The
Connection Machine
System

Generic Display Interface Reference Manual for Paris

Version 2.0
November 1991

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, November 1991

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

About This Manual

Objectives of This Manual

This manual provides detailed reference information about the Paris interface to the Generic Display Interface. Separate manuals are available for the C* and CM Fortran interfaces.

Intended Audience

This manual is intended for programmers using the Generic Display Interface Version 2.0. The reader is assumed to be familiar with basic Paris programming.

Revision Information

This manual documents the Generic Display Interface, Version 2.0.

This manual replaces the *Generic Display Interface Reference Manual, Version 5.2*.

Organization of This Manual

The manual is divided into four chapters:

Chapter 1 Introduction to the Generic Display Interface
Provides an overview of the Generic Display library and basic information about how to include the Generic Display routines in your program.

Chapter 2 Workstation and Display Routines
Introduces and provides detailed descriptions of the Generic Display routines that create and control the Generic Display workstation and display.

Chapter 3 Generic Text Routines
Introduces and provides detailed descriptions of the Generic Display text routines.

Chapter 4 Mouse Interface Routines
Introduces and provides detailed descriptions of the Generic Display mouse interface routines.

Related Documents

This manual is one of three that make up the Connection Machine Visualization Programming documentation set. The other two are:

- **Render Reference Manual for Paris*
- *Image File Interface Reference Manual for Paris*

Notation Conventions

The table below displays the notation conventions observed in this manual.

Convention	Meaning
`bold typewriter`	C/Paris, Fortran/Paris, and Lisp/Paris language elements, such as operators, keywords, and function names, when they appear embedded in text or in syntax lines. Also UNIX and CM System Software commands, command options, and file names.
italics	Argument or parameter names and placeholders, when they appear embedded in text or syntax lines.
`typewriter`	Code examples and code fragments.
`% bold typewriter` `typewriter`	In interactive examples, user input is shown in **`bold typewriter`** and system output is shown in `regular typewriter` font.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a back-trace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an Applications Engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
uucp Electronic Mail:	ames!think!customer–support
Telephone:	(617) 234–4000 (617) 876–1111

For Symbolics Users Only

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl–M to create a report. In the mail window that appears, the `To:` field should be addressed as follows:

`To:` customer–support@think.com

Please supplement the automatic report with any further pertinent information.

Chapter 1

Introduction to the Generic Display Interface

The Generic Display Interface is a library of routines that provides a single simple interface through which your application can

- create and initialize Generic Display workstations and displays by having the user select them from a menu (the display provides a display space for images from CM memory; the workstation provides resources to support text and cursor routines)
- transfer image data from CM memory to different types of displays without specialized routines
- query and modify the characteristics of the physical displays from the Generic Display Interface, including the display color maps
- display text strings to any selected generic display
- display, track, and interact with a cursor on the generic display with your workstation mouse

The Generic Display Interface simplifies image display and interaction and allows you to write *device-independent* applications that can be moved to different displays at run time without changing your application.

1.1 The CM Visualization Libraries

The Generic Display Interface is one of three libraries that support visualization programming on the CM. The other two libraries are *Render and the Image File Interface.

As illustrated in Figure 1, these three libraries provide the basic tools for building visualization applications on the CM. With *Render you can process the data produced by your application to create an image in an *image buffer* in CM memory. With the Generic Display Interface you can create and control a display space and write the image buffer to it. Finally, the Image File Interface enables you to store images for future display or processing, or to transfer the image to other graphics environments.

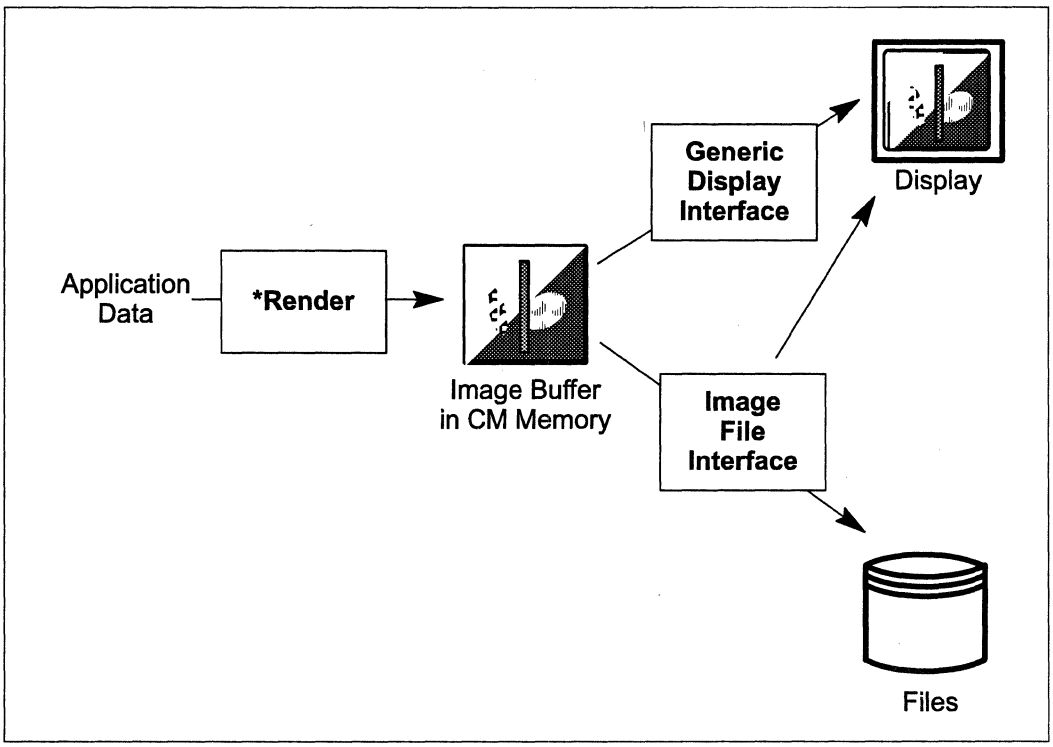


Figure 1. Basic data flow in Connection Machine visualization.

1.1.1 The Image Buffer

The *image buffer* is a CM field or variable used to collect and store pixel values for display. It is a 2D Paris VP set allocated in the size and shape of the image to be displayed. Each virtual processor in the image buffer VP set contains a color value and, if 3D, a z coordinate for the pixel at the corresponding (x, y) location on the display.

*Render and the Generic Display Interface allow you to operate on the image buffer like a virtual display space by specifying locations in screen coordinates. The visualization

libraries assume the right-handed screen coordinate system shown in Figure 2. The origin (0,0) is at the upper left corner of the image, positive x increases to the right, positive y increases toward the bottom of the screen, and positive z increases into the screen. The coordinate values are specified in terms of pixels.

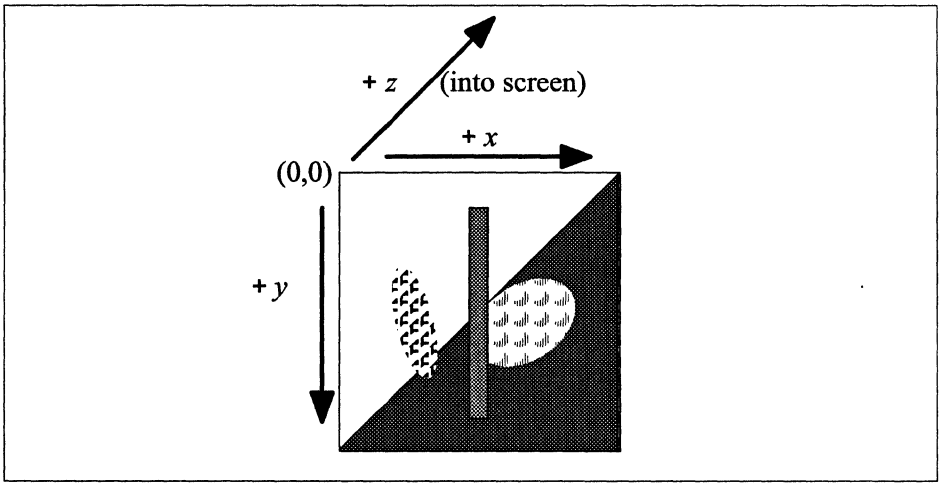


Figure 2. The image buffer coordinate system.

1.1.2 *Render

*Render helps you create and manipulate an image in an image buffer in Connection Machine memory. *Render provides drawing routines that draw points, lines, arrays, and spheres into the image buffer by writing color values into the appropriate elements of the image buffer.

*Render also contains a large collection of graphic math utilities to perform standard graphics math functions on vectors or matrices stored in scalar data structures on the front-end computer or in parallel variables on the CM. For example, functions are provided to create transformation matrices and to perform vector and matrix multiplication.

1.1.3 The Image File Interface

The Image File Interface supports the transfer of images to files in TIFF (Tagged Image File Format), a standard specification for image data files. TIFF is supported by many other

graphics software packages, so you can easily move CM images stored with the Image File Interface to other graphics environments for editing or display. The TIFF format also provides for compression of the image data in the file and stores information about the image that can be used when reading the file back into the Connection Machine system or into some other TIFF reader.

The Image File Interface transfers images between files and an image buffer on the CM, a scalar array on the front-end computer, or even directly to or from a Generic Display Interface display.

1.2 The Generic Display Interface

The Generic Display Interface is made up of the following major components:

- **Workstation Routines**

  These routines allow you to create, initialize, and select any X11 server as a generic workstation by choosing the server from a menu. The workstation is used to provide support for the Generic Display text and cursor facilities.

  These routines are described in Chapter 2 of this manual.

- **Display Routines**

  These routines allow you to establish any X11 window or a CM framebuffer as a generic display, using a menu similar to that for the workstation routines. Routines are also provided that allow you to transfer images to the selected display, and to change the set-up of the display.

  These routines are also described in Chapter 2.

- **Text Routines**

  These routines allow you to display text on the selected display using two built-in fonts or any X11 font available on the generic workstation. Also included are support routines that allow you to control the text color and drawing mode, and to position text effectively.

  These routines are described in Chapter 3.

- **Mouse Routines**

  The mouse routines allow you to display a cursor on the selected generic display and control it with your workstation mouse. A full set of support routines are provided that allow you to control the behavior and appearance of the display cursor.

1.3 Using the Generic Display Routines

To use the Generic Display routines you must include the appropriate header file in your program and link with the supporting libraries when compiling.

C/Paris. For C/Paris programs you must include the header file **`display.h`** as follows:

```
#include <cm/display.h>
```

and you must link with the following libraries when compiling:

```
cc prog.c -lcmsr -lX11 -lparis -lm
```

Fortran/Paris. For Fortran/Paris programs you must include the header file **`display-fort.h`** as follows:

```
INCLUDE '/usr/include/cm/display-fort.h'
```

and you must use the following links to compile:

```
f77 prog.f -lcmsr -lX11 -lparisfort -lparis -lm
```

Lisp/Paris. For Lisp programs you must use a band in which the graphics package has been loaded. If necessary, you can load it by entering:

```
(lcmw:load-optional-system 'graphics)
```

Chapter 2

Workstation and Display Routines

This chapter documents the Generic Display Interface routines that are used to create, select, and control generic displays and workstations. With the workstation and display routines described in this chapter you can

- establish and manage an X11 server or a CM framebuffer as a Generic Display display
- establish and manage any X11 server as a Generic Display workstation (the selected workstation's resources support the Generic Display text and mouse capabilities)
- transfer images from CM memory to the selected display

A *generic display* can be either an X11 server's display or a CM framebuffer and its attached monitor. The generic display provides the display space for images transferred from the CM system and for the generic workstation's text and cursor display.

A *generic workstation* is an X11 server and a generic display. If the generic display is an X11 display, the generic workstation and display are one and the same. The generic workstation supplies text fonts and mouse and cursor support for the Generic Display Interface.

If your application does not use text display or mouse interaction, you can establish a generic display without any generic workstation. A generic workstation, on the other hand, always includes a generic display; many of the Generic Display workstation routines implicitly operate on its associated display as well.

The Generic Display Interface allows you to select the physical devices you will use as your workstation and display from within your program at run time. Other Generic Display routines that write to, query, or control the workstation and display automatically operate on this *currently selected* workstation or display. This mechanism allows you to write your program without being tied to specific physical devices. For example, you could use your local workstation to develop and debug your visualization in black and white and then,

without changing your program, display the image on a color workstation or a CM framebuffer for final editing or presentation.

The next section of this chapter gives an overview of the basic use of the Generic Display Interface.

Later sections provide more detailed information on the many routines that allow you to control the generic workstation and display, and integrate them with other X or CM framebuffer applications.

The last section of this chapter provides an alphabetical reference to all the Generic Display workstation and display routines.

The Generic Display text display routines are documented in Chapter 3 of this manual; the cursor and mouse routines are documented in Chapter 4.

2.1 Overview

This overview describes the easiest way to get started using the Generic Display workstation and display. However, the Generic Display also includes lower-level routines that allow you more direct control over the selection and configuration of workstations and displays. See the later sections of this chapter, which detail all the routines, for more information.

2.1.1 Selecting a Generic Display Workstation and Display

A single Generic Display routine, **`CMSR_select_workstation_menu`**, allows you to create both a generic display and a workstation. **`CMSR_select_display_menu`** creates only a generic display.

