td>total number of keys in the record</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>6</td>
<td>number of keys in the row header</td>
</tr>
<tr>
<td>7</td>
<td>number of keys in the column header</td>
</tr>
<tr>
<td>8</td>
<td>number of keys in the data portion</td>
</tr>
<tr>
<td>9</td>
<td>1-based position of the column header</td>
</tr>
<tr>
<td>10</td>
<td>1-based position of the data portion</td>
</tr>
<tr>
<td>11</td>
<td>number of repeat groups</td>
</tr>
<tr>
<td>12</td>
<td>size of the repeat group</td>
</tr>
<tr>
<td>13</td>
<td>starting position of the repeat group</td>
</tr>
<tr>
<td>14</td>
<td>missing data code</td>
</tr>
<tr>
<td>15</td>
<td>integer ID of the file</td>
</tr>
<tr>
<td>16 - 23</td>
<td>text ID of the file</td>
</tr>
<tr>
<td>24</td>
<td>creator's project number</td>
</tr>
<tr>
<td>25</td>
<td>creation date, yyyyddd</td>
</tr>
<tr>
<td>26</td>
<td>creator's ID</td>
</tr>
<tr>
<td>27</td>
<td>zero-based offset to the row header</td>
</tr>
<tr>
<td>28</td>
<td>zero-based offset to the column header</td>
</tr>
<tr>
<td>29</td>
<td>zero-based offset to the data portion</td>
</tr>
<tr>
<td>30</td>
<td>first unused word in the file</td>
</tr>
<tr>
<td>31</td>
<td>start of the user record</td>
</tr>
<tr>
<td>32</td>
<td>start of the key names</td>
</tr>
<tr>
<td>33</td>
<td>start of the scale factors</td>
</tr>
<tr>
<td>34</td>
<td>start of the units</td>
</tr>
<tr>
<td>35 - 38</td>
<td>reserved</td>
</tr>
<tr>
<td>39</td>
<td>beginning Julian day of the data, ccyyddd</td>
</tr>
<tr>
<td>40</td>
<td>beginning time of the data, hhmmss</td>
</tr>
<tr>
<td>41</td>
<td>ending Julian day of the data, ccyyddd</td>
</tr>
<tr>
<td>42</td>
<td>ending time of the data, hhmmss</td>
</tr>
<tr>
<td>43 - 60</td>
<td>reserved</td>
</tr>
<tr>
<td>61 - 62</td>
<td>file name</td>
</tr>
<tr>
<td>63</td>
<td>MD file number</td>
</tr>
<tr>
<td>64 - 463</td>
<td>user record, MD coordinates (0,0); not described by the schema; use for storing arbitrary information</td>
</tr>
<tr>
<td>464 - 863</td>
<td>names of the file keys</td>
</tr>
<tr>
<td>864 - 1263</td>
<td>scale factors for the keys</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>1264 - 1663</td>
<td>units of the keys</td>
</tr>
<tr>
<td>1664 - 4095</td>
<td>reserved</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m0pthdr</td>
<td>opens a connection to read a point source data header</td>
</tr>
<tr>
<td>m0ptrhdr</td>
<td>reads the point source data header</td>
</tr>
<tr>
<td>m0ptget</td>
<td>opens a connection to read point source data</td>
</tr>
<tr>
<td>m0ptrd</td>
<td>reads point source data</td>
</tr>
<tr>
<td>mdo</td>
<td>writes point source data</td>
</tr>
<tr>
<td>mdquit</td>
<td>deletes a point source file</td>
</tr>
<tr>
<td>m0psort</td>
<td>gets point source selection parameters from the command line</td>
</tr>
</tbody>
</table>
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 and -OS2.

**Word allocation for record 1**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>total number of records in the file</td>
</tr>
<tr>
<td>1 - 47</td>
<td>unused</td>
</tr>
</tbody>
</table>

**Word allocation for records 2 through n**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 3</td>
<td>resistance-to-temperature coefficients, zeroth order; if word 0 is the missing value, the record was deleted</td>
</tr>
<tr>
<td>4 - 7</td>
<td>resistance-to-temperature coefficients, first order</td>
</tr>
<tr>
<td>8 - 11</td>
<td>resistance-to-temperature coefficients, second order</td>
</tr>
<tr>
<td>12 - 23</td>
<td>coefficients to linearize MSU counts</td>
</tr>
<tr>
<td>24 - 27</td>
<td>high and low calibration points for electronic systems A and B</td>
</tr>
<tr>
<td>28 - 35</td>
<td>average space and target temperatures for channels 1 to 4</td>
</tr>
<tr>
<td>36 - 43</td>
<td>standard deviations of space and target temperatures for channels 1 to 4</td>
</tr>
<tr>
<td>44</td>
<td>nominal target temperature</td>
</tr>
<tr>
<td>45</td>
<td>MSU serial number</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>46</td>
<td>NOAA satellite number</td>
</tr>
<tr>
<td>47</td>
<td>date, yyyddd</td>
</tr>
</tbody>
</table>

**API functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msucon</td>
<td>initializes the M$U$ constants</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Map file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTLSUPU</td>
<td>high resolution USA</td>
</tr>
<tr>
<td>OUTLSUPW</td>
<td>world coastal</td>
</tr>
<tr>
<td>OUTLUSAM</td>
<td>medium resolution USA</td>
</tr>
<tr>
<td>OUTLHPOL</td>
<td>world political boundaries</td>
</tr>
<tr>
<td>OUTLHRES</td>
<td>high resolution world coastal outline</td>
</tr>
<tr>
<td>OUTLUSAL</td>
<td>low resolution USA and North America</td>
</tr>
<tr>
<td>OUTLWRLD</td>
<td>world coastal</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>number of blocks (line segments)</td>
</tr>
<tr>
<td>1</td>
<td>minimum latitude, degrees * 10000</td>
</tr>
<tr>
<td>2</td>
<td>maximum latitude, degrees * 10000</td>
</tr>
<tr>
<td>3</td>
<td>minimum longitude, degrees * 10000; west is positive</td>
</tr>
<tr>
<td>4</td>
<td>maximum longitude, degrees * 10000; west is positive</td>
</tr>
<tr>
<td>5</td>
<td>beginning word of data start for segment 1</td>
</tr>
<tr>
<td>6</td>
<td>number of words to read</td>
</tr>
<tr>
<td>7 - 12</td>
<td>directory for line segment 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5995 - 6000</td>
<td>directory for line segment 1000</td>
</tr>
<tr>
<td>6001</td>
<td>latitude degrees * 10000</td>
</tr>
<tr>
<td>6002</td>
<td>longitude degrees * 10000</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------</td>
</tr>
<tr>
<td>6003</td>
<td>latitude degrees × 10000</td>
</tr>
<tr>
<td>6004</td>
<td>longitude degrees × 10000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0...n-1</td>
<td>(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</td>
</tr>
<tr>
<td>(n + 1)</td>
<td>end of file marker; HEX FFFF0000</td>
</tr>
</tbody>
</table>

**API functions**

Currently, no API library functions exist for reading and writing base map files.
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:

- a satellite name, in columns 1 to 19
- a band specifier code, if applicable, in column 20
- a sensor source number, in columns 30 and 31

File size depends on the number of satellite description entries and is computed by \((20 \times \text{number of entries})\) words. For example, two entries may look like this:

<table>
<thead>
<tr>
<th>Columns 1 to 19</th>
<th>Column 20</th>
<th>Columns 30 and 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOES-7 IR</td>
<td>D</td>
<td>33</td>
</tr>
<tr>
<td>NOAA-10</td>
<td>K</td>
<td>60</td>
</tr>
</tbody>
</table>

This file is supplied with McIDAS-X and -OS2.

API functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mOssdesc</td>
<td>retrieves the satellite sensor information</td>
</tr>
</tbody>
</table>
SKEDFILE

Scheduled McIDAS 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 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.

**Word allocation for the file**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 7</td>
<td>unused</td>
</tr>
<tr>
<td>8 - 807</td>
<td>first section of scheduler entry information</td>
</tr>
<tr>
<td>808 - 871</td>
<td>unused</td>
</tr>
<tr>
<td>872 - 7171</td>
<td>second section of scheduler entry information</td>
</tr>
</tbody>
</table>

**Word allocation for the first section**

This section contains eight words for each entry (maximum of 100).

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>time of the next scheduled run; all times are kept internally, in seconds, since 1 January 1972</td>
</tr>
<tr>
<td>1</td>
<td>number of runs remaining</td>
</tr>
<tr>
<td>2</td>
<td>interval between runs</td>
</tr>
<tr>
<td>3</td>
<td>late tolerance if the scheduler is delayed</td>
</tr>
<tr>
<td>4</td>
<td>terminal at which the command runs</td>
</tr>
<tr>
<td>5</td>
<td>ID number or name assigned when the command is entered in the schedule</td>
</tr>
</tbody>
</table>
Word allocation for the second section
This section contains 64 words for each entry.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>time that the command first runs</td>
</tr>
<tr>
<td>1</td>
<td>total number of times the command runs</td>
</tr>
<tr>
<td>2</td>
<td>unused</td>
</tr>
<tr>
<td>3 - 63</td>
<td>text of the McIDAS command</td>
</tr>
</tbody>
</table>

API functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>skio</td>
<td>schedule file I/O routines</td>
</tr>
</tbody>
</table>
User Common is a block of data stored in shared memory. Its contents are lost if McIDAS stops; otherwise, it retains values that allow for interprocess communication of information usually related to the state of the McIDAS 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.

**Word allocation**

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-123 to -120</td>
<td>current font information</td>
</tr>
<tr>
<td>-82</td>
<td>logon initialization flag; 1 = 1 option used by PC</td>
</tr>
<tr>
<td>-58</td>
<td>ending column number in the current MD wind file</td>
</tr>
<tr>
<td>-57</td>
<td>ending row number in the current MD wind file</td>
</tr>
<tr>
<td>-56</td>
<td>starting row number in the current MD wind file</td>
</tr>
<tr>
<td>-55</td>
<td>current MD wind file used for core output</td>
</tr>
<tr>
<td>-54</td>
<td>current MD wind file used for imv, selector and core output</td>
</tr>
<tr>
<td>-53</td>
<td>element size of the target cursor</td>
</tr>
<tr>
<td>-52</td>
<td>line size of the target cursor</td>
</tr>
<tr>
<td>-51</td>
<td>space bar toggle</td>
</tr>
<tr>
<td>-50</td>
<td>error status word</td>
</tr>
<tr>
<td>-46</td>
<td>current TCOL</td>
</tr>
<tr>
<td>-45</td>
<td>current TWIN</td>
</tr>
<tr>
<td>-44</td>
<td>second cursor state control word; bits 0-2 are not used; bits 3-5 are cursor color; bits 6-8 are cursor type</td>
</tr>
<tr>
<td>-43</td>
<td>second cursor element position</td>
</tr>
<tr>
<td>-42</td>
<td>second cursor line position</td>
</tr>
<tr>
<td>-41</td>
<td>second cursor element size</td>
</tr>
<tr>
<td>-40</td>
<td>second cursor line size</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>-38</td>
<td>next record pointer for the file created with DEV=F</td>
</tr>
<tr>
<td>-37 to -35</td>
<td>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</td>
</tr>
<tr>
<td>-34</td>
<td>current graphics virtual frame; if less than zero, doesn’t plot</td>
</tr>
<tr>
<td>-33</td>
<td>current diagnostic message (ddest) device: 0 = suppressed; 1 = terminal screen; 2 = local printer; 3 = system printer</td>
</tr>
<tr>
<td>-32</td>
<td>current error message (edest) device: 0 = suppressed; 1 = terminal screen; 2 = local printer; 3 = system printer</td>
</tr>
<tr>
<td>-31</td>
<td>device for standard text messages: 0 = suppressed; 1 = terminal screen; 2 = local printer; 3 = system printer</td>
</tr>
<tr>
<td>-30</td>
<td>auto context table search: 0 = no keyword substitution; 1 = parameter fetching subroutines; for example, CKWP, resort to system string table for missing keyword parameters</td>
</tr>
<tr>
<td>-26</td>
<td>0 = program currently running is not a macro 1 = program currently running is a macro</td>
</tr>
<tr>
<td>-25</td>
<td>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</td>
</tr>
<tr>
<td>-24</td>
<td>current level when starting another process: 0 = scanner; 1 = next program, etc.</td>
</tr>
<tr>
<td>-23</td>
<td>number of this initiator</td>
</tr>
<tr>
<td>-22</td>
<td>0 = command was not started by the scheduler 1 = command was started by the scheduler</td>
</tr>
<tr>
<td>-21</td>
<td>abort signal handling; see word 455; McIDAS-X only</td>
</tr>
<tr>
<td>-16</td>
<td>project number the command runs under; may be different from the logged on project number in UC1(1)</td>
</tr>
<tr>
<td>-13</td>
<td>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)</td>
</tr>
<tr>
<td>-12</td>
<td>cursor element position</td>
</tr>
<tr>
<td>-11</td>
<td>cursor line position</td>
</tr>
<tr>
<td>-10</td>
<td>cursor element size</td>
</tr>
<tr>
<td>-9</td>
<td>cursor line size</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -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 UCI(-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 dddmss;  
1 = lat/lon are displayed in decimal |
<p>| 20   | set to 1 by the G key |
| 21   | set to 1 by the Q key |
| 22   | import pointer |
| 37   | image display hardware: 0 = tower; 1 = WIDE WORD; 3 = VGA; 4 = MODE-13; 6 = X Windows |</p>
<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>graphics state flag: 0 = no graphics; 1 = draws graphics</td>
</tr>
<tr>
<td>39</td>
<td>virtual graphics state flag: 0 = doesn’t write virtual graphics; nonzero = virtual frame to write</td>
</tr>
<tr>
<td>46</td>
<td>default graphics line width</td>
</tr>
<tr>
<td>47</td>
<td>graphics dash length, in pixels</td>
</tr>
<tr>
<td>48</td>
<td>graphics dash gap length, in pixels</td>
</tr>
<tr>
<td>49</td>
<td>graphics dash gap color</td>
</tr>
<tr>
<td>50</td>
<td>loop control system; consists of the LS command and the A, B, J, Y, O and I 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</td>
</tr>
<tr>
<td>51</td>
<td>current image frame</td>
</tr>
<tr>
<td>52</td>
<td>image frame loop, upper bound</td>
</tr>
<tr>
<td>53</td>
<td>image frame loop, lower bound</td>
</tr>
<tr>
<td>54</td>
<td>1 = image frames are connected to the loop control; 0 = image frames aren’t connected</td>
</tr>
<tr>
<td>55</td>
<td>1 = image frames are visible; 0 = image frames are blanked (Alt K)</td>
</tr>
<tr>
<td>56</td>
<td>current graphics frame</td>
</tr>
<tr>
<td>57</td>
<td>graphics frame loop, upper bound</td>
</tr>
<tr>
<td>58</td>
<td>graphics frame loop, lower bound</td>
</tr>
<tr>
<td>59</td>
<td>1 = graphics frames connected to loop control; 0 = graphics frames not connected</td>
</tr>
<tr>
<td>60</td>
<td>1 = graphics frames are visible; 0 = graphics frames are blanked (Alt W)</td>
</tr>
<tr>
<td>61</td>
<td>cursor size, vertical</td>
</tr>
<tr>
<td>62</td>
<td>cursor size, horizontal</td>
</tr>
<tr>
<td>63</td>
<td>cursor position, line number</td>
</tr>
<tr>
<td>64</td>
<td>cursor position, element number</td>
</tr>
<tr>
<td>65</td>
<td>cursor type: 1 = box; 2 = crosshair; 3 = box and crosshair; 4 = solid box; 5 = star wars</td>
</tr>
<tr>
<td>66</td>
<td>cursor color (0 - 7)</td>
</tr>
<tr>
<td>67</td>
<td>joystick: 0 = disconnected; 1 = controls cursor position; 2 = vernier cursor control; 3 = controls cursor size; 4 = velocity cursor</td>
</tr>
<tr>
<td>69</td>
<td>second cursor size, vertical</td>
</tr>
<tr>
<td>70</td>
<td>second cursor size, horizontal</td>
</tr>
<tr>
<td>71</td>
<td>second cursor position, line number</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>72</td>
<td>second cursor position, element number</td>
</tr>
<tr>
<td>73</td>
<td>second cursor type; see word 65</td>
</tr>
<tr>
<td>74</td>
<td>second cursor color; see word 66</td>
</tr>
<tr>
<td>75</td>
<td>cursor mode: 0 = single cursor mode; 1 = dual cursor mode</td>
</tr>
<tr>
<td>76</td>
<td>saved cursor type; set when word 65 = 0 to turn off the cursor</td>
</tr>
<tr>
<td>77 - 79</td>
<td>reserved</td>
</tr>
<tr>
<td>87</td>
<td>McBASI list window size</td>
</tr>
<tr>
<td>88</td>
<td>McBASI auto-line number line number</td>
</tr>
<tr>
<td>89</td>
<td>McBASI run abort flag</td>
</tr>
<tr>
<td>98</td>
<td>current MD core output file for motion vectors (WINDCO)</td>
</tr>
<tr>
<td>99</td>
<td>nortle toggle</td>
</tr>
<tr>
<td>100</td>
<td>current MD selector file for motion vectors (WINDCO)</td>
</tr>
<tr>
<td>121</td>
<td>single-letter command; otherwise, zero</td>
</tr>
<tr>
<td>122</td>
<td>used by the scheduler (sked); last time through all entries flag</td>
</tr>
<tr>
<td>158</td>
<td>vdisk drive letter</td>
</tr>
<tr>
<td>160</td>
<td>TCP/IP address of session partner</td>
</tr>
<tr>
<td>161</td>
<td>TCP rate measurement word</td>
</tr>
<tr>
<td>162</td>
<td>TCP connection status code</td>
</tr>
<tr>
<td>163</td>
<td>TCP reserved</td>
</tr>
<tr>
<td>164</td>
<td>WINDCO velocity cursor toggle set by Alt V: 0 = off; 1 = on</td>
</tr>
<tr>
<td>165</td>
<td>WINDCO sampling flag set by Alt X</td>
</tr>
<tr>
<td>166</td>
<td>WINDCO mode is active</td>
</tr>
<tr>
<td>167</td>
<td>mode switch for VGA; set by VGA.PGM, SCRCTL.PGM and KBDCTL.PGM</td>
</tr>
</tbody>
</table>
| 168  | 0 = asynchronous communication is OK  
 1 = async comm is suspended, port available |
<p>| 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  | VGA frame save flag: 0 = save frames; 1 = don't save them |
| 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 |</p>
<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>switch to tell mousef to ignore mouse button press</td>
</tr>
<tr>
<td>185</td>
<td>mouse button status; two 2-byte integer (I*2) values that correspond to mouse buttons 1 and 2 respectively</td>
</tr>
<tr>
<td>186</td>
<td>mouse movement values (mickies); two 2-byte integer (I*2) values that correspond to movement up/down and left/right respectively</td>
</tr>
<tr>
<td>187</td>
<td>switch to tell mousef to set mickies into UC(186)</td>
</tr>
<tr>
<td>188</td>
<td>image nortle toggle</td>
</tr>
<tr>
<td>189</td>
<td>1 = commands go to the host</td>
</tr>
<tr>
<td>190</td>
<td>outgoing spool-in-use flag</td>
</tr>
<tr>
<td>191</td>
<td>mouse PID so EXIT can kill it</td>
</tr>
<tr>
<td>192</td>
<td>outgoing message request pointer</td>
</tr>
<tr>
<td>193</td>
<td>outgoing message completed pointer</td>
</tr>
<tr>
<td>194</td>
<td>1 = system shutdown request</td>
</tr>
<tr>
<td>195</td>
<td>last text window displayed; see UC(200)</td>
</tr>
<tr>
<td>196</td>
<td>count of text waiting to be displayed</td>
</tr>
<tr>
<td>197</td>
<td>import pointer from the decoder</td>
</tr>
<tr>
<td>198</td>
<td>communications port name, binary 0 for ProNET</td>
</tr>
<tr>
<td>199</td>
<td>last written pointer for import decoder</td>
</tr>
<tr>
<td>200</td>
<td>text window to display; on displays where the DISP_TOGGLE terminal characteristic is true (VGA, for example), you can see either the text or the image but not both; -1 = the user sees the image and the current text window number is saved in UC(195)</td>
</tr>
<tr>
<td>201</td>
<td>first line in the text window to display; the range is 1 to 57; McIDAS-OS2 only; value is ignored on windows 5 through 9</td>
</tr>
<tr>
<td>202-203</td>
<td>mouse RAW position</td>
</tr>
<tr>
<td>204</td>
<td>last frame displayed (VGA only); 0 = forces redisplay of the current frame</td>
</tr>
<tr>
<td>205</td>
<td>switch to tell kbdctl to pass keystrokes to an ASK process; McIDAS-OS2 only</td>
</tr>
<tr>
<td>206</td>
<td>place where kbdctl pokes characters when UC(205) is set; McIDAS-OS2 only</td>
</tr>
<tr>
<td>207</td>
<td>nonzero means kbdctl is in insert mode</td>
</tr>
<tr>
<td>208</td>
<td>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</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>209</td>
<td>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</td>
</tr>
<tr>
<td>210</td>
<td>number of memory segments supporting frames for VGA</td>
</tr>
<tr>
<td>211+</td>
<td>addresses of memory segments supporting frames for VGA display; four segments per frame</td>
</tr>
<tr>
<td>215+</td>
<td>WIDE WORD addresses of memory segments for enhancements and graphics (16 segments)</td>
</tr>
<tr>
<td>300</td>
<td>image zoom factor for WIDE WORD; in McIDAS-X, the zoom toggle, Alt Z</td>
</tr>
<tr>
<td>301</td>
<td>secondary display (WIDE WORD only): 0 = opposite frame; &gt; 0 = image frame; &lt; 0 = graphics frame</td>
</tr>
<tr>
<td>302</td>
<td>WIDE WORD handler lock out request used by tvsync; 0 = not locked; otherwise locked</td>
</tr>
<tr>
<td>303</td>
<td>WIDE WORD briefing frame threshold, nonzero</td>
</tr>
<tr>
<td>304</td>
<td>last character from WIDE WORD briefing frame control port</td>
</tr>
<tr>
<td>305</td>
<td>cursor control for the WIDE WORD: 0 = cursor on center; 1 = cursor dragged to edge</td>
</tr>
<tr>
<td>315 - 350</td>
<td>briefing port, frame controls and movie flag for PM and McIDAS-X displays</td>
</tr>
<tr>
<td>400</td>
<td>number of text screens in use</td>
</tr>
<tr>
<td>401</td>
<td>maximum number of lines in text scrolling; McIDAS-OS2 only</td>
</tr>
<tr>
<td>451 - 453</td>
<td>McIDAS-X cursor color (red, green, blue); the range is 0 to 255</td>
</tr>
<tr>
<td>455</td>
<td>1 = traps abort signals; see UC(-21); McIDAS-X only</td>
</tr>
<tr>
<td>456</td>
<td>ID of the command window; McIDAS-X only</td>
</tr>
<tr>
<td>457</td>
<td>ID of the image window; McIDAS-X only</td>
</tr>
<tr>
<td>465</td>
<td>offset from the start of UC to the start of redirect memory</td>
</tr>
<tr>
<td>467</td>
<td>1 = independent graphics; McIDAS-X only</td>
</tr>
<tr>
<td>470</td>
<td>1 = to use fonts</td>
</tr>
<tr>
<td>471 - 480</td>
<td>family name of the font</td>
</tr>
<tr>
<td>500</td>
<td>number of graphics levels; McIDAS-X only</td>
</tr>
<tr>
<td>501</td>
<td>interpolation strategy; McIDAS-X and PM only</td>
</tr>
<tr>
<td>502</td>
<td>1 = image window should resize to the size of the current frame (Alt R); McIDAS-X only</td>
</tr>
<tr>
<td>503</td>
<td>zoom factor; used when UC(300) is one; McIDAS-X only</td>
</tr>
<tr>
<td>504</td>
<td>mcimage process ID; McIDAS-X only</td>
</tr>
<tr>
<td>Word</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>600</td>
<td>number of gray levels; McIDAS-X only</td>
</tr>
<tr>
<td>700</td>
<td>TCPIP module loaded; McIDAS-X only</td>
</tr>
<tr>
<td>701</td>
<td>IMPORT module loaded; McIDAS-X only</td>
</tr>
<tr>
<td>1000 - 1999</td>
<td>user/site defined</td>
</tr>
<tr>
<td>2000</td>
<td>if nonzero, the image window should raise itself and reset the value to zero</td>
</tr>
<tr>
<td>2001+</td>
<td>frame allocation pointers for WIDE WORD and PM displays (image frames start at 2001 and increase; graphics frames start at 2999 and decrease); frame object offsets from the start of UC for X Windows</td>
</tr>
<tr>
<td>3001+</td>
<td>terminal types with variable frame sizes; line and element sizes as halfwords</td>
</tr>
<tr>
<td>4000</td>
<td>relative pointer for random looping</td>
</tr>
<tr>
<td>4001+</td>
<td>list for random looping, where zero is the end</td>
</tr>
<tr>
<td>5001+</td>
<td>list of dwell rates</td>
</tr>
<tr>
<td>6000</td>
<td>relative pointer for random graphics loops</td>
</tr>
<tr>
<td>6001+</td>
<td>list for random graphics loop, where zero is the end</td>
</tr>
<tr>
<td>7001+</td>
<td>list of random opposites, where zero is the end</td>
</tr>
<tr>
<td>16383 -</td>
<td>end of segment</td>
</tr>
</tbody>
</table>