You can specify the display and workstation in one of three ways:

- Choose Generic Display defaults with **`CMSR_set_display_default`** or **`CMSR_set_workstation_default`**.
- Use the environment variables **`CM_DISPLAY`** or **`CM_WORKSTATION`**.

- Choose a display and workstation from menus displayed by **CMSR_select_workstation_menu** or a display from menus displayed by **CMSR_select_display_menu**.

2.1.2 Setting Workstation and Display Defaults

If you know you will always want to use the same workstation and/or display for your graphics work, you can save time by setting default values with global or environmental variables.

If the default display variable set with **CMSR_set_display_default** or the **CM_DISPLAY** environment variable is set, **CMSR_select_display_menu** and **CMSR_select_workstation_menu** establish the X11 server or CM framebuffer named there as the current display.

Similarly, if a CM framebuffer is selected as the display and a workstation default has been set with **CMSR_set_workstation_default** or in the **CM_WORKSTATION** environment variable, **CMSR_select_workstation_menu** establishes the X11 server named as the current workstation.

The string used as the display default or as the **CM_DISPLAY** environment variable can be any of the following:

- the name of an X11 server (such as **LEANDER:0**)
- the string **CMFB** to identify the default framebuffer
- the string **CMFB8** to identify the default framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24** to identify the default framebuffer and set it to 24-bits-per-pixel mode
- the string **CMFB:** followed immediately by the location string to identify a particular framebuffer display
- the string **CMFB8:** followed immediately by the location string to identify a particular framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24:** followed immediately by the location string to identify a particular framebuffer and set it to 24-bits-per-pixel mode

The string used as the workstation default or the **CM_WORKSTATION** environment variable must be the name of an X11 server.

2.1.3 The Selection Menu

If no Generic Display default or environment variable is set, **CMSR_select_workstation_menu** and **CMSR_select_display_menu** offer the user a menu of the available displays. For example, the display menu might look like this:

```
   Available Display Menu
D    The X display 'local-workstation:0'
X    Any X window display
1    CM Framebuffer: Computer Center
2    CM Framebuffer: Graphics Lab

Choose a display (either 'D', 'X', or a number):
```

If you enter 'X', the menu prompts you for the name of an X window display:

```
Enter the name of an X window display
(Example: 'Leander:0') ==>
```

If you select an X11 server, **CMSR_select_display_menu** initializes it and selects it as the current generic display and then returns; **CMSR_select_workstation_menu** initializes and selects the X11 server as *both* the current display and the current workstation, and then returns.

If you select a CM framebuffer, **CMSR_select_display_menu** initializes it and selects it as the current generic display and then returns. But **CMSR_select_workstation_menu**, after initializing and selecting the framebuffer as the current display, continues with a similar process to create a Generic Display workstation.

First, **CMSR_select_workstation_menu** checks to see whether a workstation default has been set with **CMSR_set_workstation_default**. If no default is set, it checks the **CM_WORKSTATION** environment variable. Finally, if neither of these is set, it presents a workstation menu. Like the display, the workstation specified or chosen is then initialized and selected as the current Generic Display workstation.

The selected display and workstation are then used for all further interactions with the other Generic Display routines. The current display provides the display space for the Generic Display image transfer routines. The current workstation serves as both the font host for

the Generic Display text display and the mouse host for the Generic Display cursor facilities.

2.1.4 Using the Generic Display

Once a display is established as the current display, only one other routine, **CMSR_write_to_display**, is needed to transfer an image from an *image buffer* in CM memory.

The Image Buffer

The source of the image to be transferred to the selected display is an *image buffer* in CM memory. An image buffer is a VP set in CM memory with a two-dimensional geometry.

The length of the axes of the image buffer VP set correspond to the resolution of the image to be displayed, 1 virtual processor to each pixel. Axis 0 of the geometry maps to the screen's x (horizontal) axis, and axis 1 of the geometry maps to the screen's y (vertical) axis. The x coordinate increases to the right, and the y coordinate increases downward.

The length of the image buffer field containing the color data to be transferred to the display should be the same as the depth of the display. If the field length is longer, the high-order bits are lost. If the field length is shorter than the depth of the window, an error is signaled.

Displaying the Image Buffer

When you call **CMSR_write_to_display**, the origin of the image buffer field (0,0) is displayed at the upper left corner of the display and the color value in each virtual processor is assigned to the corresponding pixel of the generic display screen.

If the image is smaller than the display, the portions of the display to the right and below the dimensions of the image are left unchanged.

If the image is larger than the display, portions of the image that extend to the right and below the display space are clipped.

Framebuffer-Ordered Geometries

The transfer of fields of color data between the image buffer and the selected display using the Generic Display Interface operations **CMSR_write_to_display**, **CMSR_write_to_display_1**, **CMSR_read_from_display**, and **CMSR_read_from_display_1** can be optimized by using image buffers created with *framebuffer ordering.*

The routine **CMFB_create_cmfb_geometry** allocates and returns a 2D geometry of a specified *width* and *height. Width* specifies the length of axis 0 of the geometry and maps to the screen's *x* (horizontal) axis. *Height* specifies the length of axis 1 and maps to the screen's *y* (vertical) axis. Both axes are created with framebuffer ordering.

Framebuffer-ordered geometries are intended to be used only as image buffers. While image transfers to the CM framebuffer are faster, Paris NEWS communication functions operate much more slowly on a framebuffer-ordered VP set. The NEWS function must perform a send to reorder a framebuffer-ordered geometry before the NEWS operation can be completed.

If you do not use NEWS functions in the image buffer, it is recommended that you use a framebuffer-ordered geometry as an image buffer. The Generic Display Interface I/O functions accept a NEWS-ordered geometry as an image buffer, but display performance is slowed significantly. These operations must perform a send to "shuffle" the field into framebuffer order before transferring it to the CM framebuffer. I/O performance to an X Window System or Symbolics display is unaffected by the choice of ordering.

2.1.5 Using the Generic Workstation

The currently selected generic workstation and its associated display are automatically used by the other Generic Display routines.

For example, **CMSR_get_mouse_point** displays a cursor on the currently selected display that is controlled by the physical mouse connected to the currently selected workstation. Similarly, **CMSR_display_text** writes text to the current display in any of the X11 text fonts available on the current workstation.

The text routines are described in Chapter 3 of this manual; the mouse routines are described in Chapter 4.

2.1.6 Low-Level Access

The Generic Display Interface controls the X Window System servers selected as the generic display or workstation with calls to `Xlib`. No widgets are used. When the generic display is a CM framebuffer, the interface uses calls to the CMFB Display Operations.

Normally you do not need to access this level of the interface at all. But if you wish to integrate your Generic Display application with an existing X11 application, or need to drive the framebuffer directly, the Generic Display provides routines that allow you to access the lower level of the interface.

X11 routines are described in Section 2.7; the CMFB routines are discussed in Section 2.8.

2.2 Workstation Routines

This section describes the Generic Display routines that allow you to select and control an X Window System host as a Generic Display workstation. The currently selected workstation supplies the resources to support the Generic Display text drawing routines and cursor interaction routines.

NOTE: A Generic Display workstation always includes a generic display. Routines that operate on the workstation also operate on the display associated with it. However, routines that operate on the display do not operate on the workstation. Therefore be careful to use only the workstation or only the display routines throughout a Generic Display application. Intermixing workstation and display routines can lead to unintended results.

For example, if **CMSR_select_display** is used to change a workstation's display from one X11 server to another, Generic Display text routines do not display correctly because the workstation resources are no longer associated with the current display. To change the display in this situation you must use **CMSR_select_workstation** to change both the workstation and the display.

The routines described here are:

CMSR_make_window_type

Creates Generic Display window type data structure for a specified visual class and depth.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_make_window_type
            (visual_class, depth, class_required_p, depth_required_p)

int   visual_class;
int   depth;
int   class_required_p;
int   depth_required_p;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_MAKE_WINDOW_TYPE
&             (visual_class, depth, class_required_p, depth_required_p)

INTEGER  visual_class
INTEGER  depth
LOGICAL  class_required_p
LOGICAL  depth_required_p
```

Lisp Syntax

```
CMSR:make-window-type (&key visual-class depth
                            class-required-p depth-required-p)
```

ARGUMENTS

visual_class The X Window System visual class to be specified in the returned window type. Valid values are:

- PseudoColor
- StaticColor
- DirectColor
- TrueColor
- GrayScale
- StaticGray

depth An integer specifying the number of bits of color information to be maintained by the display for each pixel. This *depth* argument is the same as the *desired_bits_per_pixel* argument in other Generic Display routines.

class_required_p A predicate indicating whether the *visual_class* is desired or required.

If *class_required_p* is true (.TRUE. in Fortran, non-NULL in C, non-nil in Lisp), the window type returned instructs the routine creating the Generic Display display to accept only displays with exactly the specified visual class.

If *class_required_p* is false (.FALSE. in Fortran, NULL in C, nil in Lisp), the window type returned instructs the routine creating the Generic Display display to initialize the display to the supported visual class closest to the *visual_class* argument.

depth_required_p A predicate indicating whether *depth* is desired or required.

If *depth_required_p* is true (.TRUE. in Fortran, non-NULL in C, non-nil in Lisp), the window type returned instructs the routine creating the Generic Display display that it can only accept displays with exactly the specified depth.

If *class_required_p* is false (.FALSE. in Fortran, NULL in C, nil in Lisp), the window type returned instructs the routine creating the Generic Display display that it can initialize the display to the supported depth closest to the *depth* argument.

DESCRIPTION

CMSR_make_window_type returns a Generic Display window type that may be used as an argument to the following Generic Display routines:

- **CMSR_select_workstation_menu**
- **CMSR_create-workstation-menu**
- **CMSR_select_display_menu**
- **CMSR_create-display-menu**

The window type argument is an *opaque* data structure (i.e., not accessible to the user) carrying information on both the visual class and depth of the window to be requested. The depth of a window is the number of bits per pixel of color information, or *color planes*, supported by the window. The visual class specifies how the color information is to be interpreted. The meaning of the visual classes is as follows:

- **PseudoColor**
  The pixel bits are interpreted as a single value and used as an index to an entry in a writable color map of RGB color values.
- **StaticColor**
  The pixel bits are interpreted in the same way as PseudoColor, but the color map is predefined and cannot be changed.
- **DirectColor**
  The pixel bits are decomposed into separate red, green, and blue values that are used to index separate, writable, red, green, and blue color maps.
- **TrueColor**
  The pixel bits are interpreted in the same way as DirectColor but the color maps are predefined and cannot be changed.
- **GrayScale**
  The pixel bits are interpreted as a single value and used as an index to an entry in a writable color map of grayscale intensities.
- **StaticGray**
  The pixel bits are interpreted in the same way as GrayScale, but the color map is predefined and cannot be changed.

CMSR_select_workstation_menu

Creates, initializes, and selects as current a Generic Display workstation and display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_select_workstation_menu
                    (window_type, desired_width, desired_height)

int             window_type;
unsigned int    desired_width;
unsigned int    desired_height;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SELECT_WORKSTATION_MENU
&                   (window_type, desired_width, desired_height)

INTEGER  window_type
INTEGER  desired_width
INTEGER  desired_height
```

Lisp Syntax

```
CMSR:select-workstation-menu
            (&optional window-type desired-width desired-height)
```

ARGUMENTS

window_type Can be either an integer specifying the *depth* of the display to be opened or the return value of **CMSR_make_window_type**.

The *depth* (also called the *desired_bits_per_pixel*) is the number of bits of color information maintained by the display for each

pixel. The number of bits that can actually be supported is determined by the display hardware. The Generic Display matches the *depth* as closely as possible. Call **CMSR_display_bits_per_pixel** to learn the actual depth of the initialized display.

CMSR_make_window_type returns a Generic Display window type based on *depth* and *visual_class* arguments supplied to it. The *visual_class* argument is one of the X Window System's visual classes: GrayScale, StaticGray, PseudoColor, StaticColor, DirectColor, or TrueColor. See the description of **CMSR_make_window_type** in this manual for more information.

desired_width — The width, in pixels, of the display window you wish to create. The width of the display is the *x*, or horizontal, dimension of the display. The Generic Display matches *desired_width* as closely as possible on the selected display. Call **CMSR_display_width** to determine the actual width of the created display.

If *desired_width* is specified as NULL or 0 in C, 0 in Fortran, or not specified in Lisp, the width of the display defaults to 256.

desired_height — The height, in pixels, of the display window you wish to create. The height of the display is the *y*, or vertical, dimension of the window. The Generic Display matches *desired_height* as closely as possible on the selected display. Call **CMSR_display_width** to determine the actual width of the initialized display.

DESCRIPTION

CMSR_select_workstation_menu creates and selects both a current workstation and its associated display. You can immediately operate on the workstation and display with other Generic Display routines.

NOTE: Generic Display routines that operate on the workstation also operate on the display associated with it. However, routines that operate on the display do not operate on the workstation. Therefore, be careful to use only the workstation or only the display routines throughout a Generic Display application. Intermixing workstation and display routines can lead to unintended results.

For example, if **CMSR_select_display** is used to change a workstation's display from one X11 server to another, Generic Display text routines do not display correctly because the workstation resources are no longer associated with the current display. To

change the display in this situation you must use **CMSR_select_workstation** to change both the workstation and the display.

The current display is the display space for the Generic Display routines that read and write images. It can be either an X11 window or a CM framebuffer.

The current workstation provides X11 resources and the physical mouse to support the Generic Display system's text and mouse routines. It must be an X11 server. For example, the Generic Display text routines can use the current workstation's X11 fonts to draw text onto the Generic Display display. Similarly, the Generic Display mouse routines use the current workstation's resources to handle cursor tracking and interaction.

You can specify the display and workstation to use by any of the following:

- setting Generic Display defaults with **CMSR_set_display_default** or **CMSR_set_workstation_default**
- setting the environment variables **CM_DISPLAY** or **CM_WORKSTATION**
- choosing a display and workstation from menus displayed by **CMSR_select_workstation_menu**

CMSR_select_workstation_menu first attempts to create the display by checking the default display variable set with **CMSR_set_display_default**. If no default is set, it checks the **CM_DISPLAY** environment variable. Finally, if neither of these is set, it presents the user with a menu of the available displays.

If the selected display is an X11 server, **CMSR_select_workstation_menu** initializes it as closely as possible to the specified *window_type*, *width*, and *depth* arguments and selects it as the current display. If another display is currently selected, that display is deselected. **CMSR_select_workstation_menu** also automatically initializes and selects this server as the current workstation, and then returns.

If the selected display is a CM framebuffer, **CMSR_select_workstation_menu** initializes the framebuffer and selects it as the current display. The display color mode is set as closely as possible to *window_type,* and the display height and width is set to the height and width of the framebuffer's attached monitor. (For framebuffers, *desired_width*, and *desired_height* are not used; the display space is always the entire monitor screen.) Then, because a CM framebuffer cannot be used as a workstation, **CMSR_select_workstation_menu** continues with a similar process to create a Generic Display workstation.

First, it checks to see whether a workstation default has been set with **CMSR_set_workstation_default**. Then, if no default is set, it checks the **CM_WORKSTATION** environment variable. And, finally, if neither of these is set, it presents a workstation

menu. The workstation specified or chosen is then initialized and selected as the current Generic Display workstation.

The string used as the display default or as the **CM_DISPLAY** environment variable can be any of the following:

- the name of an X11 server (such as **LEANDER:0**)
- the string **CMFB** to identify the default framebuffer
- the string **CMFB8** to identify the default framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24** to identify the default framebuffer and set it to 24-bits-per-pixel mode
- the string **CMFB:** followed immediately by the location string to identify a particular framebuffer display
- the string **CMFB8:** followed immediately by the location string to identify a particular framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24:** followed immediately by the location string to identify a particular framebuffer and set it to 24-bits-per-pixel mode

The string used as the workstation default or the **CM_WORKSTATION** environment variable must be the name of an X11 server.

ERRORS

An error is signaled if the selected display or workstation is changed while the Generic Display mouse is grabbed.

CMSR_create_workstation_menu

Creates, but does *not* select, a Generic Display display and workstation.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_workstation_t
  CMSR_create_workstation_menu
                    (window_type, desired_width, desired_height)

int             window_type;
unsigned int    desired_width;
unsigned int    desired_height;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_CREATE_WORKSTATION_MENU
&                   (window_type, desired_width, desired_height)

INTEGER  window_type
INTEGER  desired_width
INTEGER  desired_height
```

Lisp Syntax

```
CMSR:create-workstation-menu
            (&optional window_type, desired_width, desired_height)
```

ARGUMENTS

window_type — Can be either an integer specifying the *depth* of the display to be opened or the return value of **CMSR_make_window_type**.

The *depth* (also called the *desired_bits_per_pixel*) is the number of bits of color information maintained by the display for each

pixel. The number of bits that can actually be supported is determined by the display hardware. The Generic Display matches the *depth* as closely as possible. Call **CMSR_display_bits_per_pixel** to learn the actual depth of the initialized display.

CMSR_make_window_type returns a Generic Display window type based on *depth* and *visual_class* arguments supplied to it. The *visual_class* argument is one of the X Window System visual classes: GrayScale, StaticGray, PseudoColor, StaticColor, DirectColor, or TrueColor. See the description of **CMSR_make_window_type** in this manual for more information.

desired_width The width, in pixels, of the display window you wish to create. The width of the display is the *x*, or horizontal, dimension of the display. The Generic Display matches *desired_width* as closely as possible on the physical display. Call **CMSR_display_width** to determine the actual width of the initialized display.

If *desired_width* is specified as NULL (0 in C, 0 in Fortran, or not specified in Lisp), the width of the display defaults to 256.

desired_height The height, in pixels, of the display window you wish to create. The height of the display is the *y*, or vertical, dimension of the display. The Generic Display matches *desired_height* as closely as possible on the physical display. Call **CMSR_display_width** to determine the actual width of the initialized display.

DESCRIPTION

CMSR_create_workstation_menu creates both a Generic Display *display* and a Generic Display *workstation*, and then returns a **CMSR_workstation_t** data structure identifying the workstation *and its associated display*. To use the workstation and display with other Generic Display routines, you must first select them by calling **CMSR_select_workstation**.

NOTE: Generic Display routines that operate on the workstation also operate on the display associated with it. However, routines that operate on the display do not operate on the workstation. Therefore, be careful to use workstation and display routines consistently throughout a Generic Display application. Intermixing workstation and display routines can lead to unintended results.

For example, if you attempt to change a workstation's display from one X11 server to another by calling **CMSR_select_display**, Generic Display text routines do not display correctly because the workstation resources are no longer associated with the current display. To change the display in this situation you must use **CMSR_select_workstation** to change both the workstation and the display.

You can specify which display and workstation to use in one of three ways:

- Choose Generic Display defaults with **CMSR_set_display_default** or **CMSR_set_workstation_default**.
- Use the environment variable **CM_DISPLAY** or **CM_WORKSTATION**.
- Choose a display and workstation from menus displayed by **CMSR_select_workstation_menu** or a display from menus displayed by **CMSR_select_display_menu**.

CMSR_create_workstation_menu first attempts to create the display by checking the default display variable set with **CMSR_set_display_default**. If no default is set, it checks the **CM_DISPLAY** environment variable. Finally, if neither of these is set, it presents a menu of the available displays.

If the display named in a variable or selected from the menu is an X11 server, **CMSR_create_workstation_menu** establishes the server as both a display and a workstation, and then returns. The display is initialized as closely as possible to the *window_type*, *desired_width*, and *desired_height* specified in your call to **CMSR_create_workstation_menu**.

If the display named in a variable or selected from the menu is a CM framebuffer, **CMSR_create_workstation_menu** initializes the framebuffer as a display, matching the specified *window_type* as closely as possible and setting the display height and width to the height and width of the framebuffer's attached monitor. (For framebuffers, *desired_width* and *desired_height* have no meaning.) Then, because a CM framebuffer cannot be used as a workstation, **CMSR_create_workstation_menu** continues with a similar process to create a Generic Display workstation.

First, it checks to see whether a workstation default has been set with **CMSR_set_workstation_default**. Then, if no default is set, it checks the **CM_WORKSTATION** environment variable. Finally, if neither of these is set, it presents a workstation menu. The workstation specified or chosen is then initialized as a Generic Display workstation.

The string used as the display default or as the **CM_DISPLAY** environment variable can be any of the following:

- the name of an X11 server (such as **LEANDER:0**)
- the string **CMFB** to identify the default framebuffer
- the string **CMFB8** to identify the default framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24** to identify the default framebuffer and set it to 24-bits-per-pixel mode
- the string **CMFB:** followed immediately by the location string to identify a particular framebuffer display
- the string **CMFB8:** followed immediately by the location string to identify a particular framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24:** followed immediately by the location string to identify a particular framebuffer and set it to 24-bits-per-pixel mode

The string used as the workstation default or the **CM_WORKSTATION** environment variable must be the name of an X11 server.

CMSR_create_display_workstation

Creates and initializes a Generic Display workstation for an existing display.

SYNTAX

C Syntax

```
#include <cm/display.h>
CMSR_workstation_t
  CMSR_create_display_workstation (generic_display)
CMSR_display_t generic_display;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_CREATE_DISPLAY_WORKSTATION
&                                                    (generic_display)
INTEGER  generic_display
```

Lisp Syntax

```
CMSR:create-display-workstation (&optional generic–display)
```

ARGUMENTS

generic_display A **CMSR_display_t** data structure identifying a previously created Generic Display display.

The *generic_display* identifier for the currently selected display can be returned with **CMSR_selected_display**.

DESCRIPTION

CMSR_create_display_workstation creates a workstation associated with a previously created Generic Display display and returns a **CMSR_workstation_t** structure identifying it. If *generic_display* is NULL, the specified display defaults to the currently selected generic display.

CMSR_select_workstation
CMSR_deselect_workstation

Selects (deselects) an existing workstation and its display as current for the Generic Display system.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_select_workstation (workstation)

CMSR_workstation_t workstation;

void
  CMSR_deselect_workstation ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SELECT_WORKSTATION (workstation)

INTEGER  workstation

SUBROUTINE CMSR_DESELECT_WORKSTATION ()
```

Lisp Syntax

```
CMSR:select-workstation (workstation)

CMSR:deselect-workstation ( )
```

ARGUMENTS

workstation

A **CMSR_workstation_t** data structure identifying a Generic Display workstation and its associated display.

The workstation identifier is created and returned by **CMSR_create_workstation_menu** or **CMSR_create_display_workstation**. You can get the identifier of the currently selected workstation by calling **CMSR:selected_workstation**.

DESCRIPTION

CMSR_select_workstation selects the specified workstation and its associated display as the current workstation and display. The workstation and display must be selected before other Generic Display routines can operate on them.

CMSR_deselect_workstation deselects the selected workstation and its associated display.

CMSR_selected_workstation
CMSR_workstation_type

Returns identifier (workstation type) of currently selected Generic Display workstation.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_workstation_t
  CMSR_selected_workstation ()

CMSR_workstation_type_t
  CMSR_workstation_type ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_SELECTED_WORKSTATION ()

INTEGER FUNCTION CMSR_WORKSTATION_TYPE ()
```

Lisp Syntax

```
CMSR:selected-workstation ()
CMSR:workstation-type ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_selected_workstation returns the **CMSR_workstation_t** data structure identifying the currently selected workstation. This identifier can be used as an argument to other Generic Display routines that control or return information about the current workstation.

CMSR_workstation_type returns the type of the currently selected workstation. In Release 2.0 of the Generic Display system, the workstation type can only be **:x-workstation** in Lisp or **CMSR_x_workstation** in C or Fortran.

CMSR_workstation_display

Returns the identifier of the display for the current Generic Display workstation.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_display_t
  CMSR_workstation_display ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_WORKSTATION_DISPLAY ()
```

Lisp Syntax

```
CMSR:workstation-display ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_workstation_display returns the display identifier of the generic display associated with the currently selected workstation.

CMSR_set_workstation_default
CMSR_get_workstation_default

Sets (returns) the default Generic Display workstation name.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_workstation_default (string)

char *string;

char *
  CMSR_get_workstation_default ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_WORKSTATION_DEFAULT (string)

CHARACTER*(*) string

CHARACTER*(*) FUNCTION CMSR_GET_WORKSTATION_DEFAULT ()
```

Lisp Syntax

```
CMSR:set-workstation-default (string)

CMSR:get-workstation-default ()
```

ARGUMENTS

string	A character string naming the workstation host to use for the Generic Display workstation. Currently, this name must be the name of an X11 server.

DESCRIPTION

CMSR_set_workstation_default sets the Generic Display default workstation name to *name*. When the default workstation name is set, **CMSR_select_workstation_menu** or **CMSR_create_workstation_menu** automatically uses this default to create the Generic Display workstation, bypassing both the workstation environment variable and the workstation menu.

CMSR_get_workstation_default returns a string containing the default workstation name currently set.

CMSR_deallocate_workstation

Deallocates resources of a Generic Display workstation and its display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_deallocate_workstation (workstation)

CMSR_workstation_t    workstation
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_DEALLOCATE_WORKSTATION (workstation)
```

Lisp Syntax

```
CMSR:deallocate-workstation (&optional workstation)
```

ARGUMENTS

workstation — A **CMSR_workstation_t** data structure identifying a Generic Display workstation and its associated display.

The workstation identifier is created and returned by **CMSR_create_workstation_menu** or **CMSR_create_display_workstation**. You can get the identifier of the currently selected workstation by calling **CMSR_selected_workstation**.

DESCRIPTION

`CMSR_deallocate_workstation` deallocates the specified workstation and its associated display, freeing up all memory associated with them. If the workstation is the currently selected workstation, the workstation and display are also deselected.

If you specify NULL in C or 0 in Fortran for the *workstation* argument, then the currently selected workstation is deallocated.

2.3 Display Routines

This section describes the Generic Display routines that allow you to select and control an X11 server or CM framebuffer as a generic display. The currently selected display provides the display space for the Generic Display image display, text drawing, and cursor interaction routines.