**API functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>luc</td>
<td>reads a word from User Common</td>
</tr>
<tr>
<td>puc</td>
<td>writes a word to User Common</td>
</tr>
<tr>
<td>mcmoubtn</td>
<td>reports the state of the mouse buttons as soon as a mouse button event occurs</td>
</tr>
<tr>
<td>Mcluc</td>
<td>returns a value from User Common</td>
</tr>
<tr>
<td>Mcpuc</td>
<td>changes a value in User Common</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 540671</td>
<td>radiance calibration block (watts/meter**2/steradian*1000)</td>
</tr>
<tr>
<td>540672 - 1081343</td>
<td>temperature calibration block, in Kelvin</td>
</tr>
<tr>
<td>1081344 - 1622015</td>
<td>brightness calibration block, 0 to 255</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 45055</td>
<td>band 1 block</td>
</tr>
<tr>
<td>45056 - 90111</td>
<td>band 2 block</td>
</tr>
<tr>
<td>90112 - 135167</td>
<td>band 3 block</td>
</tr>
<tr>
<td>135168 - 180223</td>
<td>band 4 block</td>
</tr>
<tr>
<td>180224 - 225279</td>
<td>band 5 block</td>
</tr>
<tr>
<td>225280 - 270335</td>
<td>band 6 block</td>
</tr>
<tr>
<td>270336 - 315391</td>
<td>band 7 block</td>
</tr>
<tr>
<td>315392 - 360447</td>
<td>band 8 block</td>
</tr>
<tr>
<td>360448 - 405503</td>
<td>band 9 block</td>
</tr>
<tr>
<td>405504 - 450559</td>
<td>band 10 block</td>
</tr>
<tr>
<td>450560 - 495615</td>
<td>band 11 block</td>
</tr>
<tr>
<td>495616 - 540671</td>
<td>band 12 block</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 4095</td>
<td>Delta-F = -5</td>
</tr>
<tr>
<td>4096 - 8191</td>
<td>Delta-F = -4</td>
</tr>
<tr>
<td>8192 - 12287</td>
<td>Delta-F = -3</td>
</tr>
<tr>
<td>12288 - 16383</td>
<td>Delta-F = -2</td>
</tr>
<tr>
<td>16384 - 20479</td>
<td>Delta-F = -1</td>
</tr>
<tr>
<td>20480 - 24575</td>
<td>Delta-F = 0</td>
</tr>
<tr>
<td>24576 - 28671</td>
<td>Delta-F = +1</td>
</tr>
<tr>
<td>28672 - 32767</td>
<td>Delta-F = +2</td>
</tr>
<tr>
<td>32768 - 36863</td>
<td>Delta-F = +3</td>
</tr>
<tr>
<td>36864 - 40959</td>
<td>Delta-F = +4</td>
</tr>
<tr>
<td>40960 - 45055</td>
<td>Delta-F = +5</td>
</tr>
</tbody>
</table>

API functions

Currently, no API library functions exist for reading and writing this file.
**VIRTnnnn**

*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.

**Word allocation**
Below is a description of one scene in a virtual graphics file.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>top of the page boundary and virtual graphics version as halfwords</td>
</tr>
<tr>
<td>1</td>
<td>bottom of the page boundary and line width as halfwords</td>
</tr>
<tr>
<td>2</td>
<td>left edge of the page boundary</td>
</tr>
<tr>
<td>3</td>
<td>right edge of the page boundary</td>
</tr>
<tr>
<td>4</td>
<td>beginning pen position/color triples where (in an x-y plane):</td>
</tr>
<tr>
<td></td>
<td>word n = pen y position</td>
</tr>
<tr>
<td></td>
<td>word n + 1 = pen x position</td>
</tr>
<tr>
<td></td>
<td>word n + 2 = pen color</td>
</tr>
<tr>
<td></td>
<td>Decimal 256 marks the end of a scene.</td>
</tr>
</tbody>
</table>

**API functions**

<table>
<thead>
<tr>
<th>Fortran Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpout</td>
<td>writes to a virtual graphics file</td>
</tr>
</tbody>
</table>
Writing ADDE Servers

This chapter describes the procedures that you will use to build data servers for McIDAS ADDE applications. You’ll learn:

♦ information about McIDAS servers that you should know before you begin writing a server

♦ the steps for creating a server and the McIDAS library functions that you will use

♦ how to debug your server

♦ the request syntax and transmission format for each McIDAS data type

Contents

<table>
<thead>
<tr>
<th>Overview</th>
<th>8-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting started</td>
<td>8-4</td>
</tr>
<tr>
<td>Client and server communication</td>
<td>8-4</td>
</tr>
<tr>
<td>Rules for transmitting data</td>
<td>8-5</td>
</tr>
<tr>
<td>Primary servers</td>
<td>8-5</td>
</tr>
<tr>
<td>Secondary servers</td>
<td>8-7</td>
</tr>
<tr>
<td>Sample data request</td>
<td>8-8</td>
</tr>
</tbody>
</table>
Contents

Building your ADDE server .................................. 8-10
  Initializing the McIDAS file system .......................... 8-12
  Reading the ADDE client request ............................. 8-12
  Reading the server mapping table ............................ 8-15
  Interpreting the client request .............................. 8-16
  Retrieving the requested data from disk .................... 8-17
  Sending the data to the client ............................... 8-17
  Ending the transaction ...................................... 8-19

Debugging servers ............................................ 8-21

Request syntax and data transmission formats ................. 8-23
  Serving image data .......................................... 8-25
  Serving image directory data ................................ 8-31
  Serving grid data ........................................... 8-34
  Serving grid directories ..................................... 8-38
  Serving point data .......................................... 8-42
  Serving weather text data ................................... 8-45
  Serving observational weather text data .................... 8-49
Overview

ADDE (Abstract Data Distribution Environment) is the client/server mechanism for distributing data in McIDAS. 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 have?

McIDAS 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 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. This section describes the following:

- how the client and server communicate
- the rules for transmitting data between a client and server
- the role of the primary server
- why McIDAS also has secondary servers

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.
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 Unix platforms currently supported on McIDAS are big-endian machines. McIDAS-OS2 runs on Intel-based machines, which use the little-endian integer representation. 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 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. The table below lists the current primary servers provided in McIDAS.

<table>
<thead>
<tr>
<th>Client APIs</th>
<th>Client request type</th>
<th>Data type</th>
<th>Server name</th>
<th>Description</th>
<th>Secondary server suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcdel</td>
<td>ADEL</td>
<td>image</td>
<td>adelserv</td>
<td>deletes images from a dataset</td>
<td></td>
</tr>
<tr>
<td>mcdir</td>
<td>ADIR</td>
<td>image</td>
<td>adirserv</td>
<td>retrieves image header information</td>
<td>ADIR</td>
</tr>
<tr>
<td>mcdard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcaget</td>
<td>AGET</td>
<td>image</td>
<td>agetserv</td>
<td>retrieves the image header, navigation, calibration and data; data is returned line by line</td>
<td>AGET</td>
</tr>
<tr>
<td>mcalin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcapfx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcanav</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcacal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcacrd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client APIs</td>
<td>Client request type</td>
<td>Data type</td>
<td>Server name</td>
<td>Description</td>
<td>Secondary server suffix</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>--------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>mcaput</td>
<td>APUT</td>
<td>image</td>
<td>aputserv</td>
<td>writes an image object to a dataset</td>
<td></td>
</tr>
<tr>
<td>mcaput</td>
<td>ATOK</td>
<td>image</td>
<td>atokserv</td>
<td>checks file permissions for writing image objects</td>
<td></td>
</tr>
<tr>
<td>mcgdir</td>
<td>GDIR</td>
<td>grid</td>
<td>gdirserv</td>
<td>retrieves grid header information</td>
<td>GDIR</td>
</tr>
<tr>
<td>mcgdirf</td>
<td>GGET</td>
<td>grid</td>
<td>ggetserv</td>
<td>retrieves the grid header and entire grid</td>
<td>GGET</td>
</tr>
<tr>
<td>mcgput</td>
<td>GPUT</td>
<td>grid</td>
<td>gputserv</td>
<td>writes a grid object to a dataset</td>
<td></td>
</tr>
<tr>
<td>m0pthdr</td>
<td>MDFH</td>
<td>point</td>
<td>mdfhserv</td>
<td>retrieves point file header information</td>
<td></td>
</tr>
<tr>
<td>m0pthdrd</td>
<td>MDHD</td>
<td>point</td>
<td>mdhdserv</td>
<td>retrieves point header information</td>
<td></td>
</tr>
<tr>
<td>m0pthrd</td>
<td>MDKS</td>
<td>point</td>
<td>mdksserv</td>
<td>retrieves the point header and data</td>
<td>KS</td>
</tr>
<tr>
<td>m0navget</td>
<td>NAVG</td>
<td>nav</td>
<td>navgserv</td>
<td>retrieves satellite navigation</td>
<td></td>
</tr>
<tr>
<td>M0obtxget</td>
<td>OBTG</td>
<td>text</td>
<td>obtgserv</td>
<td>retrieves observational weather text</td>
<td>OBTG</td>
</tr>
<tr>
<td>M0textget</td>
<td>TXTG</td>
<td>text</td>
<td>txtgserv</td>
<td>retrieves an ASCII text file</td>
<td></td>
</tr>
<tr>
<td>M0txtread</td>
<td>WTXG</td>
<td>text</td>
<td>wtxgserv</td>
<td>retrieves textual weather information</td>
<td>WTXG</td>
</tr>
</tbody>
</table>
The ADDE communications module, **meserv**, starts the primary servers based on the server’s IP address, which tells **meserv** if the request will be fulfilled locally or remotely. If the request is handled locally, **meserv** 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, **meserv** 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 **meserv** 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 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

Most servers written by developers outside of SSEC are secondary servers.
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:

```
DSERVE ADD ETA/00 GRID 101 110 TYPE=GRID "002 ETA Model Run
DSERVE ADD LOCAL/MODEL FLAT TYPE=GRID "Local Model Grids"
```