NOTE: A Generic Display workstation always includes a generic display. Routines that operate on the workstation also operate on the display associated with it. However, routines that operate on the display do not operate on the workstation. Therefore, be careful to use only the workstation or only the display routines throughout a Generic Display application. Intermixing workstation and display routines can lead to unintended results.

For example, if **CMSR_select_display** is used to change a workstation's display from one X11 server to another, Generic Display text routines do not display correctly because the workstation resources are no longer associated with the current display. To change the display in this situation you must use **CMSR_select_workstation** to change both the workstation and the display.

This section describes the Generic Display routines that operate on the current display:

CMSR_select_display_menu

Presents a menu of available displays. Initializes and selects as current the display chosen by the user.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_select_display_menu
                        (window_type, desired_width, desired_height);
int         window_type;
unsigned int desired_width;
unsigned int desired_height;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SELECT_DISPLAY_MENU
&                       (window_type, desired_width, desired_height)
INTEGER  window_type
INTEGER  desired_width
INTEGER  desired_height
```

Lisp Syntax

```
CMSR:select-display-menu (&optional
                        window-type desired-width desired-height)
```

ARGUMENTS

window_type Can be either an integer specifying the *depth* of the display to be opened or the result of calling **CMSR_make_window_type**.

The *depth* (also called the *desired_bits_per_pixel*) is the number of bits of color information maintained by the display for each pixel. The number of bits that can actually be supported is

determined by the display hardware. The Generic Display matches the *depth* as closely as possible. Call **CMSR_display_bits_per_pixel** to learn the actual depth of the initialized display.

CMSR_make_window_type creates a Generic Display window type based on *depth* and *visual_class* arguments supplied to it. The *visual_class* argument is one of the X Window System visual classes: GrayScale, StaticGray, PseudoColor, StaticColor, DirectColor, or TrueColor. See the description of **CMSR_make_window_type** in this manual for more information.

desired_width The desired width, in pixels, of the display window to be created. The Generic Display Interface system attempts to match the *desired_width* as closely as possible. If *desired_width* is not specified or is specified as NULL or 0, the width of the display defaults to 256. Call **CMSR_display_width** to determine the actual width of the initialized display.

If the display is a framebuffer, *desired_width* has no effect. The width of a framebuffer display is always equal to the maximum width allowed by the attached monitor type.

desired_height The desired height, in pixels, of the display window to be created. The Generic Display Interface system attempts to match the *desired_height* as closely as possible. If *desired_height* is not specified or is specified as NULL or 0, the height of the display defaults to 256. Call **CMSR_display_height** to determine the actual height of the initialized display.

If the display is a framebuffer, *desired_height* has no effect. The height of a framebuffer display is always equal to the maximum height allowed by the attached monitor type.

DESCRIPTION

CMSR_select_display_menu creates a Generic Display display and selects it as the current display. You can immediately operate on the display with other Generic Display routines. The current display is the display space for the Generic Display routines that read and write images. It can be either an X11 window or a CM framebuffer.

You can specify the display you wish **CMSR_select_display_menu** to create and select by any of the following:

- setting a Generic Display default with **CMSR_set_display_default**
- setting the environment variable **CM_DISPLAY**
- choosing a display from the menu displayed by **CMSR_select_display_menu**

CMSR_select_display_menu, first attempts to create the display by checking the default display variable set with **CMSR_set_display_default**. If no default is set, it checks the **CM_DISPLAY** environment variable. Finally, if neither of these is set, it presents the user with a menu of the available displays.

If the selected display is an X11 server, **CMSR_select_display_menu** initializes it as closely as possible to the specified *window_type*, *width*, and *depth* arguments and selects it as the current display.

If the selected display is a CM framebuffer, **CMSR_create_display_menu** initializes the framebuffer and selects it as the current display. The display color mode is set as closely as possible to *window_type,* and the display height and width is set to the height and width of the framebuffer's attached monitor. For framebuffers, *desired_width*, and *desired_height* are not used; the display space is always the entire monitor screen.

If another display is currently selected when **CMSR_select_display_menu** is called, that other display is deselected.

The string used as the display default or as the **CM_DISPLAY** environment variable can be any of the following:

- the name of an X11 server (such as **LEANDER:0**)
- the string **CMFB** to identify the default framebuffer
- the string **CMFB8** to identify the default framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24** to identify the default framebuffer and set it to 24-bits-per-pixel mode
- the string **CMFB:** followed immediately by the location string to identify a particular framebuffer display
- the string **CMFB8:** followed immediately by the location string to identify a particular framebuffer and set it to 8-bits-per-pixel mode

- the string **CMFB24:** followed immediately by the location string to identify a particular framebuffer and set it to 24-bits-per-pixel mode

The string used as the workstation default or the **CM_WORKSTATION** environment variable must be the name of an X11 server.

CMSR_create_display_menu

Presents a menu of available displays. Initializes, but does *not* make current, the display chosen by the user.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_display_t
  CMSR_create_display_menu
                    (window_type, desired_width, desired_height);
int             window_type;
unsigned int    width;
unsigned int    height;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CREATE_DISPLAY_MENU
&                   (window_type, desired_width, desired_height)
INTEGER  window_type
INTEGER  width
INTEGER  height
```

Lisp Syntax

```
CMSR:create-display-menu (&optional window-type
                                    desired-width desired-height)
```

ARGUMENTS

window_type Can be either an integer specifying the *depth* of the display to be opened or the result of calling **CMSR_make_window_type**.

The *depth* (also called the *desired_bits_per_pixel*) is the number of bits of color information maintained by the display for each

pixel. The number of bits that can actually be supported is determined by the display hardware. The Generic Display matches the *depth* as closely as possible. Call **CMSR_display_bits_per_pixel** to learn the actual depth of the initialized display.

CMSR_make_window_type creates a Generic Display window type based on *depth* and *visual_class* arguments supplied to it. The *visual_class* argument is one of the X Window System visual classes: GrayScale, StaticGray, PseudoColor, StaticColor, DirectColor, or TrueColor. See the description of **CMSR_make_window_type** in this manual for more information.

desired_width The width, in pixels, of the display window you wish to create. The width of the display is the *x*, or horizontal, dimension of the display. The Generic Display matches *desired_width* as closely as possible on the selected display. Call **CMSR_display_width** to determine the actual width of the created display.

If *desired_width* is specified as NULL (0 in C, 0 in Fortran, or not specified in Lisp), the width of the display defaults to 256.

desired_height The height, in pixels, of the display window you wish to create. The height of the display is the *y*, or vertical, dimension of the display. The Generic Display matches *desired_height* as closely as possible on the selected display. Call **CMSR_display_width** to determine the actual width of the initialized display.

DESCRIPTION

CMSR_create_display_menu creates a Generic Display *display*, initializes it, and returns a **CMSR_display_t** data structure identifying the display. Before using the display, you must select it as the current display by calling **CMSR_select_display**.

You can specify the display to be created by **CMSR_create_display** by any of the following:

- setting a Generic Display default with **CMSR_set_display_default**
- setting the environment variables **CM_DISPLAY**
- choosing a display from the menu displayed by **CMSR_create_display_menu**

CMSR_create_display_menu first attempts to create the display by checking the default display variable set with **CMSR_set_display_default**. If no default is set, it checks the **CM_DISPLAY** environment variable. Finally, if neither of these is set, it presents the user with a menu of the available displays.

If the display named in a variable or selected from the menu is an X11 server, **CMSR_create_display_menu** initializes it as closely as possible to the *window_type*, *desired_width*, and *desired_height* specified.

If the display named in a variable or selected from the menu is a CM framebuffer, **CMSR_create_display_menu** initializes the framebuffer as a display by setting the color mode as closely as possible to *window_type* and by setting the display height and width to the height and width of the framebuffer's attached monitor. For framebuffers, *desired_width*, and *desired_height* are not used; the display space is always the entire monitor screen.

For more details on the use of this menu, see Chapter 1, "Introduction to the Generic Display Interface."

The string used as the display default or as the **CM_DISPLAY** environment variable can be any of the following:

- the name of an X11 server (such as **LEANDER:0**)
- the string **CMFB** to identify the default framebuffer
- the string **CMFB8** to identify the default framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24** to identify the default framebuffer and set it to 24-bits-per-pixel mode
- the string **CMFB:** followed immediately by the location string to identify a particular framebuffer display
- the string **CMFB8:** followed immediately by the location string to identify a particular framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24:** followed immediately by the location string to identify a particular framebuffer and set it to 24-bits-per-pixel mode.

CMSR_set_display_default
CMSR_get_display_default

Sets (returns) the default display to be created by the Generic Display Interface.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_display_default (string)
char *string;

char *
  CMSR_get_display_default ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_DISPLAY_DEFAULT (string)
CHARACTER*(*) string

CHARACTER*(*) FUNCTION CMSR_GET_DISPLAY_DEFAULT ()
```

Lisp Syntax

```
CMSR:set-display-default (string)
CMSR:get-display-default ()
```

ARGUMENTS

string — A character string specifying the default Generic Display to be created. This string can be any of the following:

- the name of an X11 server (such as **LEANDER:0**)

- the string **CMFB** to identify the default framebuffer
- the string **CMFB8** to identify the default framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24** to identify the default framebuffer and set it to 24-bits-per-pixel mode
- the string **CMFB:** followed immediately by a location string to identify a particular framebuffer display
- the string **CMFB8:** followed immediately by the location string to identify a particular framebuffer and set it to 8-bits-per-pixel mode
- the string **CMFB24:** followed immediately by the location string to identify a particular framebuffer and set it to 24-bits-per-pixel mode

DESCRIPTION

CMSR_set_display_default sets the Generic Display default display variable to *string*. If set, this variable is used by the following routines to create a Generic Display:

- **CMSR_select_display_menu**
- **CMSR_create_display_menu**
- **CMSR_select_workstation_menu**
- **CMSR_create_workstation_menu**

When called, these routines first attempt to create the display by checking the default display variable set with **CMSR_set_display_default**. If no default is set, they check the **CM_DISPLAY** environment variable. Finally, if neither of these is set, they present a menu of the available displays.

CMSR_get_display_default returns the string to which the Generic Display default variable is currently set.

CMSR_select_display
CMSR_deselect_display

Selects (deselects) the specified display as the current Generic Display Interface display.

SYNTAX

C Syntax

```
#include <cm/display.h>
void
  CMSR_select_display (display)
CMSR_display_t display;

#include <cm/display.h>
void
  CMSR_deselect_display ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_SELECT_DISPLAY (display)
INTEGER  display
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_DESELECT_DISPLAY ()
```

Lisp Syntax

```
CMSR:select-display (display)
CMSR:deselect-display ()
```

ARGUMENTS

display	The CM generic display interface structure that is to be selected. The display specified must have been previously created and initialized.

DESCRIPTION

CMSR_select_display makes the specified display the currently selected display. If another display is currently selected when **CMSR_select_display** is called, that display is automatically deselected.

The currently selected display is the display operated on by the other CMSR display operations including the Display I/O Operations and the Information Operations.

CMSR_deselect_display deselects the currently selected display. No display I/O routine may be performed until another display is selected. If no display is selected, this routine does nothing.

ERRORS

An error is signaled if *display* has not been allocated or is NULL.

CMSR_selected_display

Returns the Generic Display structure of the currently selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_display_t
  CMSR_selected_display ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_SELECTED_DISPLAY ()
```

Lisp Syntax

```
CMSR:selected-display ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_selected_display returns the generic display structure corresponding to the selected display. This returns NULL if no display is selected.

CMSR_deallocate_display

Deallocates the specified display and frees associated resources.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_deallocate_display (display)

CMSR_display_t display;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_DEALLOCATE_DISPLAY (display)

INTEGER  display
```

Lisp Syntax

```
CMSR:deallocate-display (&optional display)
```

ARGUMENTS

display	The CM generic display interface structure that is to be deallocated.

DESCRIPTION

CMSR_deallocate_display deallocates the CM generic display structure specified by *display* and frees the resources associated with it. If the specified display was also the currently selected display, it is first deselected.

If *display* is a framebuffer interface, the CM display is detached. If the *display* is an X window interface, the window is destroyed. If *display* is passed in as NULL or omitted in Lisp, the current display is deallocated.

The CM generic display structure may be created by **CMSR_create_display_menu**, **CMSR_create_init_cmfb_display**, or **CMSR_create_cmfb_display**.

CMSR_set_display_offset

Sets the upper left location on the selected display at which to begin image transfers.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
   CMSR_set_display_offset (x_offset, y_offset)

int    x_offset, y_offset;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_DISPLAY_OFFSET (x_offset, y_offset)

INTEGER  x_offset, y_offset
```

Lisp Syntax

```
CMSR:set-display-offset (x-offset y-offset)
```

ARGUMENTS

x_offset, y_offset	The starting offset, in pixels, from the upper left corner of the display window.

DESCRIPTION

CMSR_set_display_offset sets the offsets in the *x* and *y* dimensions of the display window to be used for image transfers. The image is offset *x_offset* pixels in the *x* dimension and *y_offset* pixels in the *y* dimension from the upper left corner of the display window.

The *x* (horizontal) dimension of the display corresponds to axis 0 of the image buffer in CM memory. The *y* (vertical) dimension of the display corresponds to axis 1 of the image buffer. The origin (0,0) of the display is at the upper left corner. *x* values increase to the right, and *y* values increase toward the bottom of the display. When an offset is applied to the image data, the pixel value at location (*x*, *y*) in the image data field is transferred to location (*x* + *x_offset*, *y* + *y_offset*) in the display window.

SEE ALSO

CMSR_display_x_offset

CMSR_display_y_offset

2.4 Display I/O Routines

This section describes the Generic Display routines that transfer image data to or from the currently selected generic display:

CMSR_write_to_display

Writes the image data in the specified CM field to the currently selected Generic Display Interface display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_write_to_display (field)

CM_field_id_t field;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_WRITE_TO_DISPLAY (field)

INTEGER field
```

Lisp Syntax

```
CMSR:write-to-display (field)
```

ARGUMENTS

field	The CM Paris field to be transferred to the current display.

DESCRIPTION

CMSR_write_to_display writes *field* from CM memory to the currently selected display.

The geometry in which *field* has been defined must be two-dimensional. Axis 0 of the geometry maps to the screen's *x* (horizontal) axis and axis 1 of the geometry maps to the screen's *y* (vertical) axis. The pixel value stored at NEWS location (0,0) in the field

is displayed in the upper left corner of the screen, offset by the generic display's *x* and *y* offset values, if any. The *x* coordinate values increase to the right and the *y* coordinate values increase towards the bottom of the display.

Each virtual processor in the *field* VP set, both active and inactive, draws a single pixel on the screen in raster order; thus, the total number of values (pixels) transferred is equal to the total number of virtual processors in the VP set. If the field is larger than the display, the portion to the right and bottom of the display is clipped.

The length of the field must be at least that which **CMSR_display_bits_per_pixel** returns. If the length of the field is shorter than the bits-per-pixel supported by the display, an error is signaled. If the length of the field is longer than the bits-per-pixel, the field's high-order bits are not used.

ERRORS

An error is signaled if the length of the field to be transferred is shorter than the bits-per-pixel supported by the display.

SEE ALSO

CMSR_write_to_display_1

CMSR_set_display_offset

CMSR_write_to_display_1

Writes the image data from the specified portion of the specified field to the current display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_write_to_display_1
        (field, field_x_offset, field_y_offset, field_width, field_height)
CM_field_id_t  field;
int            field_x_offset;
int            field_y_offset;
int            field_width;
int            field_height;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_WRITE_TO_DISPLAY_1
&                    (field field_x_offset field_y_offset field_width
&                     field_height)
INTEGER  field
INTEGER  field_x_offset
INTEGER  field_y_offset
INTEGER  field_width
INTEGER  field_height
```

Lisp Syntax

```
CMSR:write-to-display-1  (field field-x-offset field-y-offset
                          field-width field-height)
```

ARGUMENTS

field	The CM Paris field to be transferred to the current display.
field_x_offset	The starting virtual processor along axis 0 within *field* from which to begin transferring the field. Axis 0 of the field geometry corresponds to the screen's *x* (horizontal) dimension. The offset is the number of processor positions from location (0,0).
field_y_offset	The starting virtual processor along axis 1 within *field* from which to begin transferring the field. Axis 1 of the field geometry corresponds to the screen's *y* (vertical) dimension. The offset is the number of processor positions from location (0,0).
field_width	The number of virtual processors in the horizontal axis of the field (axis 0) to be transferred.
field_height	The number of virtual processors in the vertical axis of the field (axis 1) to be transferred.

DESCRIPTION

`CMSR_write_to_display_1` writes a specified subarray of the field to a specified display location.

The geometry in which *field* has been defined must be two-dimensional. Axis 0 of the geometry maps to the screen's *x* (horizontal) axis and axis 1 of the geometry maps to the screen's *y* (vertical) axis. The origin (0,0) of the screen display is at the upper left corner. The *x* coordinate values increase to the right, and the *y* coordinate values increase towards the bottom of the display.

The upper left corner of the subarray of the *field* VP set to be transferred is specified by the NEWS location (*field_x_offset, field_y_offset*). The lower right corner of the subarray is ((*field_x_offset* + *field–width* – 1), (*field_y_offset* + *field–height* – *1*)).

The pixel value stored at (*field_x_offset, field_y_offset*) in the field is displayed at the display pixel located at the upper left corner of the screen, offset by the Generic Display's *x* and *y* offset values, if any. (The screen offset values are set with **`CMSR_set_display_offset`**.) Each virtual processor in the *field* VP set subarray draws a single pixel on the screen in raster order. If the field is larger than the display, the portion to the right and bottom is clipped.

The length of the field must be at least that which **`CMSR_display_bits_per_pixel`** returns. If the length of the field is shorter than the bits-per-pixel supported by the

display, an error is signaled. If the length of the field is longer than the bits-per-pixel, the field's high-order bits are not used.

ERRORS

An error is signaled if the length of the field to be transferred is shorter than the bits-per-pixel supported by the display.

SEE ALSO

CMSR_write_to_display

CMSR_set_display_offset

CMSR_write_array_to_display

Copies a front-end array to the currently selected generic display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_write_array_to_display (array, array_width, array_height)

char *array
int   array_width, array_height
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_WRITE_ARRAY_TO_DISPLAY
&                                    (array, array_width, array_height,)

CHAR*(*) array
INTEGER  array_width
INTEGER  array_height
```

Lisp Syntax

```
CMSR:write-array-to-display (array)
```

ARGUMENTS

array	A 2D array on the front-end computer to be copied to the display.
array_width	The number of elements along the faster-varying dimension of the front-end array. For Fortran this is the first index; for C this is the second index. This is the axis that is mapped to the x, or horizontal, axis on the display.

array_height The number of elements along the slower varying dimension of the front-end array. For Fortran this is the second index; for C this is the first index. This axis is mapped to the *y*, or vertical, axis on the display.

DESCRIPTION

CMSR_write_array_to_display copies the front-end array, *array,* to the current Generic Display.

The array must be a 2D array but can be any front-end data type that provides an appropriate number of bits for the depth of the display. It is assumed that the number of bits in each element of the array is the next power of two higher than the number of bits-per-pixel in the display. If the length of an array element is not equal to the next power of two higher than the display's bits-per-pixel, the results are unpredictable.

Beginning at the first element of the array, an *array_width* by *array_height* rectangle of color values is rendered directly onto the display, overwriting whatever pixel values are being displayed. The transfer begins in the display at the point defined by the Generic Display offsets.

If the array is larger than the display space, the portion of the array beyond the display boundaries to the right and bottom are clipped.

It the array is smaller than the display space, the portion of the display beyond the array width and height is left unchanged.

SEE ALSO

CMSR_write_array_to_display_1

CMSR_write_array_to_display_1

Copies a specified portion of a front-end array to the current Generic Display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_write_array_to_display_1
        (array, array_width, array_height, array_element_size,
         xoffset, yoffset, width, height, x_varies_fastest_p, combiner)
char *array;
int   array_width, array_height;
int   array_element_size;
int   xoffset, yoffset;
int   width, height;
int   x_varies_fastest_p;
CMSR_combiner_t   combiner;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_WRITE_ARRAY_TO_DISPLAY_1
&     (array, array_width, array_height, array_element_size,
&      xoffset, yoffset, width, height, x_varies_fastest_p, combiner)

CHARACTER*(*)  array;
INTEGER        array_width, array_height;
INTEGER        array_element_size;
INTEGER        xoffset, yoffset;
INTEGER        width, height;
INTEGER        x_varies_fastest_p;
INTEGER        combiner;
```

Lisp Syntax

CMSR:write-array-to-display-1 (*array*, **&key** *array-element-size xoffset*, *yoffset*, *width*, *height*, (*x_varies-fastest-p* **t**), *combiner*)

ARGUMENTS

array — A 2D array on the front-end computer to be copied to the display.

array_width — The number of elements along the faster-varying dimension of the front-end array. For Fortran this is the first index; for C this is the second index.

array_height — The number of elements along the slower-varying dimension of the front-end array. For Fortran this is the second index; for C this is the first index.

array_element_size — The length, in bits, of an array element in *array*. In Lisp, this defaults to the actual size of an array element.

xoffset, yoffset — The location in *array* at which to begin copying data. The *xoffset* is the number of elements along the width (i.e., the faster-varying) dimension of the array. The *yoffset* is the number of elements along the height (i.e., the slower-varying) dimension. In Lisp, this defaults to (0,0).

width — The number of array elements to be transferred along the horizontal (i.e., the faster-varying) dimension of the array. In Lisp, this defaults to *array_width*.

height — The number of array elements to be transferred along the vertical (i.e., the slower-varying) dimension of the array. In Lisp, this defaults to *array_height*.

x_varies_fastest_p — If *x_varies_fastest_p* is true (.TRUE. in Fortran, non-NULL in C, non-nil in Lisp), the front-end array is mapped directly onto the display, aligning the faster-varying axis of the array to the horizontal axis of the display. This produces the correct results for Fortran arrays and for C arrays that are referenced [*y*][*x*].

If *x_varies_fastest_p* is false (.FALSE. in Fortran, NULL in C, nil in Lisp), the front-end array is transposed as it is transferred to the display; the faster- varying axis of the array is mapped onto the

vertical axis of the display. This produces correct results for C arrays that are referenced [*x*][*y*].

combiner A symbol defining the method used to combine the color values being written from the array into the display with the values already in the display. Valid values are listed in the table below.

C Values	Fortran Values	Lisp Keywords
`CMSR_default`	`CMSR_default`	`:DEFAULT`
`CMSR_overwrite`	`CMSR_overwrite`	`:OVERWRITE`
`CMSR_logior`	`CMSR_logior`	`:LOGIOR`
`CMSR_logand`	`CMSR_logand`	`:LOGAND`
`CMSR_logxor`	`CMSR_logxor`	`:LOGXOR`
`CMSR_u_add`	`CMSR_u_add`	`:U-ADD`
`CMSR_s_add`	`CMSR_s_add`	`:S-ADD`
`CMSR_u_min`	`CMSR_u_min`	`:U-MIN`
`CMSR_s_min`	`CMSR_s_min`	`:S-MIN`
`CMSR_u_max`	`CMSR_u_max`	`:U-MAX`
`CMSR_s_max`	`CMSR_s_max`	`:S-MAX`

DESCRIPTION

`CMSR_write_array_to_display_1` copies a specified subarray of *array* from the front end to the current Generic Display. The array must be a 2D array, but it can be any front-end data type that provides an appropriate number of bits for the depth of the display.

The three parameters *array_width, array_height*, and *array_element_size* define *array.* The arguments *array_width* and *array_height* are the total number of elements in each dimension of the array. The *array_element_size* argument specifies the length in bits of each element of the array. If the length of the array element is smaller than the number of bits-per-pixel, an error is signaled. If the array element is larger than the bits-per-pixel, only the lower-order bits of the array element, up to the depth of the display, are used.

The arguments *xoffset*, *yoffset*, *width*, and *height* define the subarray within *array* that is to be transferred. *xoffset* and *yoffset* define the location in *array* at which the transfer should begin, and *width* and *height* are the number of array elements to be transferred

in each direction. So, the portion of *array* to be transferred is the subarray from (*xoffset, yoffset*) at the upper left corner to ((*xoffset* + *width*), (*yoffset* + *height*)) at the lower right corner. Each element of the subarray is transferred to the corresponding pixel of the display beginning at the point defined by any Generic Display offsets that may be set for the current display.

Each array element value is combined with the pixel value previously stored at the corresponding display location according to the value of *combiner*; the default value is to overwrite. Valid values for this parameter are:

- **DEFAULT** Overwrite
- **OVERWRITE** Replace existing pixel value with incoming value from array.
- **LOGIOR** Combine using bitwise logical inclusive OR.
- **LOGAND** Combine using bitwise logical AND.
- **LOGXOR** Combine using bitwise logical exclusive OR.
- **U-ADD** Combine using unsigned integer addition.
- **S-ADD** Combine using signed integer addition.
- **U-MIN** Save smaller of the values using unsigned integer minimum.
- **S-MIN** Save smaller of the values using signed integer minimum.
- **U-MAX** Save larger of the values using unsigned integer maximum.
- **S-MAX** Save larger of the values using signed integer maximum.

If the subarray is larger than the display space, the portion of the array beyond the display boundaries to the right and bottom are clipped.

It the subarray is smaller than the display space, the portion of the display space beyond the array width and height, to the right and bottom, is left unchanged.

SEE ALSO

CMSR_write_array_to_display

CMSR_clear_display

Clears the selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>
void
  CMSR_clear_display ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_CLEAR_DISPLAY ()
```

Lisp Syntax

```
CMSR:clear-display ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_clear_display clears the currently selected display.

If the selected display is an X window, the display is filled with the background color.

If the selected display is a CM framebuffer or a Symbolics display, the framebuffer memory is filled with zeros.

SEE ALSO

CMSR_randomize_display

CMSR_randomize_display

Fills the current display with random data.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_randomize_display ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_RANDOMIZE_DISPLAY ()
```

Lisp Syntax

```
CMSR:randomize-display ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_randomize_display fills the entire display with random bits. This is useful for determining whether the display is responsive.

SEE ALSO

CMSR_clear_display.

CMSR_fill_display

Fills the display with the value given in *value.*

SYNTAX

C Syntax

```
#include <cm/display.h>
void
  CMSR_fill_display (value)
int value;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_FILL_DISPLAY (value)
INTEGER value
```

Lisp Syntax

```
CMSR:fill-display (value)
```

ARGUMENTS

value	The color value with which to fill the display. This value is an index into the color map of the currently selected display.

DESCRIPTION

CMSR_fill_display fills the entire display with the value given in *value.*

SEE ALSO

CMSR_clear_display

CMSR_fe_display_rectangle

Fills a rectangle on the display with the specified color.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_fe_display_rectangle (x, y, width, height, color)
int x;
int y;
int width;
int height;
unsigned int color;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_FE_DISPLAY_RECTANGLE (x, y, width, height, color)
INTEGER x;
INTEGER y;
INTEGER width;
INTEGER height;
INTEGER color;
```

Lisp Syntax

```
CMSR:fe-display-rectangle (x, y, width, height, color)
```

ARGUMENTS

x, y	The position on the display at which to begin drawing the rectangle. The position is measured in pixels from the upper left corner of the display. *x* is the horizontal distance to the right. *y* is the vertical distance down.

width, height The dimensions in pixels of the rectangle to be drawn. *width* is the horizontal distance of the rectangle from (x, y). *height* is the vertical distance of the rectangle from (x, y).

color An integer specifying the color in which the rectangle is to be drawn. *color* is an index into the color map of the currently selected Generic Display.