The first entry created the grid dataset ETA/00, which is stored in the standard McIDAS 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 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 ETA/00
```

1. `meserv` starts the primary server `ggetserv`.

2. `ggetserv` reads the server mapping table entry for the dataset ETA/00.

3. Because the grids from ETA/00 are in the standard McIDAS 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 format called FLAT, `ggetserv` starts the secondary server `flatget`.

4. `flatget` 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 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.

The McIDAS library functions that you will use when creating an ADDE server are listed alphabetically in the table below.

<table>
<thead>
<tr>
<th>C function</th>
<th>Fortran function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initblok</td>
<td>m0isrtdataset</td>
<td>initializes the McIDAS environment for the server</td>
</tr>
<tr>
<td>M0isrtdataset</td>
<td>m0isrtdataset</td>
<td>determines if the dataset requested was flagged as a real-time dataset</td>
</tr>
<tr>
<td>M0isTraceSet</td>
<td>m0istraceset</td>
<td>given a client request, activates tracing if TRACE = nonzero value</td>
</tr>
<tr>
<td>M0swbyt4</td>
<td>swbyt4</td>
<td>conditionally flips 4-byte data buffers</td>
</tr>
<tr>
<td>M0swbyt2</td>
<td>swbyt2</td>
<td>conditionally flips 2-byte data buffers</td>
</tr>
<tr>
<td>M0sxdatasetinfo</td>
<td>m0xsxdatasetinfo</td>
<td>extracts the server mapping table information</td>
</tr>
<tr>
<td>M0sxdone</td>
<td>m0sxdone</td>
<td>ends the ADDE transaction</td>
</tr>
<tr>
<td>C function</td>
<td>Fortran function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MOSxGetClientRequest</td>
<td>MOSxgetclientrequest</td>
<td>reads the request header and request string sent from the client</td>
</tr>
<tr>
<td>MOSxRead</td>
<td>MOSxread</td>
<td>reads data sent from the client to the server</td>
</tr>
<tr>
<td></td>
<td>MOSxresolv</td>
<td>appends information from the server mapping table to the client request</td>
</tr>
<tr>
<td>MOSxsend</td>
<td>MOSxsend</td>
<td>sends data from the server to the client</td>
</tr>
<tr>
<td>MOSxSetTraceOff</td>
<td>MOSxxsettraceoff</td>
<td>turns tracing off</td>
</tr>
<tr>
<td>MOSxSetTraceOn</td>
<td>MOSxxsettraceon</td>
<td>turns tracing on</td>
</tr>
<tr>
<td>MOSxtrce</td>
<td>MOSxtrce</td>
<td>writes tracing information to a trace file</td>
</tr>
<tr>
<td>M0InitLocalServer</td>
<td></td>
<td>initializes the environment for secondary servers</td>
</tr>
<tr>
<td>Mtrace</td>
<td>mtrace</td>
<td>writes tracing information to a trace file</td>
</tr>
</tbody>
</table>
Initializing the McIDAS file system

Your server must be able to access McIDAS disk files, since most servers are built using disk file utilities. Use the `initblok` function to initialize the McIDAS 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:

```fortran
integer*2 init_stat
ok = initblok (init_stat)
```

In C:

```c
short init_stat;
ok = initblok_(&init_stat);
```

For more information about disk files, see the section titled *McIDAS disk files* in Chapter 6, *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.
<table>
<thead>
<tr>
<th>Request header components</th>
<th>Length, in bytes</th>
<th>Word number in Fortran</th>
<th>servacct structure names in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>server IP address</td>
<td>4</td>
<td>1</td>
<td>server_address</td>
</tr>
<tr>
<td>server port</td>
<td>4</td>
<td>2</td>
<td>server_port</td>
</tr>
<tr>
<td>client IP address</td>
<td>4</td>
<td>3</td>
<td>client_address</td>
</tr>
<tr>
<td>user initials</td>
<td>4 (ASCII)</td>
<td>4</td>
<td>user</td>
</tr>
<tr>
<td>project number</td>
<td>4</td>
<td>5</td>
<td>project</td>
</tr>
<tr>
<td>password</td>
<td>12 (ASCII)</td>
<td>6 - 8</td>
<td>password</td>
</tr>
<tr>
<td>service name</td>
<td>4 (ASCII)</td>
<td>9</td>
<td>transaction</td>
</tr>
<tr>
<td>input data length</td>
<td>4</td>
<td>10</td>
<td>input_length in excess of 160 bytes</td>
</tr>
<tr>
<td>request string</td>
<td>120 (ASCII)</td>
<td>11 - 40</td>
<td>text</td>
</tr>
</tbody>
</table>

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:

```fortran
include 'fileparm.inc'
character*(MAXPATHLENGTH) request
integer request_block(64)
:
:
ok = m0sxgetclientrequest(request_block, request)
```
In C:

```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.

```c
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 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 `execvp`. 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:

- the ADDE group and descriptor names
- the stored data type and its format
- the minimum and maximum file numbers in the dataset
- any additional information needed to fulfill the request
- the dataset comment section
- a 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"
```

```c
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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>GMS</td>
</tr>
<tr>
<td>dataset</td>
<td>IR</td>
</tr>
<tr>
<td>type</td>
<td>IMAGE</td>
</tr>
<tr>
<td>format</td>
<td>AUST</td>
</tr>
<tr>
<td>info</td>
<td>east_ir.cfg</td>
</tr>
<tr>
<td>comment</td>
<td>GMS Infrared Imagery</td>
</tr>
<tr>
<td>min_range</td>
<td>3</td>
</tr>
<tr>
<td>max_range</td>
<td>8</td>
</tr>
<tr>
<td>real_time</td>
<td>0 (default)</td>
</tr>
</tbody>
</table>

**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 command, it is easiest to use the McIDAS command line parsing routines to extract information from the request. To use common command line data retrievers, such as `Mcmdstr` and `Mcmdint`, you must initialize the command line subsystem. You only need to do this for primary servers. The `MlnitLocalServer` 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:**

```fortran
character*256 request
integer len_req :

--- request already contains the request string to be parsed

call m0cmdput (m0cmdparse (request, len_req))
```
In C:

```c
char *request;
int len_req;
int stat;

/* request already contains the request string to be parsed */
stat = MCmdput (MCmparse (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 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 M0EXREXQ 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, M0EXREXQ or M0CXREQ assumes the server was unable to fulfill the request. The complete transmission format for each MclDAS 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 M0EXREXQ, followed by two 40-byte records. Words 2 through 4 in the record are character values; the remainder of the record contains integer values.

```c
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)

--- send the value 4 to the client indicating that it is to read an
--- additional 4 bytes; the additional 4 bytes will contain the
--- 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)

--- now we will loop 2 times, getting new data and sending it
--- back to the client; remember that words 2 through 4 are
--- character strings.

do 10 i = 1, 2

--- 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
--- 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.

<table>
<thead>
<tr>
<th>Trailer description</th>
<th>Length, in bytes</th>
<th>Word number in Fortran</th>
<th>servacct structure names in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>total reply length</td>
<td>4</td>
<td>41</td>
<td>reply_length</td>
</tr>
<tr>
<td>cpu used</td>
<td>4</td>
<td>42</td>
<td>cpu</td>
</tr>
<tr>
<td>return status code</td>
<td>4</td>
<td>43</td>
<td>returncode</td>
</tr>
<tr>
<td>error messages</td>
<td>72</td>
<td>44 - 61</td>
<td>errmsg</td>
</tr>
<tr>
<td>date</td>
<td>4</td>
<td>62</td>
<td>date</td>
</tr>
<tr>
<td>start time</td>
<td>4</td>
<td>63</td>
<td>start_time</td>
</tr>
<tr>
<td>end time</td>
<td>4</td>
<td>64</td>
<td>end_time</td>
</tr>
</tbody>
</table>

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.

```c
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 movcf (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 M0sxtrace or m0sxtrace will write trace messages to a file named trace. If tracing is not activated, M0sxtrace or m0sxtrace 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 M01sTraceSet or m01straceset 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 M01sTraceSet is made within the function M01initLocalServer. 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.

```c
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 = MGsxGetClientRequest (&request_block, &request);
ok = MGIsTraceSet (request);

/* print a trace message containing the entire user request */
sprintf (trace_string, "%s: %s", Server, request);
MGsxtrce (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

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. The table below provides a summary of the necessary components for each request type, in alphabetical order.

<table>
<thead>
<tr>
<th>Request type</th>
<th>Data type</th>
<th>Description</th>
<th>Client requester</th>
<th>Client reader</th>
<th>Sample McIDAS command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADIR</td>
<td>image</td>
<td>image header information</td>
<td>mcadir</td>
<td>mcadrd</td>
<td>IMGLIST</td>
</tr>
<tr>
<td>AGET</td>
<td>image</td>
<td>image header, navigation, calibration and data; data is returned line by line; comments</td>
<td>mcaget</td>
<td>mcalin</td>
<td>IMGDISP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcapfx</td>
<td>IMGCOPY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcacrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcacal</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcanav</td>
<td></td>
</tr>
<tr>
<td>GDIR</td>
<td>grid</td>
<td>grid header information</td>
<td>mcgdir</td>
<td>mcgdird</td>
<td>GRDLIST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mcgdird</td>
<td></td>
</tr>
<tr>
<td>GGET</td>
<td>grid</td>
<td>grid header and entire grid</td>
<td>mcgget</td>
<td>mcgridf</td>
<td>GRDDISP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GRDCOPY</td>
</tr>
<tr>
<td>MDFH</td>
<td>point</td>
<td>point file header information</td>
<td>m0pthdr</td>
<td>m0ptrdhdr</td>
<td>PTLIST FORM=FILE</td>
</tr>
<tr>
<td>MDHD</td>
<td>point</td>
<td>point header information</td>
<td>m0pthdr</td>
<td>m0ptrdhdr</td>
<td>PTLIST FORM=PARAM</td>
</tr>
<tr>
<td>MDKS</td>
<td>point</td>
<td>point header and data</td>
<td>m0ptget</td>
<td>m0ptrd</td>
<td>PTLIST</td>
</tr>
<tr>
<td>NAVG</td>
<td>nav</td>
<td>satellite navigation</td>
<td>m0navget</td>
<td>m0snlist</td>
<td>NAVLIST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m0nvblk</td>
<td></td>
</tr>
<tr>
<td>OBTG</td>
<td>text</td>
<td>observational weather text</td>
<td>M0obtxget</td>
<td>M0obtxread</td>
<td>OBSRPT</td>
</tr>
<tr>
<td>TXTG</td>
<td>text</td>
<td>ASCII text file</td>
<td>M0textgt</td>
<td>M0txrread</td>
<td>READ</td>
</tr>
<tr>
<td>WTXG</td>
<td>text</td>
<td>textual weather information</td>
<td>M0wtxget</td>
<td>M0wtxread</td>
<td>WXTLIST</td>
</tr>
</tbody>
</table>
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 commands, it is easiest to use the McIDAS 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 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 command IIMGDISP 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.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;1&quot;</td>
<td>group name</td>
<td></td>
</tr>
<tr>
<td>&quot;2&quot;</td>
<td>dataset name</td>
<td></td>
</tr>
<tr>
<td>&quot;3&quot;</td>
<td>position in the dataset</td>
<td>relative position in the dataset; negative means Nth most recent</td>
</tr>
<tr>
<td>&quot;4&quot;</td>
<td>coordinate type and position</td>
<td>(E)arth, (I)mage, (A)rea; (E)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</td>
</tr>
<tr>
<td>&quot;5&quot;</td>
<td>latitude or line number</td>
<td>latitude of the Earth (dd:mm:ss) or line number</td>
</tr>
<tr>
<td>&quot;6&quot;</td>
<td>longitude or element number</td>
<td>longitude of the Earth (dd:mm:ss) or element number</td>
</tr>
<tr>
<td>&quot;7&quot;</td>
<td>image resolution</td>
<td>default = 1</td>
</tr>
<tr>
<td>&quot;8&quot;</td>
<td>number of lines to transmit</td>
<td>default = 480</td>
</tr>
<tr>
<td>&quot;9&quot;</td>
<td>number of elements to transmit</td>
<td>default = 640</td>
</tr>
<tr>
<td>VERSION =</td>
<td>transmission version number</td>
<td>value as of 11/96 is 1</td>
</tr>
<tr>
<td>TIME = btime etime</td>
<td>time of day selection range</td>
<td>the format is hh:mm:ss (no default)</td>
</tr>
<tr>
<td>DAY =</td>
<td>the day that data is valid</td>
<td>not a range of days; format must be yyydd or cccyydd (no default)</td>
</tr>
<tr>
<td>Keywords</td>
<td>Description</td>
<td>Remarks</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UNITS</td>
<td>calibration type requested</td>
<td>not all calibration types are valid for a data type (default = stored units)</td>
</tr>
<tr>
<td>SPAC</td>
<td>number of bytes per data point</td>
<td>be careful when reducing data point resolution, if the data can be sent back in lower resolution (default = stored format)</td>
</tr>
<tr>
<td>CAL</td>
<td>set to QTIR to return TIROS in quick calibration</td>
<td></td>
</tr>
<tr>
<td>STYPE=VISR</td>
<td>sets calibration to UNITS of BRIT and type of VISR</td>
<td>converts data type to 1-byte data</td>
</tr>
<tr>
<td>BAND=band</td>
<td>specific band number to send or ALL bands</td>
<td></td>
</tr>
<tr>
<td>BAND=ALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMAG</td>
<td>line magnification factor</td>
<td>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</td>
</tr>
<tr>
<td>EMAG</td>
<td>element magnification factor</td>
<td>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</td>
</tr>
<tr>
<td>DOC</td>
<td>if YES, include the line documentation block</td>
<td>default = NO</td>
</tr>
<tr>
<td>AUX</td>
<td>if YES, additional calibration information is sent</td>
<td></td>
</tr>
</tbody>
</table>
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.

- If the UNITS requested is BRIT, SPAC=1. If UNITS is TEMP/ALB, SPAC=2.

- 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.

If the server has problems fulfilling the request, use the standard error values and messages found in -mcidas/src/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 7 of this manual in the AREA data file documentation. Below are the common modifications needed for the image header.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
<th>Reason for modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADDE dataset relative position number</td>
<td>always modified</td>
</tr>
</tbody>
</table>
| 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 | if client does not specify SIZE = ALL |
| 10   | number of elements in the image; must be divisible by 4 | if client does not specify SIZE = ALL |
| 11   | number of bytes per band | 1) if SPAC≠source value  
  |           | 2) if client requests STYPE = VISR |
| 12   | line resolution | if LMAC≠1 |
| 13   | element resolution | if EMAC≠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  
<p>|           | 2) if BAND≠ALL |
| 52   | source type; satellite-specific (ASCII) | if STYPE = VISR, set to VISR |</p>
<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
<th>Reason for modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>calibration type; satellite-specific (ASCII)</td>
<td>if STYPE = VISR, set to BRIT</td>
</tr>
<tr>
<td>54</td>
<td>sampling/averaging flag</td>
<td>set to 1 if blow down is performed by sampling</td>
</tr>
<tr>
<td>57</td>
<td>source's original satellite type (ASCII)</td>
<td>if STYPE = VISR, set to original satellite type</td>
</tr>
<tr>
<td>58</td>
<td>units of values returned</td>
<td>if AUX = YES and UNIT is specified</td>
</tr>
<tr>
<td>59</td>
<td>scaling of values returned</td>
<td>1) if UNIT is specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) if AUX = YES, set to 1</td>
</tr>
<tr>
<td>63</td>
<td>byte offset to the start of the calibration block</td>
<td>if calibration is sent (usually 256 + navlength)</td>
</tr>
<tr>
<td>64</td>
<td>number of comment cards</td>
<td>if comment cards exist</td>
</tr>
</tbody>
</table>

**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.

```plaintext
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_lines * 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

do 10 i = 1, n_lines
ok = getoneline(i, line, prefix)
call swbyt4(line, n_elements/4)
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_comments.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 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.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>' ' 1</td>
<td>group name</td>
<td></td>
</tr>
<tr>
<td>' ' 2</td>
<td>descriptor name</td>
<td></td>
</tr>
<tr>
<td>' ' 3</td>
<td>beginning file position number</td>
<td>can be the value ALL; positive numbers represent absolute locations; negative numbers are time-relative offsets (no default)</td>
</tr>
<tr>
<td>' ' 4</td>
<td>ending file position number</td>
<td>default = beginning file position number</td>
</tr>
<tr>
<td>AUX='</td>
<td>when YES, sends the center lat/lon, latitude resolution in km, longitude resolution in km, and valid calibration types as comment cards</td>
<td>default = NO</td>
</tr>
<tr>
<td>DAY=bdaysadays</td>
<td>day range to search</td>
<td>ccyyddd or yyddd format (no default)</td>
</tr>
<tr>
<td>SS=ss1 ss2</td>
<td>satellite sensor number</td>
<td>no default</td>
</tr>
<tr>
<td>TIME=stime etime</td>
<td>time range to search</td>
<td>hh:mm:ss format (no default)</td>
</tr>
</tbody>
</table>

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/src/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 AREA data file documentation in Chapter 7.

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
♦ the 256-byte image directory
♦ the 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

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, **geetserv**, 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 GRDDISP command accesses grid objects.