DESCRIPTION

CMSR_fe_display_rectangle draws a filled rectangle of the specified width and height at the specified (x, y) position in the specified color. The rectangle fills the display from (x, y) at the upper left corner to ((x + *width*), (y + *height*) at the lower right.

The rectangle is drawn directly on the display surface, *not* into a field. Like **CMSR_display_text**, any writing to the display (as with **CMSR_write_to_display**) overwrites the rectangle.

SEE ALSO

CMSR_fe_draw_rectangle

CMSR_clear_display

CMSR_read_from_display

Reads the image data from the current display into the specified field of CM memory.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_read_from_display (field)

CM_field_id_t field;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_READ_FROM_DISPLAY (field)

INTEGER field
```

Lisp Syntax

```
CMSR:read-from-display (field)
```

ARGUMENTS

field	The Paris field in CM memory to which the display image data is to be sent.

DESCRIPTION

CMSR_read_from_display reads the image data from the current display into the specified field of CM memory.

The geometry in which *field* has been defined must be two-dimensional. The *x* (horizontal) dimension of the display maps to axis 0 of the field geometry, and the *y* (vertical) dimension of the display maps to axis 1 of the geometry. The *x* coordinate

values increase to the right and the *y* coordinate values increase towards the bottom of the display. The length of the field must be at least that which **CMSR_display_bits_per_pixel** returns.

A pixel value is read from the display for each virtual processor in the field VP set. If the field dimensions are larger than the display's, the portion of the field outside the display boundaries is left unchanged.

SEE ALSO

CMSR_read_from_display_1

CMSR_read_from_display_1

Reads the image data from the current display, beginning at the specified offset to the specified location in the CM field.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_read_from_display_1
            (field, field_x_offset, field_y_offset, field_width, field_height)

CM_field_id_t  field;
int            field_x_offset;
int            field_y_offset;
int            field_width;
int            field_height;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_READ_FROM_DISPLAY_1
&             (field, field_x_offset, field_y_offset, field_width, field_height)

INTEGER       field
INTEGER       field_x_offset
INTEGER       field_y_offset
INTEGER       field_width
INTEGER       field_height
```

Lisp Syntax

```
CMSR:read-from-display-1    (field field-x-offset field-y-offset
                             field-width field-height)
```

ARGUMENTS

field	The Paris field in CM memory to which the display image data is to be sent. *field* must be in the current VP set.
field_x_offset	The starting virtual processor along axis 0 of the field at which to begin loading the image data. Axis 0 of the field geometry corresponds to the screen's *x* (horizontal) dimension. The offset is the number of processor positions to the right from location (0,0).
field_y_offset	The starting virtual processor along axis 1 of the field at which to begin loading the image data. Axis 1 of the field geometry corresponds to the screen's *y* (vertical) dimension. The offset is the number of processor positions down from location (0,0).
field_width	The number of virtual processors in the horizontal axis of the field (axis 0) to which image data is to be transferred.
field_height	The number of virtual processors in the vertical axis of the field (axis 1) to which image data is to be transferred.

DESCRIPTION

`CMSR_read_from_display_1` reads a portion of the display image data into a specified subarray of field in CM memory.

The geometry in which *field* has been defined must be two-dimensional. The *x* (horizontal) dimension of the display maps to axis 0 of the field geometry, and the *y* (vertical) dimension of the display maps to axis 1 of the geometry. The *x* coordinate values increase to the right, and the *y* coordinate values increase towards the bottom of the display. The length of the field must be at least that which **`CMSR_display_bits_per_pixel`** returns.

Beginning at the location specified by the Generic Display Interface *x* and *y* offsets, if any, a pixel value is read from the display in raster order for each virtual processor in the specified subarray of the field VP set. The upper left corner of the subarray of the *field* VP set is specified by the NEWS location (*field_x_offset, field_y_offset*). The lower right corner of the subarray is ((*field_x_offset* + *field_width* – 1), (*field_y_offset* + *field_height* –1)). If the field is not as large as the display, the portion of the field outside the display boundaries is left unchanged.

SEE ALSO

CMSR_read_from_display

CMSR_set_display_offset

CMSR_read_array_from_display

Reads image data from the current display into a front-end array.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_read_array_from_display (array, array_width, array_height)

char *array
int   array_width, array_height
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_READ_ARRAY_FROM_DISPLAY
&                                    array, array_width, array_height,)

CHAR*(*) array
INTEGER  array_width, array_height
```

Lisp Syntax

```
CMSR:read-array-from-display(array)
```

ARGUMENTS

array A 2D front-end array into which the data from the display is to be read.

array_width, array_height
The dimensions of the *array*. The *array_width* must end up on a byte boundary, that is,
(*array_width* * *array_element_size*) % 8 = 0.

DESCRIPTION

`CMSR_read_array_from_display` reads pixel values from the currently selected generic display into the corresponding elements of *array* on the front-end computer. The read begins at the upper left corner of the display defined by any Generic Display offsets that might be set. If the array is not large enough to hold the entire display, the portions of the display image beyond the array dimensions to the right and bottom ($+x$, $+y$) are clipped.

The array must be a 2D array but can be any front-end data type that provides an appropriate number of bits for the depth of the display. It is assumed that the number of bits in each element of the array is the next power of two higher than the number of bits-per-pixel in the display. If the length of an array element is not equal to the next power of two higher than the display's bits-per-pixel, the results are unpredictable.

CMSR_read_array_from_display_1

Reads image data from a subrectangle of the current display into a front-end array.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_read_array_from_display_1
      (array, array_width, array_height, array_element_size,
       xoffset, yoffset, width, height, x_varies_fastest_p, combiner)

char *array;
unsigned int   array_width, array_height;
unsigned int   array_element_size
int    xoffset, yoffset;
int    width, height;
int    x_varies_fastest_p;
CMSR_combiner_t    combiner;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_READ_ARRAY_FROM_DISPLAY_1
&       (array, array_width, array_height, array_element_size,
&        xoffset, yoffset, width, height, x_varies_fastest_p, combiner)

CHAR*(*) array
INTEGER  array_width, array_height
INTEGER  array_element_size
INTEGER  xoffset, yoffset
INTEGER  width, height
LOGICAL  x_varies_fastest_p
INTEGER  combiner
```

Lisp Syntax

```
CMSR:read-array-from-display_1
                    (array, array-width, array-height,
                     &key array-element-size, xoffset, yoffset,
                          width, height, x-varies-fastest-p, combiner)
```

ARGUMENTS

array A 2D front-end array into which the data from the display is to be read.

array_width, array_height
The dimensions of the *array*. The *array_width* must end up on a byte boundary, that is, (*array_width* * *array_element_size*) % 8 = 0.

array_element_size The depth of the front-end array elements in bits. This value must be a power of two between 1 and 128.

xoffset, yoffset The offset into the array at which to begin writing the data from the display.

width, height The dimensions of the subrectangle in the display that is to be transferred.

x_varies_fastest_p Indicates whether the first or second array index varies fastest in *array*.

If *x_varies_fastest_p* is true (.TRUE. in Fortran, non-NULL in C, non-nil in Lisp), the horizontal (*x*) axis of the display is mapped directly to the faster-varying axis of the array. This produces the correct results for Fortran arrays and for C arrays that are referenced [*y*][*x*].

If *x_varies_fastest_p* is false (.FALSE. in Fortran, NULL in C, nil in Lisp), the display data is transposed as it is transferred into the array; the vertical (*y*) axis of the display is mapped to the faster-varying axis of the array. This produces correct results for C arrays that are referenced [*x*][*y*].

combiner A symbol defining the method used to combine the color values being written from the display into the array with the values already in the array. Valid values are listed in the table below.

C Values	Fortran Values	Lisp Keywords
`CMSR_default`	`CMSR_default`	`:DEFAULT`
`CMSR_overwrite`	`CMSR_overwrite`	`:OVERWRITE`
`CMSR_logior`	`CMSR_logior`	`:LOGIOR`
`CMSR_logand`	`CMSR_logand`	`:LOGAND`
`CMSR_logxor`	`CMSR_logxor`	`:LOGXOR`
`CMSR_u_add`	`CMSR_u_add`	`:U-ADD`
`CMSR_s_add`	`CMSR_s_add`	`:S-ADD`
`CMSR_u_min`	`CMSR_u_min`	`:U-MIN`
`CMSR_s_min`	`CMSR_s_min`	`:S-MIN`
`CMSR_u_max`	`CMSR_u_max`	`:U-MAX`
`CMSR_s_max`	`CMSR_s_max`	`:S-MAX`

DESCRIPTION

`CMSR_read_array_from_display_1` copies a subarray of the current generic display to *array* on the front-end computer. The array must be a 2D array, but it can be any front-end data type that provides an appropriate number of bits for the depth of the display.

The three parameters *array_width, array_height*, and *array_element_size* define *array.* The arguments *array_width* and *array_height* are the total number of elements in each dimension of the array. The *array_element_size* argument specifies the length in bits of each element of the array. If the number of bits-per-pixel is smaller than the length of the array element, an error is signaled. If bits-per-pixel is larger than the array element, only the lower-order bits of the array element, up to *array_element_size*, are used.

The arguments *xoffset* and *yoffset* specify the location in the *array* at which to begin reading in the data from the display. The subarray of the display to be read is defined by any Generic Display offsets and the arguments *width* and *height*. So, the portion of the display to be read is from (*generic_display_x_offset, generic_display_y_offset*) at the upper left corner to ((*generic_display_x_offset* + *width*), (*generic_display_y_offset* + *height*)) at the lower right. If the array is not large enough to hold the entire display subarray, the portions of the display image beyond the array dimensions to the right and bottom (+*x*, +*y*) are clipped.

2.5 Color Map Utilities

This section describes the Generic Display routines that write to, read from, or return information about the generic display's color map:

CMSR_display_write_color

Writes a single color entry into the current display's color map.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_display_write_color (index, red, green, blue)

int      index;
double   red, green, blue;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_DISPLAY_WRITE_COLOR (index, red, green, blue)

INTEGER [index
DOUBLE PRECISION red, green, blue
```

Lisp Syntax

```
CMSR:display-write-color (index, red, green, blue)
```

ARGUMENTS

index	The entry in the color map to which the *red*, *green*, and *blue* color values are to be written.
red, green, blue	The color values to be written to the corresponding color buffer of the color map. Values may range from 0.0 (off) to 1.0 (full intensity).

DESCRIPTION

CMSR_display_write_color writes a color value to the entry in the color map of the currently selected display specified by *index*. The color values *red, green,* and *blue* are written to the corresponding color buffer at the specified entry.

ERRORS

An error is signaled if the currently selected display doesn't support color maps. (**CMSR_display_has_color_map** returns NULL if the currently selected display does not support color maps.)

SEE ALSO

CMSR_display_write_color_map

CMSR_set_display_color_map

CMSR_set_direct_color_default

CMSR_set_pseudo_color_default

CMSR_display_write_color_map

Writes a sequence of color values into entries on the color map of the currently selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_display_write_color_map
                              (red_map, green_map, blue_map, size)

float *red_map, *green_map, *blue_map;
int   size;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_DISPLAY_WRITE_COLOR_MAP
&                             (red_map green_map blue_map size)

REAL   red_map(*), green_map(*), blue_map(*)
INTEGER  size
```

Lisp Syntax

```
CMSR:display-write-color-map  (red-map  green-map  blue-map
                               &optional  size)
```

ARGUMENTS

red_map, green_map, blue_map

Arrays containing the color values to be installed in the currently selected display's color map. The entries in the arrays must be in the range 0.0 (off) to 1.0 (full intensity).

size The number of entries in the arrays.

In Lisp only, *size* defaults to the size of the first array.