**Grid object request syntax**

The table below lists the client request syntax options for grid objects. The table below lists the client request syntax options for image directories. Note that keywords 1 through 3 are placed in the request block as positional parameters

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>••1</td>
<td>group name</td>
<td></td>
</tr>
<tr>
<td>••2</td>
<td>descriptor name</td>
<td></td>
</tr>
<tr>
<td>••3</td>
<td>maximum number of bytes that can be stored in the destination grid buffer on the client</td>
<td></td>
</tr>
<tr>
<td>DAY = day1 .. dayn</td>
<td>list of model-run days</td>
<td>ccyyddd format (no default)</td>
</tr>
<tr>
<td>DERIVE =</td>
<td>derived parameters</td>
<td></td>
</tr>
<tr>
<td>DRANGE = bday eday dinc</td>
<td>range of model-run Julian days</td>
<td>ccyyddd format</td>
</tr>
<tr>
<td>FDAY =</td>
<td>model-forecast valid day</td>
<td>ccyyddd format</td>
</tr>
<tr>
<td>FRANGE = fhr1 .. fhrn</td>
<td>range of model-forecast hours</td>
<td></td>
</tr>
<tr>
<td>FTIME =</td>
<td>model-forecast valid time</td>
<td>hhmms format</td>
</tr>
<tr>
<td>GRID = sgrid egrid</td>
<td>range of grid numbers to get from a McIDAS grid file</td>
<td></td>
</tr>
<tr>
<td>LEV = lev1 .. levn</td>
<td>list of data levels to retrieve</td>
<td>can be a number, SFC, MSL or TRO</td>
</tr>
<tr>
<td>NUM =</td>
<td>number of grids to retrieve</td>
<td>can be a number or ALL</td>
</tr>
<tr>
<td>PARM = p1 .. pn</td>
<td>list of parameters to retrieve</td>
<td></td>
</tr>
<tr>
<td>PARSE = select1 .. selectn</td>
<td>list of grid selection conditions for multiple grid parsing</td>
<td>format is a subset of other selection conditions specified; each selection is isolated by single quotes</td>
</tr>
<tr>
<td>PNUM =</td>
<td>number of parseable grids specified in the selection</td>
<td></td>
</tr>
<tr>
<td>Keywords</td>
<td>Description</td>
<td>Remarks</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>POS=</td>
<td>relative file position number in the dataset</td>
<td></td>
</tr>
<tr>
<td>PRO=</td>
<td>model-projection type</td>
<td></td>
</tr>
<tr>
<td>SRC=src1 .. srcn</td>
<td>list of model grid sources</td>
<td></td>
</tr>
<tr>
<td>TIME=time1 .. timen</td>
<td>model run times to retrieve</td>
<td>hhmmss format</td>
</tr>
<tr>
<td>TRANGE=btime etime tinc</td>
<td>range of model-run times</td>
<td>hhmmss format for time</td>
</tr>
<tr>
<td>VERSION=</td>
<td>grid transmission version</td>
<td>value is A as of 11/96</td>
</tr>
<tr>
<td>VT=vt1 .. vt2</td>
<td>valid hour offsets</td>
<td>hhmmss format</td>
</tr>
</tbody>
</table>

**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- 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.

- 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/src/adderror.doc. The range -2100 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 7 of this manual in the GRID 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: \((\text{numgrids} \times 260 + (\text{total number of data points} \times 4)) + 8\); 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, \((\text{number of rows} \times \text{number of columns}) \times 4\)
- 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.

```c
#include 'gridparm.inc'
integer grid_header(64)
integer grid(MAXGRIDPNT)
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 m0sxsends(4, temp_int)
call M0sxsends(4, temp_int)

C--- send the number of grids

temp_int = 1
call swbyt4(temp_int, 1)
call m0sxsends(4, temp_int)

C--- send the grid header; assume gridheaderno switches integer
C--- values in the header to network byte order

call gridheaderno(grid_header)
call m0sxsends(256, grid_header)

C--- send the data

call swbyt4(grid_data, grid_size)
call m0sxsends(grid_size*4, grid)

C--- send 0 separate

temp = 0
call swbyt4(temp_int, 1)
call M0sxsends(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 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.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>' ' '1</td>
<td>group name</td>
<td></td>
</tr>
<tr>
<td>' ' '2</td>
<td>descriptor name</td>
<td></td>
</tr>
<tr>
<td>DAY=day1 .. dayn</td>
<td>list of model-run days</td>
<td>ccyyddd format</td>
</tr>
<tr>
<td>DERIVE=</td>
<td>derived parameters</td>
<td></td>
</tr>
<tr>
<td>DRANGE=bday eday dinc</td>
<td>range of model-run Julian days</td>
<td>ccyyddd format</td>
</tr>
<tr>
<td>FDAY=</td>
<td>model forecast valid day</td>
<td>ccyyddd format</td>
</tr>
<tr>
<td>FRANGE=lhr1 .. lhrn</td>
<td>range of model-forecast hours</td>
<td></td>
</tr>
<tr>
<td>FTIME=</td>
<td>model-forecast valid time</td>
<td>hhmmss format</td>
</tr>
<tr>
<td>GRID=sgrid egrid</td>
<td>range of grid numbers to get</td>
<td></td>
</tr>
<tr>
<td></td>
<td>from a McIDAS grid file</td>
<td></td>
</tr>
<tr>
<td>LEV=lev1 .. levn</td>
<td>list of data levels to retrieve</td>
<td>can be a number, SFC,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSL or TRO</td>
</tr>
<tr>
<td>NUM=</td>
<td>number of grids to retrieve</td>
<td>can be a number or ALL</td>
</tr>
<tr>
<td>OUT=</td>
<td>output format</td>
<td>default = ALL</td>
</tr>
<tr>
<td>PARM=p1 .. pn</td>
<td>list of parameters to retrieve</td>
<td></td>
</tr>
<tr>
<td>PARSE=select1 .. selectn</td>
<td>list of grid selection conditions for multiple grid parsing</td>
<td>format will be a subset of other selection conditions specified; each selection is isolated by single quotes</td>
</tr>
<tr>
<td>PNUM=</td>
<td>number of parseable grids specified in the selection</td>
<td></td>
</tr>
<tr>
<td>POS=</td>
<td>relative file position number in the dataset</td>
<td></td>
</tr>
<tr>
<td>PRO=</td>
<td>model-projection type to retrieve</td>
<td></td>
</tr>
<tr>
<td>SRC=src1 .. srcn</td>
<td>list of model grid sources</td>
<td></td>
</tr>
<tr>
<td>Keywords</td>
<td>Description</td>
<td>Remarks</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td><code>TIME=time1 .. timeN</code></td>
<td>list of model-run times to retrieve</td>
<td>hhmmss format</td>
</tr>
<tr>
<td><code>TRANGE=bttime etime tinc</code></td>
<td>range of model-run times</td>
<td>hhmmss format for time</td>
</tr>
<tr>
<td><code>VERSION=</code></td>
<td>ADDE grid transmission version</td>
<td>value is 1 as of 11/96</td>
</tr>
<tr>
<td><code>VT=vt1 .. vt2</code></td>
<td>list of valid hour offsets</td>
<td>hhmmss format</td>
</tr>
</tbody>
</table>

**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 7 of this manual in the GRID 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/src/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 gridfileheaderno
C--- switches integer words to network byte order

   nbyets = 256
call gridfileheaderno(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 gridheaderno switches
C--- integer words to network byte order

   nbyets = 256
call gridheaderno (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 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.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Descriptions</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>' ' 1</td>
<td>group name</td>
<td></td>
</tr>
<tr>
<td>' ' 2</td>
<td>descriptor name</td>
<td></td>
</tr>
<tr>
<td>MAX=</td>
<td>maximum number of matches to find given the selection conditions</td>
<td></td>
</tr>
<tr>
<td>POS=</td>
<td>file position number in a dataset</td>
<td></td>
</tr>
<tr>
<td>SELECT=</td>
<td>data selection clause</td>
<td>see comments below</td>
</tr>
<tr>
<td>TRACE=</td>
<td>trace file activation</td>
<td>set to 1 to activate tracing</td>
</tr>
<tr>
<td>VERSION=</td>
<td>transmission version number</td>
<td>value is 1 as of 11/96</td>
</tr>
</tbody>
</table>

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 PREQUEST.

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 6, 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 ~/meidas/src/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 temp
integer scales(N_KEYS)
integer length
character*1 NULL
data NULL="/0"
integer temp

C--- send four parameters to the client: station ID, temperature 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 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 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.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>APRO=</td>
<td>AFOS/AWIPS product headers to match</td>
<td>three characters; don't use APRO with the WMO keyword</td>
</tr>
<tr>
<td>ASTN=</td>
<td>AFOS/AWIPS stations to match</td>
<td>two or three characters</td>
</tr>
<tr>
<td>DAY=</td>
<td>most recent day to begin the search</td>
<td>ccyyddd format (default = current day)</td>
</tr>
<tr>
<td>DTIME=</td>
<td>maximum number of hours back in time to search</td>
<td>no default</td>
</tr>
<tr>
<td>MATCH=</td>
<td>list of character match strings to find from the text</td>
<td></td>
</tr>
<tr>
<td>NUM=</td>
<td>number of matches to find</td>
<td>default = 1</td>
</tr>
<tr>
<td>PROD=</td>
<td>predefined product name</td>
<td></td>
</tr>
<tr>
<td>SOURCE=</td>
<td>circuit source</td>
<td>default = ALL</td>
</tr>
<tr>
<td>WMO=</td>
<td>WMO product headers to match</td>
<td>at least two characters; wildcard characters are allowed; don't use WMO with the APRO keyword</td>
</tr>
<tr>
<td>WSTN=</td>
<td>WMO stations to match</td>
<td>four characters</td>
</tr>
</tbody>
</table>
**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 *wtxgser* 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.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 3</td>
<td>circuit source</td>
<td>blank-padded character string</td>
</tr>
<tr>
<td>4 - 7</td>
<td>number of bytes in the data section</td>
<td></td>
</tr>
<tr>
<td>8 - 11</td>
<td>file address where the data is located</td>
<td>usually not important to the client</td>
</tr>
<tr>
<td>12 - 15</td>
<td>time the data was ingested</td>
<td>hhmms format</td>
</tr>
<tr>
<td>16 - 19</td>
<td>four-character WMO header</td>
<td>product code and country code</td>
</tr>
<tr>
<td>20 - 23</td>
<td>WMO product number</td>
<td>integer value</td>
</tr>
<tr>
<td>24 - 27</td>
<td>WMO station origin</td>
<td>four-character ICAO ID</td>
</tr>
<tr>
<td>28 - 31</td>
<td>AFOS product header</td>
<td>three characters</td>
</tr>
<tr>
<td>32 - 35</td>
<td>AFOS product location</td>
<td>two or three characters</td>
</tr>
<tr>
<td>36 - 39</td>
<td>AFOS product issuing station</td>
<td>three characters</td>
</tr>
<tr>
<td>40 - 43</td>
<td>Julian day of the data</td>
<td>ccyyddd format</td>
</tr>
<tr>
<td>44 - 47</td>
<td>number of bytes per line</td>
<td>usually 80</td>
</tr>
<tr>
<td>60 - 63</td>
<td>FAA catalog number</td>
<td></td>
</tr>
</tbody>
</table>
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 to 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
- the 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 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.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CO=co1 .. con</code></td>
<td>list of countries</td>
<td>2- character country IDs are read from COUNTRY.DAT, which is provided with McIDAS-XCD</td>
</tr>
<tr>
<td><code>ID=id1 .. idn</code></td>
<td>list of stations</td>
<td></td>
</tr>
<tr>
<td><code>IDREQ=</code></td>
<td>ID request type</td>
<td>must be set to LIST if CO, REG, or ID is specified; if a LATLON option is added to this server, specify GEO</td>
</tr>
<tr>
<td><code>NEWEST=day hour</code></td>
<td>most recent observation</td>
<td>default = most recent observation filed</td>
</tr>
<tr>
<td></td>
<td>time to allow in the request</td>
<td></td>
</tr>
<tr>
<td><code>NHIRS=</code></td>
<td>maximum number of hours back in</td>
<td>no default</td>
</tr>
<tr>
<td></td>
<td>time to search from the value in</td>
<td></td>
</tr>
<tr>
<td><code>NPERIOD=</code></td>
<td><code>NEWEST</code></td>
<td></td>
</tr>
<tr>
<td><code>NUM=</code></td>
<td>number of time periods to list</td>
<td>varies among data types</td>
</tr>
<tr>
<td><code>OLDEST=day hour</code></td>
<td>oldest observation time to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>allow in the request</td>
<td></td>
</tr>
<tr>
<td><code>REG=regn</code></td>
<td>list of station regions</td>
<td>the regions list is stored in GROUPS.DAT, which contains the stations for a particular state/province</td>
</tr>
<tr>
<td><code>TYPE=</code></td>
<td>numeric value for the type of</td>
<td>varies among data types</td>
</tr>
<tr>
<td></td>
<td>observation</td>
<td></td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 3</td>
<td>version number</td>
<td>currently zero</td>
</tr>
<tr>
<td>4 - 7</td>
<td>observation type number</td>
<td>varies among data types</td>
</tr>
<tr>
<td>8 - 11</td>
<td>active observation flag</td>
<td>0 = inactive, 1 = active</td>
</tr>
<tr>
<td>12 - 15</td>
<td>starting observation day</td>
<td>ccyyddd format</td>
</tr>
<tr>
<td>16 - 19</td>
<td>starting observation time</td>
<td>hhmmss format</td>
</tr>
<tr>
<td>20 - 23</td>
<td>starting observation hour</td>
<td>hhmmss format</td>
</tr>
<tr>
<td>24 - 27</td>
<td>ending observation day</td>
<td>ccyyddd format</td>
</tr>
<tr>
<td>28 - 31</td>
<td>ending observation time</td>
<td>hhmmss format</td>
</tr>
<tr>
<td>32 - 35</td>
<td>ending observation hour</td>
<td>hhmmss format</td>
</tr>
<tr>
<td>36 - 39</td>
<td>ID type flag</td>
<td>1 if ICAO ID</td>
</tr>
<tr>
<td>40 - 47</td>
<td>station ID</td>
<td>ccyyddd format</td>
</tr>
<tr>
<td>48 - 51</td>
<td>number of bytes of data</td>
<td></td>
</tr>
<tr>
<td>52 - 55</td>
<td>number of lines of data</td>
<td></td>
</tr>
<tr>
<td>56 - 95</td>
<td>reserved for future use</td>
<td></td>
</tr>
</tbody>
</table>
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.