DESCRIPTION

CMSR_display_write_color_map writes the color values specified in *red_map green_map*, and *blue_map* into the corresponding color buffer of the color map for the currently selected display.

ERRORS

An error is signaled if

- *size* is longer than the size of the display's color map (**CMSR_display_color_map_size** returns the size of the currently selected display)
- the display does not support color maps (**CMSR_display_has_color_map** returns NULL if the currently selected display does not support color maps)

SEE ALSO

CMSR_display_write_color

CMSR_set_display_color_map

CMSR_set_direct_color_default

CMSR_set_pseudo_color_default

CMSR_set_display_color_map

Writes a standard color map (density, grayscale, random, or rainbow) to the currently selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_display_color_map (color_map_name)

char  *color_map_name;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_DISPLAY_COLOR_MAP (color_map_name)

CHARACTER*(*)      color_map_name(*)
```

Lisp Syntax

```
CMSR:set-display-color-map (color-map-name)
```

ARGUMENTS

color_map_name The name of the standard color map to be installed. Valid values are:

- density
- greyscale or grayscale
- random
- rainbow

Capitalization in the color map name is ignored.

DESCRIPTION

CMSR_set_display_color_map installs a standard color map in the currently selected display. This routine is not ordinarily useful if colors are specified as RGB values.

Density is the default map. It is a scale running from dark blue to cyan to yellow to red over the length of the color map.

Greyscale or *grayscale* loads the color map with a linear sequence of gray shades from black in the first index, to white at the last.

Random randomizes the entire red, green, and blue color maps.

Rainbow puts a sinusoidal distribution of colors in the color table. The first index is set to black. The remaining positions on the color table range smoothly from red to green to blue and back to red. The intensity is constant throughout, only the hue changes.

This routine does nothing if the selected display does not have a color map.

SEE ALSO

CMSR_set_direct_color_default

CMSR_set_pseudo_color_default

CMSR_display_write_color

CMSR_display_write_color_map

CMSR_create_color_map_named

Loads color map arrays with standard Generic Display color maps.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_create_color_map_named
    (name, red_array, green_array, blue_array, red_size, green_size, blue_size)

char  *name;
float *red_array, *green_array, *blue_array;
int   red_size, green_size, blue_size;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_CREATE_COLOR_MAP_NAMED
&     (name, red_array, green_array, blue_array, red_size, green_size, blue_size)

CHARACTER*(*)  name
REAL*(*)       red_array, green_array, blue_array
INTEGER        red_size, green_size, blue_size
```

Lisp Syntax

```
CMSR:create-color-map-named
                          (name red-array green-array blue-array
                           &optional red-size green-size blue-size)
```

ARGUMENTS

name — The name of one of the standard Generic Display color maps. Valid values are:

- density
- greyscale or grayscale
- random
- rainbow

The names are not case-sensitive, that is, capitalization is ignored.

red_array, green_array, blue_array
Arrays of single floats of length *red_size, green_size,* or *blue_size,* respectively. (Values can be loaded into the arrays with **CMSR_create_color_map_named**.

red_size, green_size, blue_size
The respective length of *red_array, green_array,* and *blue_array.*

DESCRIPTION

CMSR_create_color_map_named fills in the user-supplied arrays with the standard Generic Display color map named by *name*:

- *Density* is the default map. It is a scale running from dark blue to cyan to yellow to red over the length of the color map.
- *Greyscale* or *grayscale* loads the color map with a linear sequence of gray shades from black in the first index to white at the last.
- *Random* randomizes the entire red, green, and blue color maps.
- *Rainbow* puts a sinusoidal distribution of colors in the color table. The first index is set to black. The remaining positions on the color table range smoothly from red to green to blue and back to red. The intensity is constant throughout; only the hue changes.

The color values in the arrays are single floats between 0.0 (off) and 1.0 (full intensity). The arrays may be different sizes (for example, in the case of an 8-bit true-color display with 2 bits of red, 3 bits of green, and 3 bits of blue).

The filled arrays can be written to a display color map, for example, with **CMSR_display_write_color_map**.

CMSR_display_read_color_map

Returns the color values of the current display's color map.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_display_read_color_map
                              (red_map, green_map, blue_map, size)

float *red_map, *green_map, *blue_map;
int   size;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_DISPLAY_READ_COLOR_MAP
&                             (red_map green_map blue_map size)

REAL    red_map(*), green_map(*), blue_map(*)
INTEGER size
```

Lisp Syntax

```
CMSR:display-read-color-map  (red-map green-map  blue-map
                              &optional size)
```

ARGUMENTS

red_map, green_map, blue_map
: These arguments return the red, green, and blue color values, respectively, for each entry in the color map.

size
: The number of entries in the arrays. In Lisp only, *size* defaults to the length of the first array.

DESCRIPTION

CMSR_display_read_color_map reads the color components for each entry of the color map of the currently selected display into the arrays *red_map, green_map,* and *blue_map.* The color values may range from 0.0 (off) to 1.0 (full intensity). The *size* argument specifies the length of the arrays.

ERRORS

An error is signaled if

- *size* is longer than the length of the color map (**CMSR_display_color_map_size** returns the size of the color map for the currently selected display)
- the currently selected display does not support color maps

SEE ALSO

CMSR_display_read_color_blue

CMSR_display_read_color_green

CMSR_display_read_color_red

CMSR_get_direct_color_default

CMSR_get_pseudo_color_default

CMSR_display_read_color_red
CMSR_display_read_color_green
CMSR_display_read_color_blue

Returns the value of the named color component from a specified color map entry.

SYNTAX

C Syntax

```
#include <cm/display.h>
double
  CMSR_display_read_color_red (index)
double
  CMSR_display_read_color_green (index)
double
  CMSR_display_read_color_blue (index)
int   index;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
DOUBLE PRECISION FUNCTION CMSR_DISPLAY_READ_COLOR_RED (index)
DOUBLE PRECISION FUNCTION CMSR_DISPLAY_READ_COLOR_GREEN (index)
DOUBLE PRECISION FUNCTION CMSR_DISPLAY_READ_COLOR_BLUE (index)
INTEGER  index
```

Lisp Syntax

```
CMSR:display-read-color-red (index)
CMSR:display-read-color-green (index)
CMSR:display-read-color-blue (index)
```

ARGUMENTS

index — The entry in the color map of the currently selected display from which the color component is to be returned.

DESCRIPTION

CMSR_display_read_color_red, **CMSR_display_read_color_green**, and **CMSR_display_read_color_blue** return the value of the color component specified in their name from the color map entry specified by *index*. The value may range from 0.0 (off) to 1.0 (full intensity).

ERRORS

An error is signaled if the display does not support color maps. **CMSR_display_has_color_map** returns NULL if the current display does not support color maps.

SEE ALSO

CMSR_display_read_color_map

CMSR_set_direct_color_default
CMSR_get_direct_color_default

Sets (returns) the default standard color map for direct color displays.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_direct_color_default   (color_map_name)

char  *color_map_name;


char *
  CMSR_get_direct_color_default   ( )
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_DIRECT_COLOR_DEFAULT (color_map_name)

CHARACTER*(*)      color_map_name(*)


CHARACTER*(*) FUNCTION CMSR_GET_DIRECT_COLOR_DEFAULT ()
```

Lisp Syntax

```
CMSR:set-direct-color-default (color-map-name)

CMSR:get-direct-color-default ()
```

ARGUMENTS

color_map_name The name of the color map to be installed as the default for direct color displays. Valid values are:

- density
- greyscale or grayscale
- random
- rainbow

Capitalization of the color map name is ignored.

DESCRIPTION

CMSR_set_direct_color_default sets the default color map name for direct color displays (including X Window System DirectColor displays and 24-bit CM framebuffers).

NOTE: The direct color default is checked by the system only when the display is created. In order to set the direct color default, **CMSR_set_direct_color_default** must be called before the routine used to create the display, for example, **CMSR_select_display_menu**, **CMSR_select_workstation_menu**, or **CMSR_create_x_workstation**.

CMSR_get_direct_color_default returns the default color map name for direct color displays. In C and Lisp, **CMSR_get_direct_color_default** returns a pointer to a character string containing the default color map name. In Fortran, the return parameter *color_defaults* is used to return the color map name.

The default color map is used to initialize a display's color map when it is created. The initial direct color default is the grayscale color map. The default value may be changed with **CMSR_set_direct_color_default**.

Greyscale or *grayscale* loads the color map with a linear sequence of gray shades from black in the first index to white at the last.

Density is a scale from dark blue to cyan to yellow to red.

Random randomizes the entire red, green, and blue color maps.

Rainbow puts a sinusoidal distribution of colors in the color table. The first index is set to black. The remaining positions on the color table range smoothly from red to green to blue and back to red. The intensity is constant throughout, only the hue changes.

SEE ALSO

CMSR_set_pseudo_color_default

CMSR_get_pseudo_color_default

CMSR_set_pseudo_color_default
CMSR_get_pseudo_color_default

Sets (returns) the default color map for pseudo-color (8-bit) displays.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_pseudo_color_default (color_map_name)

char  *color_map_name;


char *
  CMSR_get_pseudo_color_default ( )
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_PSEUDO_COLOR_DEFAULT (color_map_name)

CHARACTER*(*)  color_map_name(*)


CMSR_GET_PSEUDO_COLOR_DEFAULT (color_map_name)

CHARACTER*(*)  color_map_name(*)
```

Lisp Syntax

```
CMSR:set-pseudo-color-default (color-map-name)

CMSR:get-pseudo-color-default ()
```

ARGUMENTS

color_map_name The name of the color map to be installed as the default for pseudo color displays. Valid values are:

- density
- greyscale or grayscale
- random
- rainbow

Capitalization of the color map name is ignored.

DESCRIPTION

CMSR_set_pseudo_color_default sets the default color map name for pseudo-color displays (including X Window System PseudoColor displays and 8-bit CM framebuffers).

NOTE: The pseudo color default is checked by the system only when the display is created. In order to set the direct color default, **CMSR_set_direct_color_default** must be called before the routine used to create the display, for example, **CMSR_select_display_menu**, **CMSR_select_workstation_menu**, or **CMSR_create_x_workstation**.

CMSR_get_pseudo_color_default returns the default color map name for pseudocolor displays (including X Window PseudoColor displays and 8-bit CM framebuffers). In C and Lisp, **CMSR_get_direct_color_default** returns a pointer to a character string containing the default color map name. In Fortran, the return parameter *color_defaults* is used to return the color map name.

The default color map is used to initialize a display's color map when it is created. The initial pseudo-color default is the *density* color map. The default value may be changed with **CMSR_set_pseudo_color_default**.

Density is a scale from dark blue to cyan to yellow to red.

Greyscale or *grayscale* loads the color map with a linear sequence of gray shades from black in the first index to white at the last.

Random randomizes the entire red, green, and blue color maps.

Rainbow puts a sinusoidal distribution of colors in the color table. The first index is set to black. The remaining positions on the color table range smoothly from red to green to blue and back to red. The intensity is constant throughout, only the hue changes.

SEE ALSO

CMSR_set_direct_color_default

CMSR_get_direct_color_default

CMSR_set_gray_scale_default
CMSR_get_gray_scale_default

Sets (returns) the default color map for grayscale displays.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_gray_scale_default (color_map_name)

char  *color_map_name;


char *
  CMSR_get_gray_scale_default ( )
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_GRAY_SCALE_DEFAULT (color_map_name)

CHARACTER*(*)  color_map_name(*)


CHARACTER*(*) FUNCTION CMSR_GET_GRAY_SCALE_DEFAULT ()
```

Lisp Syntax

```
CMSR:set-gray-scale-default (color-map-name)

CMSR:get-gray-scale-default ()
```

ARGUMENTS

color–map–name The name of the color map to be installed as the default for grayscale displays. Valid values are:

- greyscale or grayscale
- random

Capitalization of the color map name is ignored.

DESCRIPTION

CMSR_set_gray_scale_default sets the default color map name for grayscale displays (this includes X Window System GrayScale displays and 8-bit CM framebuffers using a grayscale color map).

CMSR_get_gray_scale_default returns the current default color map name for grayscale displays. The default color map is used to initialize a display's color map when it is created.

Greyscale or *grayscale* loads the color map with a linear sequence of gray shades from black in the first index to white at the last.

Random randomizes the entire color map.

SEE ALSO

CMSR_set_direct_color_default

CMSR_get_direct_color_default

CMSR_get_pseudo_color_default

CMSR_display_has_color_map

Returns true if the currently selected display has a writable color map.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_display_has_color_map ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

LOGICAL FUNCTION CMSR_DISPLAY_HAS_COLOR_MAP ()
```

Lisp Syntax

```
CMSR:display-has-color-map ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_display_has_color_map returns true (.TRUE. in Fortran, non-NULL in C, non-nil in Lisp) if the currently selected display has a writable color map.