```plaintext
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 sswbyt4(nbyte_nbo, 1)
  call m0sxsSend(4, n_bytes_nbo)
  call m0sxsSend(n_bytes, header)
  n_bytes = header(13)
  n_bytes_nbo = n_bytes
  call sswbyt4(n_bytes_nbo, 1)
  call m0sxsSend(4, n_bytes_nbo)
  call m0sxsSend(n_bytes, line)
goto 100
else
  n_bytes = 4
  n_bytes_nbo = n_bytes
  call sswbyt4(n_bytes_nbo, 1)
  call m0sxsSend(4, n_bytes_nbo)
call m0sxsSend(4, 0)
endif
```
Guidelines for Writing Helps

This appendix provides the guidelines to follow when creating or modifying a help to meet the McIDAS core standards. You will learn:

♦ the general guidelines to use for all command helps

♦ the specific guidelines to use for parameters, keywords and remarks

Sample helps for the McIDAS ZA and MG commands are also included. See the McIDAS-X or McIDAS-OS2 User’s Guide and online command helps for additional examples.

Contents

General guidelines for all helps ........................................... A-3

Specific guidelines ......................................................... A-4
  Parameters and keywords ............................................. A-4
  Parameters only ......................................................... A-4
  Keywords only ........................................................ A-5
  Remarks ................................................................. A-5

Sample helps ............................................................. A-6
  ZA command .......................................................... A-6
  MG command .......................................................... A-7
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 76 columns or less, and the total length of the help to 79 lines or less. This will accommodate McIDAS-X and -OS2 displays, which are formatted to 80 columns, and -OS2 text windows, which have 80 scrollable lines.

Verify that the first line of the help contains the command name, one or two hyphens and an accurate functional description. For example:

MG -- Plots a 24-hour 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. In xvi, use set format=os2, then :e!.

Include any specific information for McIDAS-X and/or -OS2, if the command resides in COM. See the EB, GU, DR and TERM command helps for examples.

See the following commands as examples if you’re writing a complex help: MDX, ROUTE, REFRESH, BATCH, ZA, MG, LVF, IMGCOPY, IMGOPER and GRDDISP.
Specific guidelines

Parameters and keywords

Check existing helps when naming parameters and keywords, since many McIDAS 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=value).

Use a vertical bar character ( | ) between the end of a parameter name or keyword name and its description. If possible, align these vertical bars.

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 MAKE, LIST, DIR, and DEL options of the MDU 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 keywords in uppercase with an equals sign, followed by a vertical bar character and the description. For example:

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:

FORM=ALL | lists an expanded directory
        =STN | lists the standard directory (def)

If a keyword accepts many values or a range of values, use the format shown below.

LEV=ll..ln | copies grids with these levels; for example, SFC, 1000, 850
BAND=bl..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. Extra remarks can be put in the command documentation in the McIDAS-X or -OS2 User's Guide.

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 helps

Helps for the ZA and MG commands are provided below as examples. Also look at the helps for the DSINFO, DATALOC, GINFO, GRDCOPY, MFPLT, ORBPLOT 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 -- Place annotation on graphics frame at cursor position
C ? ZA color height justify <keywords> "text"
C ? Parameters:
C ? color | graphics color level (def=1)
C ? height | pixel height to draw text/symbols (def=10 for text; cursor height for TCYC= symbol)
C ? justify | L=left, R=right, C=center, V=vertical, M=mouse
C ? N=numbered (def=L)
C ? "text" | text to write on the graphics frame
C ? "$text" | displays weather symbols or arrows on the graphics frame
C ? Keywords:
C ? ANG= | angle to write text when justify=L
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 ? GRA= | graphics frame number (def=current)
C ? TCYC=color hemisphere |
C ? | color - plot a tropical cyclone/hurricane symbol with the specified graphics color level
C ? Hemisphere - determines the direction of spirals:
C ? N=northern hemisphere
C ? S=southern hemisphere (def)
C ? WID= | graphics line width (def=current value; maximum=64)
C ? Remarks:
C ? When justify = M(ouse), the text positioning will be under the control of the mouse position and the clicking of the buttons. The MIDLE button will cause the start of the text to be repositioned to the cursor location, while the RIGHT button will reposition the end of the text. The height will be scaled as needed.
C ? When justify = N(umerous), text positioning will be centered at the cursor location when the MIDLE mouse button is clicked. The last drawn symbol will be erased when the right button is clicked.
C ? When ENCL= is used, the practical maximum number of characters is 3.
C ? ---------
**MG command**

C ? MG - Plots a 24-hour surface meteorogram
C ? MG station time day <keywords>
C ? Parameters:
C ? station | 3-4 character station identifier
C ? time | hour of latest data plotted on the graph, HH
C ? (def=current)
C ? day | year and day, YYDDD (def=current)
C ? Keywords:
C ? GRA= | graphics frame number (def=current)
C ? INIT=NO | do not initialize graphics colors (def=YES)
C ? MDF= | history MD file number (def=real-time ISFC MD file)
C ? SP=YES | display the graphics frame when completed (def=NO)
C ? Remarks:
C ? Plots surface hourly observations, temp, dew point(F),
C ? pressure, wind, weather and cloud cover for a 24-hour time
C ? period. Graphics are automatically erased before each plot.
C ?
C ? Cloud cover symbols: scattered is one small cloud; broken
C ? is clouds; overcast is a large cloud; clear is no clouds.
C ?
C ? Wind indicators: direction is wind barb's angle; sustained
C ? and gust speeds are listed below barb; calm is a
C ? diamond-shaped symbol.
C ?
C ? Pressure range labels depend on the minimum and maximum
C ? pressure for a given day and time period.
C ?
C ? MD file type can be either SVCA or ISFC.
C ? ---------
This appendix contains the SSEC-assigned sensor source numbers used in McIDAS. It also contains the band information for satellite data ingested in real time by the SSEC McIDAS-X servers.

**Contents**

<table>
<thead>
<tr>
<th>Sensor source numbers</th>
<th>B-3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bands for satellite imagery</strong></td>
<td></td>
</tr>
<tr>
<td>GVAR Imager</td>
<td>B-5</td>
</tr>
<tr>
<td>GVAR Sounder</td>
<td>B-6</td>
</tr>
<tr>
<td>GOES VISSR</td>
<td>B-7</td>
</tr>
<tr>
<td>POES</td>
<td>B-8</td>
</tr>
<tr>
<td>AVHRR sensor</td>
<td>B-8</td>
</tr>
<tr>
<td>TIP data</td>
<td>B-9</td>
</tr>
<tr>
<td>Meteosat</td>
<td>B-11</td>
</tr>
<tr>
<td>GMS</td>
<td>B-11</td>
</tr>
</tbody>
</table>
## Sensor source numbers

The table below lists the SSEC-assigned sensor source numbers used in McIDAS.

<table>
<thead>
<tr>
<th>Number</th>
<th>Sensor source</th>
<th>Number</th>
<th>Sensor source</th>
<th>Number</th>
<th>Sensor source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>derived data</td>
<td>27</td>
<td>GOES-4 infrared and water vapor (VAS)</td>
<td>71</td>
<td>GOES 8 sounder</td>
</tr>
<tr>
<td>1</td>
<td>test patterns</td>
<td>28</td>
<td>GOES-5 visible (VAS)</td>
<td>72</td>
<td>GOES 9 imager</td>
</tr>
<tr>
<td>2</td>
<td>graphics</td>
<td>29</td>
<td>GOES-5 infrared and water vapor (VAS)</td>
<td>73</td>
<td>GOES 9 sounder</td>
</tr>
<tr>
<td>3</td>
<td>miscellaneous</td>
<td>30</td>
<td>GOES-6 visible</td>
<td>74</td>
<td>GOES 10 imager</td>
</tr>
<tr>
<td>4</td>
<td>PDUS Meteosat visible</td>
<td>31</td>
<td>GOES-6 infrared</td>
<td>75</td>
<td>GOES 10 sounder</td>
</tr>
<tr>
<td>5</td>
<td>PDUS Meteosat infrared</td>
<td>32</td>
<td>GOES-7 visible, Block 1 supplemental data</td>
<td>76</td>
<td>GOES 11 imager</td>
</tr>
<tr>
<td>6</td>
<td>PDUS Meteosat water vapor</td>
<td>33</td>
<td>GOES-7 infrared</td>
<td>77</td>
<td>GOES 11 sounder</td>
</tr>
<tr>
<td>7</td>
<td>radar</td>
<td>41</td>
<td>TIROS-N (POES)</td>
<td>78</td>
<td>GOES 12 imager</td>
</tr>
<tr>
<td>8</td>
<td>miscellaneous aircraft data</td>
<td>42</td>
<td>NOAA-6</td>
<td>79</td>
<td>GOES 12 sounder</td>
</tr>
<tr>
<td>9</td>
<td>raw Meteosat</td>
<td>43</td>
<td>NOAA-7</td>
<td>80</td>
<td>ERBE</td>
</tr>
<tr>
<td>10</td>
<td>composite image</td>
<td>44</td>
<td>NOAA-8</td>
<td>82</td>
<td>GMS-4</td>
</tr>
<tr>
<td>12</td>
<td>GMS visible</td>
<td>45</td>
<td>NOAA-9</td>
<td>83</td>
<td>GMS-5</td>
</tr>
<tr>
<td>13</td>
<td>GMS infrared</td>
<td>46</td>
<td>Venus</td>
<td>84</td>
<td>GMS-6</td>
</tr>
<tr>
<td>14</td>
<td>ATS 6 visible</td>
<td>47</td>
<td>Voyager 1</td>
<td>85</td>
<td>GMS-7</td>
</tr>
<tr>
<td>15</td>
<td>ATS 6 infrared</td>
<td>48</td>
<td>Voyager 2</td>
<td>86</td>
<td>GMS-8</td>
</tr>
<tr>
<td>16</td>
<td>SMS-1 visible</td>
<td>49</td>
<td>Galileo</td>
<td>87</td>
<td>DMSP F-8</td>
</tr>
<tr>
<td>17</td>
<td>SMS-1 infrared</td>
<td>50</td>
<td>Hubble space telescope</td>
<td>88</td>
<td>DMSP F-9</td>
</tr>
<tr>
<td>18</td>
<td>SMS-2 visible</td>
<td>54</td>
<td>Meteosat-3</td>
<td>89</td>
<td>DMSP F-10</td>
</tr>
<tr>
<td>19</td>
<td>SMS-2 infrared</td>
<td>55</td>
<td>Meteosat-4</td>
<td>90</td>
<td>DMSP F-11</td>
</tr>
<tr>
<td>20</td>
<td>GOES-1 visible</td>
<td>56</td>
<td>Meteosat-5</td>
<td>91</td>
<td>DMSP F-12</td>
</tr>
<tr>
<td>21</td>
<td>GOES-1 infrared</td>
<td>60</td>
<td>NOAA-10</td>
<td>92</td>
<td>DMSP F-13</td>
</tr>
<tr>
<td>22</td>
<td>GOES-2 visible</td>
<td>61</td>
<td>NOAA-11</td>
<td>93</td>
<td>DMSP F-14</td>
</tr>
<tr>
<td>23</td>
<td>GOES-2 infrared</td>
<td>62</td>
<td>NOAA-12</td>
<td>94</td>
<td>DMSP F-15</td>
</tr>
<tr>
<td>24</td>
<td>GOES-3 visible</td>
<td>63</td>
<td>NOAA-13</td>
<td>95</td>
<td>FY-1B</td>
</tr>
<tr>
<td>25</td>
<td>GOES-3 infrared</td>
<td>64</td>
<td>NOAA-14</td>
<td>96</td>
<td>FY-1C</td>
</tr>
<tr>
<td>26</td>
<td>GOES-4 visible (VAS)</td>
<td>70</td>
<td>GOES 8 imager</td>
<td>97</td>
<td>FY-1D</td>
</tr>
</tbody>
</table>
**Bands for satellite imagery**

This section describes the bands for satellite data ingested in real time by the McIDAS-X servers. 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 over which the satellite collects data.

- 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.
The GVAR (GOES VARiable) imager produces observational data for a spatial location in five spectral bands: one visible (VIS) and four infrared (IR). Its sensor characteristics are shown below.

<table>
<thead>
<tr>
<th>Band</th>
<th>Central wavelength, µm</th>
<th>Measurement range</th>
<th>Meteorological objective; maximum temperature range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.65 (VIS)</td>
<td>1.6 to 100% albedo</td>
<td>cloud cover</td>
</tr>
<tr>
<td>2</td>
<td>3.9 (IR)</td>
<td>4 to 320 K</td>
<td>night clouds; space - 340 K</td>
</tr>
<tr>
<td>3</td>
<td>6.8 (IR)</td>
<td>4 to 320 K</td>
<td>water vapor; space - 290 K</td>
</tr>
<tr>
<td>4</td>
<td>10.7 (IR)</td>
<td>4 to 320 K</td>
<td>surface temperature; space - 340 K</td>
</tr>
<tr>
<td>5</td>
<td>12.0 (IR)</td>
<td>4 to 320 K</td>
<td>sea surface temperature and water vapor; space - 335 K</td>
</tr>
</tbody>
</table>

An image contains only one of these five bands. Word 19 in the directory block contains a spectral band map indicating the image file’s band.

The resolution is 1-km VIS, 4-km IR and 8-km water vapor. For a GVAR satellite, resolution one means approximately 1 km resolution at the satellite subpoint.

For more information about the format of GVAR Imager data, see the AREAmmnn data file described in Chapter 7, Format of the Data Files.
The GVAR Sounder produces 8-km resolution data for a given spatial location in 18 IR spectral bands and one visible band. The characteristics of its bands are listed below.

<table>
<thead>
<tr>
<th>Band</th>
<th>Central wavelength, $\mu$m</th>
<th>Absorbing gases</th>
<th>Wave number, cm$^{-1}$</th>
<th>Sounding level, mb</th>
<th>Meteorological objective; maximum temperature range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14.7</td>
<td>CO$_2$</td>
<td>680</td>
<td>90</td>
<td>temperature; space - 280 K</td>
</tr>
<tr>
<td>2</td>
<td>14.4</td>
<td>CO$_2$</td>
<td>696</td>
<td>300</td>
<td>sounding; space - 280 K</td>
</tr>
<tr>
<td>3</td>
<td>14.1</td>
<td>CO$_2$</td>
<td>711</td>
<td>550</td>
<td>sounding; space - 290 K</td>
</tr>
<tr>
<td>4</td>
<td>13.6</td>
<td>CO$_2$; H$_2$O</td>
<td>733</td>
<td>850</td>
<td>sounding; space - 310 K</td>
</tr>
<tr>
<td>5</td>
<td>13.4</td>
<td>CO$_2$; H$_2$O</td>
<td>748</td>
<td>900</td>
<td>sounding; space - 320 K</td>
</tr>
<tr>
<td>6</td>
<td>12.7</td>
<td>H$_2$O</td>
<td>790</td>
<td>surface</td>
<td>sounding; space - 330 K</td>
</tr>
<tr>
<td>7</td>
<td>12.0</td>
<td>H$_2$O</td>
<td>832</td>
<td>surface</td>
<td>surface temp; space - 340 K</td>
</tr>
<tr>
<td>8</td>
<td>11.0</td>
<td>H$_2$O</td>
<td>907</td>
<td>surface</td>
<td>surface temp; space - 345 K</td>
</tr>
<tr>
<td>9</td>
<td>9.7</td>
<td>O$_3$</td>
<td>1030</td>
<td>25</td>
<td>total ozone; space - 330 K</td>
</tr>
<tr>
<td>10</td>
<td>7.4</td>
<td>H$_2$O</td>
<td>1345</td>
<td>725</td>
<td>water vapor; space - 310 K</td>
</tr>
<tr>
<td>11</td>
<td>7.0</td>
<td>H$_2$O</td>
<td>1425</td>
<td>550</td>
<td>sounding; space - 295 K</td>
</tr>
<tr>
<td>12</td>
<td>6.5</td>
<td>H$_2$O</td>
<td>1535</td>
<td>350</td>
<td>sounding; space - 290 K</td>
</tr>
<tr>
<td>13</td>
<td>4.6</td>
<td>N$_2$O</td>
<td>2188</td>
<td>1000</td>
<td>temperature; space - 320 K</td>
</tr>
<tr>
<td>14</td>
<td>4.5</td>
<td>N$_2$O</td>
<td>2210</td>
<td>950</td>
<td>sounding; space - 310 K</td>
</tr>
<tr>
<td>15</td>
<td>4.4</td>
<td>CO$_2$; N$_2$O</td>
<td>2248</td>
<td>650</td>
<td>sounding; space - 295 K</td>
</tr>
<tr>
<td>16</td>
<td>4.1</td>
<td>CO$_2$</td>
<td>2420</td>
<td>surface</td>
<td>sounding; space - 240 K</td>
</tr>
<tr>
<td>17</td>
<td>4.0</td>
<td>window</td>
<td>2513</td>
<td>surface</td>
<td>surface temp; space - 345 K</td>
</tr>
<tr>
<td>18</td>
<td>3.7</td>
<td>window</td>
<td>2671</td>
<td>surface</td>
<td>temperature; space - 345 K</td>
</tr>
<tr>
<td>19</td>
<td>0.70</td>
<td>window</td>
<td>14367</td>
<td>surface</td>
<td>cloud cover</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>star scan</td>
</tr>
</tbody>
</table>

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.

For more information about the format of GVAR Sounder data, see the AREAnnnn data file described in Chapter 7, Format of the Data Files.

Satellite Information
B-6
McIDAS Programmer's Manual
Issued 8/97
The VAS (VISSR Atmospheric Sounder) instrument senses the atmosphere for a given spatial location in up to 12 different IR spectral bands and a visible band. All or some of the IR bands may be included in a single VAS type image file. The visible, however, must be contained in a separate image file of the type VISSR (Visible and Infrared Spin Scan Radiometer). As a result, it may require two image files to contain the total information transmitted by the satellite during a given time period. The resolution is 1 km VIS and either 4- or 8-km IR.

The characteristics of the VAS bands are provided in the table below. Band D (13) is the same as band 8 but at a lower resolution.

<table>
<thead>
<tr>
<th>Band</th>
<th>Central wavelength, μm</th>
<th>Absorbing gases</th>
<th>Sounding level, mb</th>
<th>Primary use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14.7 (IR)</td>
<td>CO₂</td>
<td>70</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>2</td>
<td>14.5 (IR)</td>
<td>CO₂</td>
<td>125</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>3</td>
<td>14.3 (IR)</td>
<td>CO₂</td>
<td>200</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>4</td>
<td>14.0 (IR)</td>
<td>CO₂</td>
<td>500</td>
<td>temperature sounding, cloud heights</td>
</tr>
<tr>
<td>5</td>
<td>13.3 (IR)</td>
<td>CO₂</td>
<td>920</td>
<td>temperature sounding, cloud heights</td>
</tr>
<tr>
<td>6</td>
<td>4.5 (IR)</td>
<td>CO₂; N₂O</td>
<td>850</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>7</td>
<td>12.7 (IR)</td>
<td>H₂O</td>
<td>1000</td>
<td>water vapor detection</td>
</tr>
<tr>
<td>8</td>
<td>11.2 (IR)</td>
<td>window</td>
<td>surface</td>
<td>nighttime cloud detection, surface temp</td>
</tr>
<tr>
<td>9</td>
<td>7.3 (IR)</td>
<td>H₂O</td>
<td>600</td>
<td>water vapor detection</td>
</tr>
<tr>
<td>A</td>
<td>6.7 (IR)</td>
<td>H₂O</td>
<td>400</td>
<td>water vapor detection</td>
</tr>
<tr>
<td>B</td>
<td>4.4 (IR)</td>
<td>CO₂; N₂O</td>
<td>300</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>C</td>
<td>3.9 (IR)</td>
<td>window</td>
<td>surface</td>
<td>surface skin temperature, cloud detection</td>
</tr>
<tr>
<td>D</td>
<td>0.65 (VIS)</td>
<td>window</td>
<td>surface</td>
<td>daytime cloud detection</td>
</tr>
</tbody>
</table>

For more information about the format of GOES data, see the AREAnnnn data file described in Chapter 7, Format of the Data Files.
The POES satellites, currently NOAA-12 and -14, contain the AVHRR (Advanced Very High Resolution Radiometer) instrument, which is a 5-channel scanning radiometer and the TIP (TIROS-N Information Processor) instrument.

**AVHRR sensor**

The AVHRR instrument has five channels: channels 1 and 2 are visible; channels 3, 4 and 5 are infrared, as shown below.

<table>
<thead>
<tr>
<th>Band</th>
<th>Central wavelength, μm</th>
<th>Primary use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.63 (VIS)</td>
<td>daytime cloud, snow and ice mapping data</td>
</tr>
<tr>
<td>2</td>
<td>0.91 (VIS)</td>
<td>surface water delineation, vegetation and agriculture assessments</td>
</tr>
<tr>
<td>3</td>
<td>3.7 (IR)</td>
<td>nighttime cloud mapping, sea surface temperature measurements, land and water distinctions, and hot spot detection such as volcanic activity or forest fires</td>
</tr>
<tr>
<td>4</td>
<td>11.0 (IR)</td>
<td>cloud mapping, sea and land surface temperature measurements, and soil moisture and volcanic eruption data</td>
</tr>
<tr>
<td>5</td>
<td>12.0 (IR)</td>
<td>sea surface temperature measurements and soil moisture data</td>
</tr>
</tbody>
</table>

The AVHRR 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 users 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.

For more information about the format of AVHRR data, see the AREA\text{nnnn} data file described in Chapter 7, *Format of the Data Files*. 

Satellite Information

B-8

McIDAS Programmer’s Manual

Issued 8/97
**TIP data**

TIP data is extracted from the AVHRR, GAC and LAC data described above. It also contains information received from these instruments:

- MSU (Microwave Sounding Unit), which has four microwave channels and provides 115-km resolution data
- SSU (Stratospheric Sounding Unit), which has three infrared channels and provides 62-km resolution data
- HIRS (High Resolution Infrared Sounder), which has one visible and 19 infrared channels, and provides 20-km resolution data

These three sources measure incoming radiation in the infrared and microwave spectrum. The band information for each is provided below.

### MSU instrument

<table>
<thead>
<tr>
<th>Band</th>
<th>Frequency, GHz</th>
<th>Absorbing gases</th>
<th>Primary use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50.3</td>
<td>O₂</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>2</td>
<td>53.8</td>
<td>O₂</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>3</td>
<td>55.0</td>
<td>O₂</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>4</td>
<td>58.0</td>
<td>O₂</td>
<td>temperature sounding</td>
</tr>
</tbody>
</table>

### SSU instrument

<table>
<thead>
<tr>
<th>Band</th>
<th>Central wavelength, µm</th>
<th>Absorbing gases</th>
<th>Sounding level, mb</th>
<th>Primary use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.0</td>
<td>CO₂</td>
<td>100</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>2</td>
<td>15.0</td>
<td>CO₂</td>
<td>35</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>3</td>
<td>15.0</td>
<td>CO₂</td>
<td>10</td>
<td>temperature sounding</td>
</tr>
</tbody>
</table>
## HIRS instrument

<table>
<thead>
<tr>
<th>Band</th>
<th>Central wavelength, µm</th>
<th>Absorbing gases</th>
<th>Sounding level, mb</th>
<th>Primary use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.0 (IR)</td>
<td>CO₂</td>
<td>30</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>2</td>
<td>14.7 (IR)</td>
<td>CO₂</td>
<td>60</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>3</td>
<td>14.5 (IR)</td>
<td>CO₂</td>
<td>100</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>4</td>
<td>14.2 (IR)</td>
<td>CO₂</td>
<td>400</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>5</td>
<td>14.0 (IR)</td>
<td>CO₂</td>
<td>600</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>6</td>
<td>13.7 (IR)</td>
<td>CO₂; H₂O</td>
<td>800</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>7</td>
<td>13.3 (IR)</td>
<td>CO₂; H₂O</td>
<td>900</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>8</td>
<td>11.1 (IR)</td>
<td>window</td>
<td>surface</td>
<td>surface temperature</td>
</tr>
<tr>
<td>9</td>
<td>9.7 (IR)</td>
<td>O₃</td>
<td>25</td>
<td>ozone measurement</td>
</tr>
<tr>
<td>10</td>
<td>8.2 (IR)</td>
<td>H₂O</td>
<td>900</td>
<td>water vapor detection; NOAA-6 through -10 and -12</td>
</tr>
<tr>
<td></td>
<td>12.5 (IR)</td>
<td>H₂O</td>
<td>surface</td>
<td>water vapor detection; NOAA-11 and -14</td>
</tr>
<tr>
<td>11</td>
<td>7.3 (IR)</td>
<td>H₂O</td>
<td>700</td>
<td>water vapor detection</td>
</tr>
<tr>
<td>12</td>
<td>6.8 (IR)</td>
<td>H₂O</td>
<td>500</td>
<td>water vapor detection</td>
</tr>
<tr>
<td>13</td>
<td>4.57 (IR)</td>
<td>N₂O</td>
<td>1000</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>14</td>
<td>4.52 (IR)</td>
<td>N₂O</td>
<td>950</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>15</td>
<td>4.47 (IR)</td>
<td>CO₂; N₂O</td>
<td>700</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>16</td>
<td>4.4 (IR)</td>
<td>CO₂; N₂O</td>
<td>400</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>17</td>
<td>4.2 (IR)</td>
<td>CO₂</td>
<td>surface</td>
<td>temperature sounding</td>
</tr>
<tr>
<td>18</td>
<td>4.0 (IR)</td>
<td>window</td>
<td>surface</td>
<td>surface temperature</td>
</tr>
<tr>
<td>19</td>
<td>3.8 (IR)</td>
<td>window</td>
<td>surface</td>
<td>surface temperature</td>
</tr>
<tr>
<td>20</td>
<td>0.70 (VIS)</td>
<td>window</td>
<td>cloud</td>
<td>daytime cloud detection</td>
</tr>
</tbody>
</table>

For more information about the format of TIP data, see the AREAmmnm data file described in Chapter 7, Format of the Data Files.
Meteosat

The Meteosat radiometer produces 2.5 km resolution visible data, 5 km resolution infrared data and B-sector data, which is data covering Europe, northern Africa and the eastern Atlantic. This radiometer is currently active on the Meteosat-3, -4 and -5 satellites (sensor source numbers 54, 55 and 56).

The characteristics of its bands are listed below.

<table>
<thead>
<tr>
<th>Band</th>
<th>Central wavelength, μm</th>
<th>Primary use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.75 (VIS)</td>
<td>daytime cloud detection</td>
</tr>
<tr>
<td>8</td>
<td>11.5 (IR)</td>
<td>nighttime cloud detection and surface temperature</td>
</tr>
<tr>
<td>10 (A)</td>
<td>6.9 (IR)</td>
<td>water- vapor detection</td>
</tr>
</tbody>
</table>

GMS

The GMS-4 satellite with sensor source numbers 12 and 13 contains a VISSR instrument that provides 1.25-km resolution visible data, 5.0-km resolution infrared data, and documentation. The characteristics of its bands are listed below.

<table>
<thead>
<tr>
<th>Band</th>
<th>Central wavelength, μm</th>
<th>Primary use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw ingest 1</td>
<td>11.5 (IR)</td>
<td>nighttime cloud detection and surface temperature</td>
</tr>
<tr>
<td>4</td>
<td>documentation</td>
<td>not displayable; ingest documentation</td>
</tr>
<tr>
<td>14</td>
<td>0.63 (VIS)</td>
<td>daytime cloud detection</td>
</tr>
<tr>
<td>Converted 1</td>
<td>0.63 (VIS)</td>
<td>daytime cloud detection</td>
</tr>
<tr>
<td>8</td>
<td>11.5 (IR)</td>
<td>nighttime cloud detection and surface temperature</td>
</tr>
</tbody>
</table>
The GMS-4, -5, -6, -7 and -8 satellites, with sensor source numbers 82 through 86, contain a VISSR instrument that provides 1.25-km resolution visible data, 5.0-km resolution infrared data, and documentation. The characteristics of its bands are listed below.

<table>
<thead>
<tr>
<th>Band</th>
<th>Central wavelength, µm</th>
<th>Primary use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.63 (VIS)</td>
<td>daytime cloud detection</td>
</tr>
<tr>
<td>2</td>
<td>11.0 (IR)</td>
<td>nighttime cloud detection and surface temperature</td>
</tr>
<tr>
<td>3</td>
<td>12.0 (IR)</td>
<td>sea surface temperatures and soil moisture</td>
</tr>
<tr>
<td>4</td>
<td>6.8 (IR)</td>
<td>water vapor detection</td>
</tr>
<tr>
<td>5</td>
<td>documentation</td>
<td>not displayable; ingest documentation</td>
</tr>
</tbody>
</table>
Valid Parameter Names

The table provided in this appendix lists the common data parameter names used in McIDAS, along with a description of each.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACFT</td>
<td>aircraft type (ASCII)</td>
</tr>
<tr>
<td>ASIZ</td>
<td>aircraft size</td>
</tr>
<tr>
<td>CA</td>
<td>cloud amount; octants of the sky covered</td>
</tr>
<tr>
<td>CAI1</td>
<td>cloud amount of the first layer</td>
</tr>
<tr>
<td>CAI2</td>
<td>cloud amount of the second layer</td>
</tr>
<tr>
<td>CAI3</td>
<td>cloud amount of the third layer</td>
</tr>
<tr>
<td>CAI4</td>
<td>cloud amount of the fourth layer</td>
</tr>
<tr>
<td>CC</td>
<td>cloud cover</td>
</tr>
<tr>
<td>CC1</td>
<td>cloud cover; first cloud layer non-ceiling</td>
</tr>
<tr>
<td>CC2</td>
<td>cloud cover; second cloud layer non-ceiling</td>
</tr>
<tr>
<td>CCH</td>
<td>high cloud cover (0=CLR, 1=SCT, 2=BKN, 3=OVC)</td>
</tr>
<tr>
<td>CCL</td>
<td>low cloud cover (0=CLR, 1=SCT, 2=BKN, 3=OVC)</td>
</tr>
<tr>
<td>CCM</td>
<td>middle cloud cover (0=CLR, 1=SCT, 2=BKN, 3=OVC)</td>
</tr>
<tr>
<td>CG1</td>
<td>genus of the first cloud (ASCII)</td>
</tr>
<tr>
<td>CG2</td>
<td>genus of the second cloud (ASCII)</td>
</tr>
<tr>
<td>CG3</td>
<td>genus of the third cloud (ASCII)</td>
</tr>
<tr>
<td>CG4</td>
<td>genus of the fourth cloud (ASCII)</td>
</tr>
<tr>
<td>CH</td>
<td>type of high level clouds (ASCII)</td>
</tr>
<tr>
<td>CIG</td>
<td>categorical forecast of the ceiling height</td>
</tr>
<tr>
<td>CIGC</td>
<td>cloud cover of the ceiling</td>
</tr>
<tr>
<td>CIGH</td>
<td>ceiling height</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>CL</td>
<td>type of low level clouds (ASCII)</td>
</tr>
<tr>
<td>CM</td>
<td>type of middle level clouds (ASCII)</td>
</tr>
<tr>
<td>CMAX</td>
<td>number of columns containing data</td>
</tr>
<tr>
<td>CO</td>
<td>country ID (ASCII); see the McIDAS CCODE command for more information</td>
</tr>
<tr>
<td>DAY</td>
<td>year and Julian day</td>
</tr>
<tr>
<td>DIR</td>
<td>wind direction</td>
</tr>
<tr>
<td>FLV1</td>
<td>first flight level</td>
</tr>
<tr>
<td>FLV2</td>
<td>second flight level</td>
</tr>
<tr>
<td>GUS</td>
<td>wind gusts</td>
</tr>
<tr>
<td>HMS</td>
<td>actual time of the observation</td>
</tr>
<tr>
<td>ICE1</td>
<td>first icing code</td>
</tr>
<tr>
<td>ICE2</td>
<td>second icing code</td>
</tr>
<tr>
<td>ICTP</td>
<td>icing type (ASCII)</td>
</tr>
<tr>
<td>ID</td>
<td>station identifier (ASCII)</td>
</tr>
<tr>
<td>IDA</td>
<td>part one of a two-part ID (ASCII)</td>
</tr>
<tr>
<td>IDB</td>
<td>part two of a two-part ID (ASCII)</td>
</tr>
<tr>
<td>IDN</td>
<td>station ID number</td>
</tr>
<tr>
<td>IFQ1</td>
<td>first icing frequency (ASCII)</td>
</tr>
<tr>
<td>IFQ2</td>
<td>second icing frequency (ASCII)</td>
</tr>
<tr>
<td>IL11</td>
<td>first icing base</td>
</tr>
<tr>
<td>IL12</td>
<td>first icing top</td>
</tr>
<tr>
<td>IL21</td>
<td>second icing base</td>
</tr>
<tr>
<td>IL22</td>
<td>second icing top</td>
</tr>
<tr>
<td>LAT</td>
<td>latitude, + North (-90 to +90)</td>
</tr>
<tr>
<td>LEV</td>
<td>level: SFC, TRO, or same as P (ASCII)</td>
</tr>
<tr>
<td>LLWS</td>
<td>low level wind shear</td>
</tr>
<tr>
<td>LON</td>
<td>longitude, + West (-180 to +180)</td>
</tr>
<tr>
<td>MOD</td>
<td>modification flag (1 = modified)</td>
</tr>
<tr>
<td>NREC</td>
<td>number of records in a row</td>
</tr>
<tr>
<td>OBSV</td>
<td>categorical forecast of obstructions to vision</td>
</tr>
<tr>
<td>P</td>
<td>atmospheric pressure</td>
</tr>
<tr>
<td>P1</td>
<td>temperature for significant temperature level</td>
</tr>
<tr>
<td>P1</td>
<td>wind direction for significant wind level</td>
</tr>
<tr>
<td>P2</td>
<td>dew point temperature for significant temperature level</td>
</tr>
<tr>
<td>P2</td>
<td>wind speed for significant wind level</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>P3</td>
<td>pressure for significant temperature level</td>
</tr>
<tr>
<td>P3</td>
<td>height for significant wind level</td>
</tr>
<tr>
<td>PC</td>
<td>characteristic of pressure tendency</td>
</tr>
<tr>
<td>PCP</td>
<td>3-hour precipitation total</td>
</tr>
<tr>
<td>PP06</td>
<td>6-hour probability of precipitation</td>
</tr>
<tr>
<td>PP12</td>
<td>12-hour probability of precipitation</td>
</tr>
<tr>
<td>PRE</td>
<td>pressure</td>
</tr>
<tr>
<td>PSD</td>
<td>direction the predominant swell comes from</td>
</tr>
<tr>
<td>PSH</td>
<td>height of the predominant swell</td>
</tr>
<tr>
<td>PSL</td>
<td>sea level pressure</td>
</tr>
<tr>
<td>PSNO</td>
<td>probability of snow</td>
</tr>
<tr>
<td>PSP</td>
<td>period of predominant swell</td>
</tr>
<tr>
<td>PST</td>
<td>station pressure</td>
</tr>
<tr>
<td>PT</td>
<td>3-hour pressure change</td>
</tr>
<tr>
<td>PTIM</td>
<td>time period for precipitation</td>
</tr>
<tr>
<td>PTYP</td>
<td>precipitation type (ASCII)</td>
</tr>
<tr>
<td>PZR</td>
<td>probability of freezing rain</td>
</tr>
<tr>
<td>QP06</td>
<td>6-hour quantitative precipitation forecast</td>
</tr>
<tr>
<td>QP12</td>
<td>12-hour quantitative precipitation forecast</td>
</tr>
<tr>
<td>SN06</td>
<td>6-hour snow probability</td>
</tr>
<tr>
<td>SN12</td>
<td>12-hour snow probability</td>
</tr>
<tr>
<td>SNO</td>
<td>cumulative snow depth</td>
</tr>
<tr>
<td>SPD</td>
<td>wind speed</td>
</tr>
<tr>
<td>ST</td>
<td>state ID (ASCII)</td>
</tr>
<tr>
<td>STI</td>
<td>insitu sea surface temperature</td>
</tr>
<tr>
<td>SV06</td>
<td>6-hour severe weather probability</td>
</tr>
<tr>
<td>SV12</td>
<td>12-hour severe weather probability</td>
</tr>
<tr>
<td>SWH</td>
<td>height of sea or wind waves</td>
</tr>
<tr>
<td>SWP</td>
<td>period of sea or wind wave</td>
</tr>
<tr>
<td>T</td>
<td>temperature</td>
</tr>
<tr>
<td>TBTP</td>
<td>turbulence type (ASCII)</td>
</tr>
<tr>
<td>TD</td>
<td>dew point temperature</td>
</tr>
<tr>
<td>TFQ1</td>
<td>first turbulence frequency (ASCII)</td>
</tr>
<tr>
<td>TFQ2</td>
<td>second turbulence frequency (ASCII)</td>
</tr>
<tr>
<td>TH06</td>
<td>6-hour thunderstorm probability</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>TH12</td>
<td>12-hour thunderstorm probability</td>
</tr>
<tr>
<td>TIME</td>
<td>time</td>
</tr>
<tr>
<td>TL11</td>
<td>first turbulence base</td>
</tr>
<tr>
<td>TL12</td>
<td>first turbulence top</td>
</tr>
<tr>
<td>TL21</td>
<td>second turbulence base</td>
</tr>
<tr>
<td>TL22</td>
<td>second turbulence top</td>
</tr>
<tr>
<td>TMAX</td>
<td>maximum temperature</td>
</tr>
<tr>
<td>TMIN</td>
<td>minimum temperature</td>
</tr>
<tr>
<td>TRB1</td>
<td>first turbulence intensity</td>
</tr>
<tr>
<td>TRB2</td>
<td>second turbulence intensity</td>
</tr>
<tr>
<td>TYPE</td>
<td>type of data</td>
</tr>
<tr>
<td>VDAY</td>
<td>Julian day of forecast verification</td>
</tr>
<tr>
<td>VIS</td>
<td>visibility</td>
</tr>
<tr>
<td>VTIM</td>
<td>time of forecast verification</td>
</tr>
<tr>
<td>WX1</td>
<td>weather - first four characters (ASCII)</td>
</tr>
<tr>
<td>WX2</td>
<td>weather - second four characters (ASCII)</td>
</tr>
<tr>
<td>WXP</td>
<td>present weather type (ASCII)</td>
</tr>
<tr>
<td>Z</td>
<td>height above sea level</td>
</tr>
<tr>
<td>ZC1</td>
<td>base of the first cloud layer</td>
</tr>
<tr>
<td>ZC11</td>
<td>first cloud base</td>
</tr>
<tr>
<td>ZC12</td>
<td>first cloud top</td>
</tr>
<tr>
<td>ZC2</td>
<td>base of the second cloud layer</td>
</tr>
<tr>
<td>ZC21</td>
<td>second cloud base</td>
</tr>
<tr>
<td>ZC22</td>
<td>second cloud top</td>
</tr>
<tr>
<td>ZC3</td>
<td>base of the third cloud layer</td>
</tr>
<tr>
<td>ZC4</td>
<td>base of the fourth cloud layer</td>
</tr>
<tr>
<td>ZCB</td>
<td>height of the cloud base</td>
</tr>
<tr>
<td>ZCL1</td>
<td>height of the first non-ceiling</td>
</tr>
<tr>
<td>ZCL2</td>
<td>height of the second non-ceiling</td>
</tr>
<tr>
<td>ZS</td>
<td>surface elevation</td>
</tr>
</tbody>
</table>
This appendix contains the following additional POES AVHRR calibration information resulting from the launch of NOAA-15 satellite in May 1998.

**Contents**

| AVHRR Calibration Background | D-2 |
| TIRO and AVHR Differences    | D-3 |
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 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 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 ingester. 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 retention 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 1000 for slopes and offsets and 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 requires 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 four constants per line for each thermal IR channel instead of two constants (80 bytes total). The internal target temperature is now redundantly written into the CAL section of the line prefix for bands 3, 4, and 5 instead of overwriting the channel 3 patch temperature and spare word. The scaling factors are also different from TIRO. These line prefix changes make AVHR calibration type images incompatible with TIRO calibration.

5. The CAL=QUICK option is no longer available for AVHR, although the first set of twenty (was 10) valid constants and the target temperature (scaled as in the line prefix) will be written to the first 21 words of the calibration codicil for backward compatibility, allowing a "quick" calibration mode for displaying AVHRR/3 imagery if desired. Because these are small integer values embedded in 4-byte words, the leading byte is always zero, so there will be no sensitivity to ADDE byte ordering problems for the calibration codicil. Application of these constants is left to the user; McIDAS will no longer support quick calibration, since current computers are fast enough now to do everything the long (and more correct) way on the fly.
0-based
A counting sequence that begins with zero.

1-based
A counting sequence that begins with one.

ADDE
Abstract Data Distribution Environment software in McIDAS-X or -OS2 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 command line.

area
The McIDAS image file format.

ASCII file
American Standard Code for Information Interchange file containing only text; for example, schema definition files and scripts.

asynchronous
McIDAS commands that run asynchronously will return control to the original calling program before they have run to completion. Also see synchronous.

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.

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 DATALOC command.

clipping region
See viewport.

command line
The command or series of commands entered in the McIDAS 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. In McIDAS-OS2, the number of characters permitted in a command line is 512.

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 library functions are placed for one year when they are no longer referenced by any core programs. Also see local library and McIDAS 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 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 display. Users manipulate the cursor to interact with McIDAS commands and the McIDAS Image Window. Several cursor sizes, types and colors are available.

**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 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 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; usually a workstation monitor or an X Terminal.

**distributed data system**
In McIDAS, 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.

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
In McIDAS, the place where applications programs reside, along with McIDAS 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 supports the sinusoidal
equal-area projection.

extended format
McIDAS 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.

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. In McIDAS, file formats include image, grid, point and text; non-McIDAS
formats include HDF and NetCDF.

frame
Contains a representation of an image sector displayed on the McIDAS 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 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.

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 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 Image Window.

gray scale
The range of black-to-white gray shades available for displaying image data on the McIDAS 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.

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.

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 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 and forms the basis for other McIDAS 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-generated images and graphics.

**include files**
Files that hold definitions of constants specific to the McIDAS 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 library function in Fortran or C.

**I/O**
Input/Output.

**Julian day**
Calendar date based on a 365-day year, usually in the form cyyddd; for example, 1996056 is February 25, 1996.

**keywords**
Alphanumeric values that provide input to a McIDAS 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.

**line**
Image line; each image line contains an optional line prefix and the actual data values. Lines run horizontally across a McIDAS 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 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.

**macro**
A McIDAS command that runs a series of McIDAS 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 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-OS2 runs on OS/2 workstations.

**McIDAS library**
A file containing all the object code for the functions and subroutines that make up the McIDAS Application Program Interface (API). In McIDAS-OS2, it is called libmc.lib; in -X, it is called libmcidas.a. 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.

**navigation**
The process of transforming image coordinates (lines and elements) to earth coordinates (latitude and longitude) and vice versa.

**navigation block**
A McIDAS 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.

**OS/2**
A multitasking PC operating system developed by IBM; McIDAS-OS2 requires this system.

**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 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
In McIDAS-X, a three-button mouse; the leftmost button is used by the window manager and the middle and right buttons are used by the McIDAS mouse interface. In McIDAS-OS2, a two-button mouse; the mouse buttons are shared between the window manager and application.

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 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.

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.

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 environment; file redirection information from the McIDAS REDIRECT command is stored in them. Also see file redirection.

resident programs
Programs that are basic to McIDAS and its environment. For example, in McIDAS-X, the resident program mettext controls command line input from the keyboard and displays text output on the Text and Command Window.
resolution
See full, image, or satellite resolution.

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 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 or -OS2 workstation session acts as both a client and a local server. McIDAS-X workstations can also be configured as remote servers, supplying data to all -X and -OS2 clients.

server mapping table
The table containing the list of dataset names on the server. Users assign these names with the McIDAS DSSERVE command.

shared memory
A component of the McIDAS 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 commands outside of McIDAS.

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. In McIDAS, 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 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 commands that run synchronously will run to completion before control is returned to the original calling program. Also see asynchronous.

**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 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 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.

**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 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).

**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 programming to generate graphical output in panels. Also called a clipping region.

**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.
Index

A
abort 5-70
Absorbing gases B-4
Accessing
  compilers 4-7
  data 2-7–2-10, 6-1, 7-42
Accounts
  mcidas 3-3
  set up 3-3
ADDE
  client request 8-12
  clients 2-8
  communication 8-4
  dataflow 2-9
  description 2-8
  naming scheme 2-9
  overview 8-3
  servers
    building 8-10–8-20
    general weather text 8-45
    grid directories 8-38
    grid objects 8-34
    image directories 8-31
    image objects 8-25
    observational weather text 8-49
    point data 8-42
    set up 2-8
  adirserv 8-31
  agetserv 8-25
  Algorithms, navigation 6-78
  alit 5-76, 5-83
  Annotation, image 7-73
  API, see Application Program Interface
  Application Program Interface
    calibration 6-97
    data access functions 7-42
    disk file 6-5
    man pages 1-14
    naming conventions 2-35
    navigation 6-71–6-75
    servers 8-5
  Applications
    debugging 2-48–2-51
    error return codes 5-70
    manipulating bytes 5-75–5-85
    programs 2-27
    suspending 5-72
    writing 2-30–2-34
  Area
    file format 7-3–7-42
    map 7-6
  areaparm.inc 2-32
  Arrays 2-45
  ASCII files 7-43
  AVHRR
    calibration block 7-31
    data block 7-30
description 7-29
directory block 7-29
navigation block 7-30
AVHRR sensor B-8
AWIPS header line 6-64

B

Bands
GMS B-11
GVAR sounder B-6
spectral 6-13, 7-7
Barbs 5-49
Base map files 7-71
Beta parameters 5-107
Blocks
  calibration
    AVHRR 7-31
    definition 7-6
    GMS 7-37
    GOES-7 AAA 7-28
    GVAR imager 7-17
    GVAR sounder 7-22
    image data 6-14
    Meteosat PDUS 7-11
    reading 6-27
  comment
    definition 7-8
    image data 6-14
    read 6-28
  data
    AVHRR 7-30
    definition 7-6
    GOES-7 AAA 7-24
    grid 6-36
    GVAR sounder 7-20
    image 6-14
    Meteosat PDUS 7-11
    TIP 7-33
    writing 6-29
  directory
    AVHRR 7-29
    definition 7-4-7-5
    GOES-7 AAA 7-24

GVAR imager 7-13
GVAR sounder 7-18
image data 6-14
image directory 6-16
TIP 7-33
documentation
  GVAR imager 7-12
  interface 2-36-2-41
  GVAR Block 11 data 7-18, 7-19
  line prefix 6-14, 6-25
  navigation
    AVHRR 7-30
    definition 7-5
    DMSP 7-39
    GMS 7-34
    GOES-7 AAA 7-26
    GVAR imager 7-15
    GVAR sounder 7-21
    image data 6-14
    Meteosat PDUS 7-11
    reading 6-27
  supplemental 7-6
  Brightness temperatures 7-60
  Buffers 5-27, 5-32
  Building
    Dynamic Link Libraries 3-12–3-14, 4-13
    servers 8-10–8-20
  Buttons, mouse 5-19
  Bytes
    converting values 5-83
    copying 5-80–5-82
    extracting and inserting values 5-78
    flipping 2-52, 5-79
    manipulating 5-75–5-85
    moving streams 5-83
    switching order 5-79

C

C programming language
  function interfaces 2-42–2-47
  programming guidelines 2-56
  writing applications 2-31
CAL. See Calibration block

Calibration

API 6-97
basic concepts 6-96, 6-97
block
AVHRR 7-31
definition 7-6
DMSP 7-40
GMS 7-37
GOES-7 AAA 7-28
GVAR imager 7-17
GVAR sounder 7-22
Meteosat PDU 7-11
reading 6-27
designing modules 6-98
file, GMS 7-53
modules, writing 6-96–6-102
MSU reference parameters 7-69
parameters 7-60
sample program 6-103
VAS tables 7-84
cartll 6-71

Central wavelength B-4

Character strings
compare 5-94
convert
case 5-96
to C 5-92
to Fortran 5-92
determine contents 5-93
extract from a quote field 5-13
extract value from command line 5-8,
5-10
manipulating 5-85–5-106
number of characters in 5-94
passing into functions 2-44

Client
requests
interpreting 8-16
reading 8-12
retrieving 8-17
sending 8-17
server communication 8-4
transmission formats 8-23
transmitting data 8-4, 8-5, 8-19
clit 5-76

Codes
error 2-33, 8-27, 8-31, 8-40, 8-43, 8-46,
8-50
missing value 5-69
status 5-16
coefficients, HIRS 7-63
colour 5-20
Color
enhancements 2-17
intensities 7-59

Command
keywords 2-14, 5-7
line
extracting values from 5-8
interrogating structure 5-13
using 5-5
multiple-letter 2-13
output 2-15
parameters 2-14, 5-6
quoted text 2-15
schedule file 7-74
single-letter 2-12, 2-13, 5-20

Comment block
definition 7-8
reading 6-28
writing 6-30

Compilers
f2c 4-7
gcc 4-7
install 4-9

Compiling
code 4-7
source files 3-9, 4-10
tools 4-7

Conformal projections 6-69
convdlm 3-12

Conventions
API naming 2-35
programming 1-7
used in manual 1-11

Conversion utilities 5-75–5-107

Coordinates
convert 5-27
systems 2-17
world 5-33
Coriolis 5-107
rack 5-76
Cursor appearance 5-20

D

Dash mode
current 5-27
turn on and off 5-27
Data
AA and AAA 7-24
accessing 2-7–2-10
ADDE 2-9, 8-3
alphanumeric 2-6
area files 7-3–7-42
block
AVHRR 7-30
Block 11 7-18
definition 7-6
GOES-7 AAA 7-24
GVAR imager 7-14
GVAR sounder 7-19, 7-20
Meteosat PDUS 7-11
VISR 7-23
writing 6-29
end transaction 8-19
flow in ADDE 2-9
GAC 7-29
global area coverage 7-29
GOES-7 7-24
grid
accessing 6-33–6-47
characteristics 2-6
serving 8-34–8-37
HRPT 7-29
image
characteristics 2-5
display 5-25
reading 6-23
serving 8-25–8-30
using 6-12–6-32
LAC 7-29
parameter names C-1–C-4

point
accessing 6-48–6-58
characteristics 2-5, 6-49
file header 6-58
objects 6-51
serving 8-42–8-44
size 6-13
requests
grid directory 8-38
grid object 8-34
image directory 8-31
image objects 8-25
interpreting 8-16
point 8-42
retrieving 8-17
sample 8-8
serving
general weather text 8-45–8-48
grid 8-34–8-37
grid directory 8-38–8-41
image 8-25–8-30
image directory 8-31–8-32
observational weather text data 8-49–8-51
point 8-42–8-44
rules 8-5
static 2-6
text 6-61–6-66
TIP 7-33, B-9
transmission formats 8-23
transmitting 8-4, 8-17
types 2-4–2-6
weather text
accessing 6-63–6-66
characteristics 2-6
serving 8-45–8-48, 8-49–8-51
Date
convert format 5-90, 5-100–5-104
extract as an integer 5-8
extract from command line 5-12
increment 5-103
retrieve current 5-99
dbx 2-49
DCFO14 7-43
DCIRAB 7-43
DCISEN 7-43
DCISFC 7-43
DCISHP 7-43
DCIPIRP 7-43
DCSYN 7-43
ddest 2-48, 5-23
Debugging
  applications 2-48–2-51
  M0ASSERT 2-48
  messages 5-22
  servers 8-21–8-22
  tools 2-49
Defining 5-20
delgra 7-59
Developer overview 2-29–2-56
Development tools
dbx 2-49
dbg 2-48
dmix 4-7
Dew point 5-108
dfline 5-25
Directories
  frame 7-51
  grid
    convert format 8-39
    error codes 8-40
    request syntax 8-38
    rules for transmitting 8-39
    send to client 8-39
    serving 8-38–8-41
  image
    convert format 8-32
    reading 6-16
    request syntax 8-31
    rules for transmitting 8-31
    send to client 8-32
    serving 8-31–8-32
McIDAS-OS2 4-4
McIDAS-X
  installation 3-6
  mclidas account 3-5
  package 3-6
  set up 2-10
  user account 3-7
Directory
  block
  AAA 7-24
  AVHRR 7-29
  entries 6-18
  GVAR imager 7-13
  GVAR sounder 7-18, 7-20
  Meteosat PDUS 7-10
  reading 6-16
  TIP 7-33
Disk files
  APIs 6-5
    assigning a pathname 6-7
    characteristics 6-3
    copying 6-11
    deleting 6-8–6-9
    determining existence 6-8
    locking and unlocking 6-10
    reading and writing 6-6
Display
  characteristics 5-53
  loops 2-17
  text messages 5-23
  utilities 5-21–5-56
dlir 5-76, 5-83
DLLs, see Dynamic Link Libraries
dmake utility 4-11
DMSP
  calibration blocks 7-40
  navigation block 7-39
DMSPCAL 7-40
Documentation
  block, GVAR 7-12
  companion manuals 1-14
  man pages 1-14
Double precision numbers
  functions for extracting 5-8
  latitude/longitude value as 5-8
  time value in units of numbers 5-8
dshof 5-27
dshoff 5-30
dshon 5-30
Dynamic Link Libraries
  building 3-12–3-14, 4-13
  generating subsystem functions 3-12
  writing modules 2-47
Index

vi

McIDAS Programmer's Manual
Revised 11/98

E
edest 5-23
emx development kit 4-7
endplt 5-27, 5-32
Enhancements
color 2-17
frame table 7-49
save files 7-46
empt 5-27
Environment
McIDAS 2-21, 2-28
user 4-3
variables
McPATH 2-10
PATH 3-4
Equal-area projections 6-70
Equidistant projections 6-35
Equivalent potential temperature 5-109
 Erafmi 7-52
Error codes
grid
directories 8-40
objects 8-35
image
directories 8-31
objects 8-27
observational weather text 8-50
point data 8-43
Errors
general weather text 8-46
handling 2-33
messages 5-22
return codes 5-70–5-71
.ET files 7-46
Examples
general weather text request 6-66
helps A-6
interface documentation block 2-36
serving grid objects 8-37
Extract values from command line 5-8

F
f2c compiler 4-7, 4-10
fileparm.inc 2-32
Files
area 7-3–7-42
calibration
GMS 7-53
HIRS reference parameters 7-60
coordinates 2-18
enhancements 7-46
flat 6-61
grid 7-54
HIRS transmittance coefficients 7-62
identify location 2-10
image annotation descriptions 7-73
include 2-31
initialize 8-12
locks 5-73
map 7-71
McIDAS disk 6-3–6-11
object 3-10, 3-11
redirection 2-10
reporting station list 7-64
scheduler 7-74
source 3-9
source file suffixes 2-29
flalo 5-104
Flipping bytes 2-52
Floating point numbers 2-52
Forecasts 6-63
Fortran
callable routines, writing 2-46
function interfaces 2-42–2-47
guidelines for programming 2-53
writing programs 2-30
Frame
coordinates 2-19
directory 7-51
enhancements 7-49
graphics 5-27
navigation 7-51
navigation API 6-72
objects 2-24
panels 7-48
size 2-17, 5-55  
FRAMED 7-48  
FRAMENH.001 7-49  
FRAMEx.y 7-51  
frtony 7-52  
fsalloc 5-85, 5-92  
fslen 5-85  
Functions  
conversion 5-75–5-107  
day and time 5-99  
display 5-21–5-51  
error handling 5-70–5-71  
graphics 5-27  
interfaces 2-42–2-47  
libraries 3-8  
M0/m0 prefix 1-8, 2-35  
naming 3-8  
passing character strings into 2-44  
passing parameters into 2-43  
prefixes 2-35  
private 1-8  
scientific 5-107–5-112  
system 5-56–5-75  
User Common 5-61  
user interface 5-5–5-21  
ggetserv 8-34  
Global area coverage data 7-29  
GMS  
bands B-11  
calibration block 7-37  
navigation block 7-34  
GMSCAL 7-53  
GOES VISSR 7-23, B-7  
GOES-7  
AAA  
calibration block 7-28  
data block 7-24  
navigation block 7-26  
data 7-24  
GRAF.PGM 5-38  
Graphics  
buffers 5-27, 5-32  
frame 5-27  
functions 5-27  
initializing the subsystem 5-29  
levels 2-17  
lines 5-29  
package initialize 5-27  
save table  
delete 7-59  
restore 7-59  
word allocation 7-59  
write 7-59  
techniques 5-28, 5-33  
text and numbers 5-31  
winds and vectors 5-49  
grddef 6-71  
Grid  
characteristics 6-34–6-36  
data  
accessing 6-33–6-47  
block 6-36  
characteristics 2-6  
reading 6-44  
serving 8-34–8-37  
directories  
convert format 8-39  
error codes 8-40  
request syntax 8-38  
rules for transmitting 8-39  
gcc compiler 4-7, 4-10  
gdirsterr 8-38  
General weather text data  
convert header format 8-46  
error codes 8-46  
reading 6-65  
rules for transmitting 8-46  
send to client 8-47  
serving 8-45–8-48  
using 6-63–6-66  
WMO format standards 6-63  
Geographic calculation API 6-71  
geolat 6-71  
Geostationary satellites 7-34  
getfrm 7-52  
gettime 5-98
send to client 8-39
serving 8-38–8-41
fields, contouring 5-46
header
contents 6-42
description 6-36
open connection 6-41
reading 6-41
levels 6-34
navigation
API 6-71
definition 6-68
objects
convert format 8-36
definition 6-36
error codes 8-35
reading 6-37–6-44
request syntax 8-34
rules for transmitting 8-35
send to client 8-36
writing to dataset 6-45–6-47
origin 6-34
parameters 6-34
projections 6-35
selection conditions 6-38
Grid file
API functions 7-58
directory
open connection 7-58
read 7-58
word allocation 7-54
header
read 7-58
word allocation 7-55–7-58
GRIDnnn 7-54
gridparm.inc 2-32
Grids
delete 7-58
read 7-58
selection parameters 7-58
write 7-58
.GRX 7-59
Guidelines
programming 2-52–2-56
serving data
grid directories 8-39
grid objects 8-35
image directories 8-31
image directory objects 8-32
image objects 8-27
point 8-42
GVAR
Block 0 7-12
Block 11 data 7-18, 7-19
imager
calibration block 7-17
data block 7-14
directory block 7-13
navigation block 7-15
sensor characteristics B-5
sensor data 7-12
supplemental data 7-12
sounder
band characteristics B-6
calibration block 7-22
data block 7-20
directory block 7-20
navigation block 7-21

H

Header
general weather text
AWIPS 6-64
convert format 8-46
grid 6-36
observational weather text 8-50
point data 6-58
Heartbeat values 8-36, 8-39, 8-47
Heat index 5-107, 5-110
Help
creating 2-34, 3-15, 4-14, A-1
line length A-3
McIDAS Help Desk 1-13
mcmkhelp 3-15
parameters and keywords A-4
samples A-6
templates 2-34
using 2-15
hircon 7-61
HIRS
  calibration parameters 7-60
  initialize 7-61
  instrument B-10
  retrieve coefficients 7-63
  transmittance coefficients 7-62
HIRSCRPF 7-60
HIRSTAUL 7-62
Hyposomeric equation 5-107

definition 6-69
implementing modules 6-76–6-84
objects
  blocks in 6-14
  convert format 8-28
  delete 6-32
  error codes 8-27
  reading 6-15, 6-19–6-28
  request syntax 8-25
  send to client 8-29
  writing to dataset 6-29–6-31
radar 2-5
resolution 6-12
satellite bands B-4–B-12
Window 2-16
Imager, GVAR 7-12
Include files 2-31
initblok 8-12
Initialize
  constants 7-70
  file system 8-12
  graphics package 5-27
initpl 5-27, 5-31
Install compilers 4-9
Installation directories 3-6
Integer values
  convert
    latitude/longitude to 5-88
    strings to 5-87
    time to 5-88
  extract from command line 5-11
  write 5-27
Interface documentation block 2-36–2-41
ischar 5-85, 5-93
isdgch 5-85, 5-93
Isentropic surfaces 5-109
itrmeh 5-53
itvll 6-72
iyxll 2-43

Julian date
  convert 5-90, 5-100–5-104
next century 2-52

calculate length 7-8
reading 6-25

K

kbxcal 6-103
kbxini 6-103
kbxopt 6-103
keyin 5-30, 5-68
Keywords
entered by user 5-13, 5-15
list parameters 5-13, 5-15
using in commands 2-14
verify validity 5-13, 5-14

L

lab 5-107, 5-109
Lambert conformal projections 6-35
Latent heat of vaporization 5-107
Latitude/Longitude
convert
to double precision and integer 5-88
to single precision 5-104
extract as double precision number 5-8
lbget 5-58
lbi 6-5, 6-6
lbo 6-5, 6-6
lpput 5-57, 5-59
lpputc 5-57, 5-60
Levels
grids 6-34
image and graphics 2-17
libmc.lib 4-5
libmcidas.a 3-8
Libraries
Dynamic Link 2-47, 3-12–3-14, 4-13
functions 1-8
local 3-8, 4-5
McIDAS 2-35–2-41, 3-8, 4-5
object files 3-11, 4-12
Line
drawing 5-27, 5-29
prefix block

M

M0ASSERT 2-48
m0distnc 6-71
m0getusr 7-52
M0gsort 6-40
m0gsort 7-58
M0InitLocalServer 8-21
M0IsTraceSet 8-21
M0lock 5-73
m0panel.h 7-51
m0ptbufinit 6-55
m0ptget 6-55
m0pthdr 6-58
m0ptrd 6-55
m0putusr 7-52
m0ssdesc 7-73

Index
x
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0sxdone 8-19</td>
<td></td>
</tr>
<tr>
<td>M0sxSetTraceOff 8-21</td>
<td></td>
</tr>
<tr>
<td>M0sxSetTraceOn 8-21</td>
<td></td>
</tr>
<tr>
<td>M0sxtrc 8-21</td>
<td></td>
</tr>
<tr>
<td>M0textgt 6-62</td>
<td></td>
</tr>
<tr>
<td>M0txtread 6-62</td>
<td></td>
</tr>
<tr>
<td>M0unlock 5-73</td>
<td></td>
</tr>
<tr>
<td>M0wtxget 6-64</td>
<td></td>
</tr>
<tr>
<td>M0wtxread 6-64</td>
<td></td>
</tr>
<tr>
<td>main 2-30</td>
<td></td>
</tr>
<tr>
<td>make utilities 4-11</td>
<td></td>
</tr>
<tr>
<td>Man pages, API 1-14</td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td></td>
</tr>
<tr>
<td>area 7-6</td>
<td></td>
</tr>
<tr>
<td>files 2-6, 7-71</td>
<td></td>
</tr>
<tr>
<td>projections 6-69</td>
<td></td>
</tr>
<tr>
<td>Mapping table 8-15</td>
<td></td>
</tr>
<tr>
<td>mcacl 7-42</td>
<td></td>
</tr>
<tr>
<td>mcaclou 6-30, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcacdr 6-28, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcadel 6-32, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcadir 6-16, 6-22, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcadr 7-42</td>
<td></td>
</tr>
<tr>
<td>mcafree 6-23, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcaget 6-20, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcain 6-23, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcnav 7-42</td>
<td></td>
</tr>
<tr>
<td>mcaout 6-29, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcapfx 6-25, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcaput 6-29, 7-42</td>
<td></td>
</tr>
<tr>
<td>mcsc 3-11</td>
<td></td>
</tr>
<tr>
<td>mcasort 6-22, 7-42</td>
<td></td>
</tr>
<tr>
<td>McBeta 5-107</td>
<td></td>
</tr>
<tr>
<td>mcbeta 5-107</td>
<td></td>
</tr>
<tr>
<td>mccmd 5-13</td>
<td></td>
</tr>
<tr>
<td>mccmcdbl 5-8</td>
<td></td>
</tr>
<tr>
<td>mccmdhdr 5-8</td>
<td></td>
</tr>
<tr>
<td>mccmddll 5-8</td>
<td></td>
</tr>
<tr>
<td>mccmdhdr 5-8</td>
<td></td>
</tr>
<tr>
<td>mccmddll 5-8</td>
<td></td>
</tr>
<tr>
<td>Mccmdint 8-16</td>
<td></td>
</tr>
<tr>
<td>mccmdint 5-8, 5-11</td>
<td></td>
</tr>
<tr>
<td>mccmdf 5-8</td>
<td></td>
</tr>
<tr>
<td>Mccmdkey 5-13</td>
<td></td>
</tr>
<tr>
<td>mcmddkey 5-13</td>
<td></td>
</tr>
</tbody>
</table>
Mcsmissingdb 5-69
McIsMissingFlt 5-69
mcsmissingreal 5-69
Mcstimeofday 5-98
mcstimeofday 5-98
Mckeyin 5-30
mckeyin 5-68
McLatentHeat 5-107
mclatvap 5-107
mclcl 5-107
McLiftCLevel 5-107
mccline 5-26
Mclocase 5-86, 5-96
mclocase 5-86
Mcluc 5-63, 7-83
McMixing 5-107
mcmkhelp 3-15
Mcmoubtn 5-72
mcmoubtn 5-18, 5-19, 7-83
McPATH 2-10
Mcpathname 6-5, 6-7
mepcal 6-16
McPFromTheta 5-107
mcpffromtheta 5-107
meplotskywind 5-49
meplotskywindnav 5-49
meplotskywindnavll 5-49
mepnav 6-16
Mcprintf 5-23
Mcpuc 5-63, 7-83
Mcread 6-5, 6-6
merelative humidity 5-110
McRelativeHumidity 5-107, 5-110
merelativehumidity 5-107, 5-110
Mcremove 6-5, 6-9
mcrmix 5-107
mcsatvap 5-108
mcsatvapi 5-108
McSatVapor 5-108
McSatVaporIce 5-108
Msectodaytime 5-98
msectodaytime 5-98
mcserv 8-3, 8-7
McSetGraphicsFrameNumber 5-63
mcsetgraphicsframenum 5-63
McSetGraphicsFrameOn 5-64
mcsetgraphicsframeon 5-64
McSetImageFrameNumber 5-63
mcsetimageframenum 5-63
McSetImageFrameOn 5-63
mcsetimageframeon 5-63
McSetStdDebugDevice 5-64
mcsetstddebugdevice 5-64
McSetStdErrorDevice 5-64
mcsetstderrordevice 5-64
McSetStdOutputDevice 5-64
mcsetstdoutputdevice 5-64
mcskeyi 5-68
mcsleep 5-72
McSpeed 5-108
McStationPres 5-108
mcstationpres 5-108
Mcstricmp 5-86, 5-94
Mcstrnicmp 5-86, 5-94
Mcstrodb 5-88
Mcstrodb 5-86, 5-88
mcstrodb 5-86
Mcstrodhr 5-86, 5-88
mcstrodhr 5-86
Mcstrodll 5-86, 5-88
mcstrodll 5-86
Mestretofs 5-86, 5-92
Mestretohex 5-86
mcstretohex 5-86
Mestretohms 5-86, 5-90
mcstretohms 5-86
Mestretoih 5-86, 5-88
mcstretoih 5-86
Mestretoil 5-86, 5-88
mcstretoil 5-86
Mestretoint 5-86, 5-87
mcstretoint 5-86
Mestretoiyd 5-86, 5-90
mcstretoiyd 5-86
mctamr 5-108
mcctasa 5-108
McTDFromMixing 5-108
McTempAtThetae 5-108
mcctext 2-26
McTFromTheta 5-108
mctfromtheta 5-108  
McTheta 5-108  
mcteta 5-108  
McTheta 5-108  
mcvrt 5-108  
mcvrt 5-108  
MCUandV 5-108  
mcuandv 5-108  
mcuandv 5-108  
mcuandv 5-108  
mcupcase 5-86, 5-96  
mcupcase 5-86  
MCVirtTemp 5-108  
mcvirttemp 5-108  
McWetBulb 5-108  
mcwetbulb 5-108  
McWindChill 5-108, 5-111  
mcwindchill 5-108, 5-111  
Mcwrite 6-5, 6-6  
Meydtoicyd 5-98, 5-100  
meydtoicyd 5-98  
MD  
   file structure 7-66  
   schema definition files 7-43–7-45  
mdksserv 8-42  
mdsvc 2-44  
MDXXnnmn 7-66  
Memory, shared 2-21  
Messages  
   route 5-23  
   text 5-22  
Meteorological Data files. See MD  
Meteorological parameters 5-107–5-112  
Meteosat 7-9, B-11  
Meteosat PDUS  
   calibration block 7-11  
   data block 7-11  
   directory block 7-10  
   navigation block 7-11  
Microwave Sounding Unit 7-69  
Missing value codes 5-69  
Mixing ratio 5-107, 5-109, 5-111  
Mouse  
functions 5-18  
report state 7-83  
state of buttons 5-19  
movb 5-76  
movblk 5-76, 5-81  
movc 5-77  
movcw 5-77  
movh 5-77  
movpix 5-77, 5-81  
movw 5-77  
movwc 5-77  
mpixel 5-77  
mpixtb 5-77, 5-83  
MSU  
   constants 7-70  
   instrument B-9  
msucon 7-70  
MSUCRPF 7-69  
MUG Help Desk 1-13  
Multiple-letter commands 2-13  

N  

Names  
ADDE datasets 2-9  
API conventions 2-35  
functions 3-8  
parameters C-1–C-4  
NAV. See Navigation block  
Navigation  
   algorithms 6-78  
   API 6-71–6-75  
   basic concepts 6-68–6-70  
   block  
      AVHRR 7-30  
      definition 6-70, 7-5  
      DMSP 7-39  
      GMS 7-34  
      GOES-7 AAA 7-26  
      GVAR imager 7-15  
      GVAR sounder 7-21  
      Meteosat PDUS 7-11  
      reading 6-27  
      coordinate systems 2-17  

Index  
xiv  

McIDAS Programmer's Manual  
Revised 11/98
frame 7-51
   APIs 6-69, 6-72
grid 6-68, 6-71
grid projections 6-35
image 6-69, 6-73
implementing 6-76–6-84
sample code 6-85
transforms 6-68
nchars 5-86, 5-94
Negative UC 2-22
newplt 5-27
nllxyz 6-71
Numbers
   double precision 5-8
   floating point 2-52
   integers 5-27
nvprep 6-73
nvset 6-73
nvtofr 7-52
nvxini 6-73
nvxopt 6-73
nvxsae 6-73
nxyzll 6-71
reading 6-15, 6-19–6-28
request syntax 8-25
rules for transmitting 8-27
send to client 8-29
writing to dataset 6-29–6-31
point
   definition 6-51
   reading 6-52
Observational weather text data
   convert header format 8-50
   error codes 8-50
   request syntax 8-49
   rules for transmitting 8-50
   serving 8-49–8-51
obtgserv 8-49
OLS 7-40
Online help
   creating 2-34, 3-15, 4-14
   guidelines for writing A-1
   line length A-3
   man pages 1-14
   parameters and keywords A-4
   samples A-6
   templates 2-34
   using 2-15
Operational Linescan System. See OLS 7-38
Origin, grids 6-34
OUTL* 7-71
Output
   characteristics 5-53
   text 2-15
Overview
   McIDAS 1-3–1-6, 2-1–2-56

P
pack 5-77
pack2 5-77
Package directories 3-6
page 5-27, 5-34
Panels 7-48
Parameters
   command 2-14
grids 6-34
meteorological 5-107–5-112
passing into functions 2-43
valid names C-1–C-4
Parcel temperature and pressure 5-107
PATH 3-4
pcoeff 7-63
Physical units, convert 5-105
Plot
   sky cover 5-49
   vectors 5-49
   windbars and vectors 5-49
plot 5-27, 5-29
Plot, wind speed 5-49
plotwind 5-49
pltdig 5-27, 5-31
pltext 5-31
pltwny 5-49
POES satellites B-8
Point
data
   accessing 6-48–6-61
   characteristics 2-5, 6-49
   error codes 8-43
   file header 6-58
   requests 8-42
   schema definition files 7-43
   send to client 8-43
   serving 8-42–8-44
objects
   reading 6-52
Pointing device 5-18–5-20
Polar orbiting satellites 7-38
Polar stereographic projections 6-35
Positive UC 2-22
Potential temperature 5-107, 5-109
Prefixes, function 2-35
Pressure, compute 5-107
Primary servers 8-5
Print statements 2-48
Programmer’s Manual
   conventions 1-11
   organization 1-10
   using 1-9–1-12
Programming
guidelines 2-52–2-56
McIDAS philosophy 1-7
upgrades, planning for 1-8
using manual 1-9–1-12
Programs
   applications 2-27
   helps 3-15, 4-14
   resident 2-25–2-27
   writing
      C 2-31
      Fortran 2-30
Projections
   conformal 6-69
   equal-area 6-70
   equidistant 6-35
   Lambert conformal 6-35
   Polar stereographic 6-35
   pseudo-Mercator 6-35, 6-70
Pseudo-Mercator projections 6-35, 6-70
pcoeff 7-63
puc 5-63, 7-83
putcur 5-20
putfrm 7-52
Q
qgdas 5-27
qgdash 5-30
qscale 5-27
Quoted text 2-15
R
racae 6-71
Radiometer, Advanced Very High Resolution 7-29
raerac 6-71
Read
   blocks
      calibration 6-27
      comment 6-28
      line prefix 6-25
      navigation 6-27

Index
xvi

McIDAS Programmer’s Manual
Revised 11/98
general weather text data 6-65
image data 6-23
image directory block 6-16
text data 6-62
Receive data 2-7-2-10
Redirection
file 2-10
tables 2-24
Relative humidity 5-107, 5-110
Reporting stations 7-64, 7-65
Requests
client
interpreting 8-16
retrieving 8-17
sending 8-17
general weather text 6-66
header 8-12
image data 6-23
syntax
general weather text 8-45
grid directories 8-38
grid objects 8-34
image directory 8-31
image objects 8-25
observational weather text 8-49
point data 8-42
types 8-23
resgra 7-59
Resident programs
mcenv 2-25
mcimage 2-27
mctext 2-26
Resolution
image 6-12
satellite bands B-4
Return codes 5-70-5-71
rmix 5-108, 5-111
Routines, Fortran callable 2-46

g
geostationary 2-5, 7-34
GMS B-11
GOES VISSR B-7
GVAR B-5, B-6
images 2-5
Meteosat bands B-11
POES B-8
polar orbiting 2-5, 7-38
retrieve sensor information 7-73
Saturation vapor pressure 5-108
savgra 7-59
sbwbyt4 5-79
Scaling 5-27
Schedule file
i/o routines 7-75
word allocation 7-74
Schema definition files 7-43
Scientific utilities 5-107-5-112
sclhgt 5-27, 5-34
scloff 5-27, 5-34
sclo 5-34
sclpnt 5-34
Scripts
cal_init 3-12
mccomp 3-9
nav_init 3-12
shell 2-47
sdest 5-23
Secondary servers 8-7
Selection conditions
grid 6-38
image 6-20
point 6-52
Sensors
AVHRR B-8
OLM 7-38
SSM/I 7-38
servacct 8-12
Servers
APIs 8-5
building 8-10-8-20
categories 8-3
client communication 8-4
data requests 8-7, 8-8
debag 8-21-8-22

SATANNO7 7-73
Satellites
bands for imagery B-4-B-12

McIDAS Programmer's Manual
Revised 11/98

Index
xvii
definition 8-3
ending transaction 8-19
flat file text 6-61
general weather text 8-45
grid data 8-34
grid directories 8-38
image data 8-25
image directory data 8-31
mapping table 8-15
observational weather text 8-49
point data 8-42
primary
description 8-3
list of current 8-5
start 8-7
using 8-13
processing data 8-23
secondary
description 8-3
functions 8-7
using 8-14
timeouts 8-36, 8-39
transmitting data 8-4, 8-5
Shared memory
block 2-21
image frame objects 2-24
redirection tables 2-24
user common 2-22
Shell scripts, writing 2-47
Single precision numbers 5-105
Single-letter commands 2-12, 2-13, 5-20
sizeur 5-20
sked 5-72
SKEDFILE 7-74
skeyin 5-68
skio 7-75
Sky cover 5-49
Sleep 5-72
sndanl 5-108, 5-112
solarp 6-71
Sounder, GVAR B-6
Sounding level B-4
Source files, compiling 3-9, 4-10
Special Sensor for Microwave Imagery (SSM/I) 7-38
Spectral bands 6-13
SSU instrument B-9
Stability parameters 5-112
Standard messages 5-22
Static data 2-6
Station
list of reporting 7-64
tables 2-6
Station pressure 5-108
Status codes 5-16
stc 5-77, 5-78
stdin 8-4
stdout 8-4
Storing arrays 2-45
stralloc 5-86, 5-95
Strings
application limitations 5-57
character tokens 5-60
concatenate 5-95
convert
case 5-96
to double precision values 5-88
to integer values 5-87
make character string 5-57
manipulate characters 5-85–5-106
read from table 5-58
tables 5-56–5-59
text 5-27
Supplemental block 7-6
Suspend applications 5-72
swbty2 2-52
swbty2 5-77, 5-79
swbty4 2-52, 5-77
System
overview 2-21–2-28
utilities 5-56–5-75

T

Table
frame
eenhancements 7-49
graphics 7-59
mapping 8-15
string 5-56–5-59  
VAS calibration 7-84  
Temperature  
along saturated adiabats 5-108  
compute 5-108  
potential temperature 5-108  
virtual 5-108  
Templates  
general weather text request 6-66  
helps A-6  
interface documentation block 2-40, 2-41  
Text  
and Command Window 2-12  
data 6-61–6-66  
display 5-31  
files 7-43  
flat file 6-61  
general weather 6-63–6-66  
generate 5-23  
messages 2-48, 5-22  
output 2-15  
routing 5-23  
strings 5-27  
Time  
convert in string 5-90  
convert to double precision and integers 5-88  
extract from command line 5-11  
increment 5-103  
retrieve current 5-99  
Timeouts 8-36, 8-39, 8-47  
TIP data 7-33, B-9  
Tools  
debug 2-49  
M0ASSERT 2-48  
typcwr 5-20  

User  
accounts 3-3, 3-5, 3-7  
commands 5-5  
common 2-22  
environment 4-3  
interface utilities 5-5–5-21  
User Common  
APIs 5-63, 7-83  
words 7-76–7-83  
Utilities  
conversion 5-75–5-107  
display 5-21–5-74  
dmake 4-11  
scientific 5-107–5-112  
system 5-56–5-75  
user interface 5-5–5-21  

V  
Valid time 6-34  
Validity code 7-7  
Values  
codes for missing 5-69  
double precision 5-88, 5-105  
extact from command line 5-8  
integer 5-87  
user common 7-83  
VAS calibration tables 7-84  
VASTBLS 7-84  
Vectors, plot 5-49  
VISR data 7-23  
VISSR, GOES 7-23  
volnam 6-5, 6-7  

W  
Weather text data  
genral  
accessing 6-63–6-66  
characteristics 2-6  
convert header format 8-46  
error codes 8-46  
reading 6-65  

McIDAS Programmer’s Manual  
Revised 11/98  
Index xix
request syntax 8-45
rules for transmitting 8-46
send to client 8-47
serving 8-45–8-48
observational
convert header format 8-50
error codes 8-50
request syntax 8-49
rules for transmitting 8-50
send to client 8-51
serving 8-49–8-51
Wet bulb temperature 5-108
Wind
  barbs and vectors 5-49
  chill 5-111
  direction 5-107
  speed 5-49, 5-108
Wind chill 5-108
Windows, see Text and Command or Image Window
WMO format 6-63
World coordinates 5-33
Writing
  Dynamic Link Library Modules 2-47
  Fortran-callable routines 2-46
  programs
    C 2-31
    Fortran 2-30
    shell scripts 2-47
wrtex 5-27, 5-31
wtxgser 8-45

Z

zeros 5-77