For the following display types, **CMSR_display_has_color_map** returns:

- CM Framebuffer 8-bit True
- CM Framebuffer 24-bit True
- X11 PseudoColor True
- X 11DirectColor True
- X GrayScale True

- X StaticGray False
- X StaticColor False
- X TrueColor False

SEE ALSO

CMSR_display_color_map_size

CMSR_display_color_is_rgb

CMSR_get_direct_color_default

CMSR_get_pseudo_color_default

CMSR_display_color_map_size

Returns the number of entries in the color map for the current generic display.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_display_color_map_size ( )
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_DISPLAY_COLOR_MAP_SIZE ( )
```

Lisp Syntax

```
CMSR:display-color-map-size ( )
```

ARGUMENTS

None.

DESCRIPTION

CMSR_display_color_map_size returns the number of entries in the color map for the currently selected display.

This value is *not* the number of color components per entry, but the number of locations in the color map. For example, an 8-bit map ordinarily supports 256 entries.

SEE ALSO

CMSR_display_has_color_map

CMSR_display_color_is_rgb

CMSR_get_direct_color_default

CMSR_get_pseudo_color_default

CMSR_display_color_is_rgb

Returns true if the currently selected display's pixel values contain separate red, green, and blue components.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_display_color_is_rgb ( )
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

LOGICAL FUNCTION CMSR_DISPLAY_COLOR_IS_RGB ( )
```

Lisp Syntax

```
CMSR:display-color-is-rgb ( )
```

ARGUMENTS

None.

DESCRIPTION

CMSR_display_color_is_rgb returns true (.TRUE. in Fortran, non-NULL in C, non-nil in Lisp) if the pixel values of the currently selected display are decomposed into separate red, green, and blue components.

For the following display types, **CMSR_display_color_is_rgb** returns:

- CM Framebuffer 8-bit False
- CM Framebuffer 24-bit True
- X DirectColor True

- X TrueColor True
- X PseudoColor False
- X StaticColor False
- X GrayScale False
- X StaticGray False

SEE ALSO

CMSR_display_has_color_map

CMSR_display_color_map_size

CMSR_get_direct_color_default

CMSR_get_pseudo_color_default

2.6 Display Information Routines

This section describes the Generic Display routines that return information about the configuration of the current generic display:

CMSR_display_type

Returns the type of the currently selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>
CMSR_display_type_t
  CMSR_display_type ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_DISPLAY_TYPE ()
```

Lisp Syntax

```
CMSR:display-type ()
```

ARGUMENTS

None.

DESCRIPTION

`CMSR_display_type` returns the type of the currently selected display. The return values for each type of supported display are as follows:

Display Type	C Values	Fortran Values	Lisp Keywords
X11	`CMSR_x_display`	`CMSR_X_DISPLAY`	`:x-display`
CM framebuffer	`CMSR_cmfb_display`	`CMSR_CMFB_DISPLAY`	`:cmfb-display`
Symbolics			**Lispms only:**
	———	———	`:symbolics-display`
	———	———	`:symbolics-frame`

SEE ALSO

CMSR_display_bits_per_pixel

CMSR_display_width

CMSR_display_height

CMSR_display_x_offset

CMSR_display_y_offset

CMSR_display_is_color

Returns true if the current display is color.

SYNTAX

C Syntax

```
#include <cm/display.h>
int
  CMSR_display_is_color ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
LOGICAL FUNCTION CMSR_DISPLAY_IS_COLOR ()
```

Lisp Syntax

```
CMSR:display-is-color ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_display_is_color returns true (.TRUE. in Fortran, non-NULL in C, non-nil in Lisp) if the currently selected generic display is a color display, or false (.FALSE. in Fortran, NULL in C, nil in Lisp) if the display is a monochrome (i.e., grayscale) display.

SEE ALSO

CMSR_display_has_color_map

CMSR_display_color_is_rgb

CMSR_display_bits_per_pixel

CMSR_display_bits_per_pixel

Returns the number of bits per pixel set for the currently selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_display_bits_per_pixel ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_DISPLAY_BITS_PER_PIXEL ()
```

Lisp Syntax

```
CMSR:display-bits-per-pixel ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_display_bits_per_pixel returns the number of bits of color information maintained per pixel on the currently selected display.

This number is also sometimes called the *depth* of the display.

SEE ALSO

CMSR_display_width

CMSR_display_height

CMSR_display_type

CMSR_display_x_offset

CMSR_display_y_offset

CMSR_display_bits_of_blue
CMSR_display_bits_of_green
CMSR_display_bits_of_red

Returns the number of bits per pixel set for the specified color on the currently selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_display_bits_of_blue ()

int
  CMSR_display_bits_of_green ()

int
  CMSR_display_bits_of_red ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_DISPLAY_BITS_OF_BLUE ()

INTEGER FUNCTION CMSR_DISPLAY_BITS_OF_GREEN ()

INTEGER FUNCTION CMSR_DISPLAY_BITS_OF_RED ()
```

Lisp Syntax

```
CMSR:display-bits-of-blue ()

CMSR:display-bits-of-green ()

CMSR:display-bits-of-red ()
```

ARGUMENTS

None.

DESCRIPTION

For true color or direct color displays, **CMSR_display_bits_of_blue**, **CMSR_display_bits_of_green**, and **CMSR_display_bits_of_red** return the number of bits per pixel supported by the currently selected display for the respective RGB components. The length of the image buffer field allocated for images to be displayed to this display should equal the sum of these three numbers.

If the display is not a true color or direct color display, these routines return zero.

For CM framebuffers in 24-bit mode, each primary has 8 bits. For X displays, the number of bits is dependent on the display. Colors are specified in a CM field with red as the least significant bits, then green, and finally blue in the most significant (highest memory address) bits. The number of actual bits per color in the CM field *must* match the return values of these functions, or the colors will not be displayed correctly.

SEE ALSO

CMSR_display_width

CMSR_display_height

CMSR_display_type

CMSR_display_x_offset

CMSR_display_y_offset

CMSR_display_read_color

Reads a color entry from the color map of the current grayscale generic display.

SYNTAX

C Syntax

```
#include <cm/display.h>

double
  CMSR_display_read_color (index)

int index;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

REAL FUNCTION CMSR_DISPLAY_READ_COLOR (index)
```

Lisp Syntax

```
CMSR:display-read-color (index)
```

ARGUMENTS

index	An integer specifying the entry in the color map to be read.

DESCRIPTION

CMSR_display_read_color returns the value of the entry *index* in the color map of the current grayscale generic display.

This routine works only with displays that are X GrayScale visuals. Grayscale displays use a single, usually 8-bit, color map to store a range of gray intensities. A Generic Display is a grayscale display if **CMSR_display_is_color** returns false and **CMSR_display_has_color_map** returns true.

CMSR_display_width
CMSR_display_height

Returns the width (height), in pixels, of the currently selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_display_width ()


int
  CMSR_display_height ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_DISPLAY_WIDTH ()


INTEGER FUNCTION CMSR_DISPLAY_HEIGHT ()
```

Lisp Syntax

```
CMSR:display-width ()

CMSR:display_height ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_display_width returns the width, in pixels, of the currently selected display. The width of the display corresponds to the *x* (horizontal) dimension of the screen.

CMSR_display_height returns the height, in pixels, of the currently selected display window. The height of the display corresponds to the *y* (vertical) dimension of the screen.

SEE ALSO

CMSR_display_bits_per_pixel

CMSR_display_type

CMSR_display_x_offset

CMSR_display_y_offset

CMSR_display_x_offset
CMSR_display_y_offset

Returns the current starting *x* (*y*) location at which to begin image transfer to the current display.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_display_x_offset ()


int
  CMSR_display_y_offset ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_DISPLAY_X_OFFSET ()

INTEGER FUNCTION CMSR_DISPLAY_Y_OFFSET ()
```

Lisp Syntax

```
CMSR:display-y-offset ()

CMSR:display-x-offset ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_display_x_offset returns the current *x* offset used for display I/O.

CMSR_display_y_offset returns the current *y* offset used for display I/O.

CMSR_set_display_offset sets the offsets in the *x* and *y* dimensions of the display window to be used for image transfers. The image is offset *x_offset* pixels in the *x* dimension and *y_offset* pixels in the *y* dimension from the upper left corner of the display window.

SEE ALSO

CMSR_set_display_offset

CMSR:GENERIC–DISPLAY–P

[NOTE: Lisp only.] Tests whether a display is a generic display.

SYNTAX

Lisp Syntax

CMSR:GENERIC–DISPLAY–P (*display*)

ARGUMENTS

display	The display to be tested.

DESCRIPTION

CMSR:GENERIC–DISPLAY–P returns *T* if *display* is a generic display, *nil* otherwise.

SEE ALSO

CMSR_display_type

2.7 X Window System Routines

Ordinarily, you will not need to use these routines. The Generic Display Interface manages the interface to the X Window System system. However, if you are integrating your Generic Display application with an existing X Window System application or user interface, the following routines give you direct access to, and information on, the X Window System resources used by the Generic Display Interface.

CMSR_create_x_workstation

Initializes an X11 server as a generic workstation for an existing generic display.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_workstation_t
  CMSR_create_x_workstation (x_display, screen, generic_display)

Display         *x_display;
Screen          *screen;
CMSR_display_t  generic_display;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CREATE_X_WORKSTATION
&                                (x_display, screen, generic_display)

INTEGER  x_display
INTEGER  screen
INTEGER  generic_display
```

Lisp Syntax

```
CMSR:create-x-workstation
            (x-display &optional screen &optional generic-display)
```

ARGUMENTS

x_display	A pointer to an Xlib Display structure identifying the X11 server to use as the Generic Display workstation.
screen	A pointer to an Xlib Screen structure identifying which screen in *x_display* is to be used for the Generic Display workstation I/O.

Note that this is not necessarily the screen to be used as the Generic Display display space.

generic_display The Generic Display display for which the Generic Display workstation is to be created.

DESCRIPTION

CMSR_create_x_workstation initializes the X11 server identified with the Display *x_display* as a Generic Display workstation associated with the specified *generic_display,* and returns a Generic Display workstation identifier. If *screen* is set, it is used for user interaction with the workstation. If *screen* is NULL, *x_display*'s default screen is used.

If *generic_display* is NULL, the currently selected display is used. If *generic_display* is an X11 display, it must be the same display as *x_display*.

CMSR_create_x_display

Initializes and returns a Generic Display identifier for an existing X11 window.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_display_t
  CMSR_create_x_display (x_display, window, gc)

Display  *x_display;
Drawable window;
GC       gc;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CREATE_X_DISPLAY (x_display, window, gc)

INTEGER  x_display
INTEGER  window
INTEGER  gc
```

Lisp Syntax

```
CMSR:create-x-display (x-display window &optional gc)
```

ARGUMENTS

x_display	A pointer to an X Window System Display structure. This display structure must be created by calling the Xlib routine XOpen-Display (or **xlib:open-display** in Lisp).
window	An X Window System Drawable identifying the destination window for CM image data transferred to this display.

gc An X Window System GC data structure specifying the graphics context to be applied to the X window.

DESCRIPTION

Given an X window previously created and mapped, **CMSR_create_x_display** creates and returns a Generic Display display structure initialized to the desired X display and window.

If the *gc* argument is supplied, **CMSR_create_x_display** uses the defined graphics context to clear the window. If *gc* is NULL, a default GC is created that clears the window to color 0.

NOTE: The programmer must properly create and map the X11 window before calling **CMSR_create_x_display**.

SEE ALSO

CMSR_create_cmfb_display

CMSR:create-symbolics-display

CMSR_create_init_x_display

Opens a display window and returns a Generic Display display identifier for an existing X11 display.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_display_t
  CMSR_create_init_x_display
                    (x_display, default_width, default_height, visual)

Display  *x_display;
int      default_width;
int      default_height;
Visual   *visual;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CREATE_INIT_X_DISPLAY
&                   (x_display, default_width, default_height, visual)

INTEGER  x_display
INTEGER  default_width
INTEGER  default_height
INTEGER  visual
```

Lisp Syntax

```
CMSR:create-init-x-display (x–display &optional
                            default–width default–height visual)
```

ARGUMENTS

x_display	A pointer to an X11 display structure. This display structure must be created by calling the Xlib function XOpenDisplay (or **xlib:open-display** in Lisp).
default_width	The suggested width, in pixels, of the X window to be opened.
default_height	The suggested height, in pixels, of the X window to be opened.
visual	In C, a pointer to an X11 Visual structure. In Lisp, *visual* is the **visual-id**, not the Visual information structure. If *visual* is NULL, the display defaults to the screen's root visual.

DESCRIPTION

Given *x_display,* an X11 display structure, **CMSR_create_init_x_display** creates and maps a window on the default screen and returns a Generic Display display structure initialized to the desired X display and visual. An X11 display structure is created by calling the Xlib function XOpenDisplay (**xlib:open-display** in Lisp).

The *default_width* and *default_height* parameters are used as size suggestions to the X11 window manager. However, since the user may select a different size, the programmer must call **CMSR_display_width** and **CMSR_display_height** to determine the actual size of the display window.

The *visual* argument may optionally be used to reference an X11 Visual structure. This allows you to control the screen's depth and color map characteristics from the Generic Display Interface. In Lisp, visual is the **visual-id**, not the Visual information structure.

When the Generic Display display structure is deallocated, the window is destroyed.

This interface uses the X11 resource manager to set up default values. The interface has a resource class of **"CM_Display"**. The resource name is stored in the variable **CMSR_x_resource_name** (**CMSR:*X-RESOURCE-NAME*** in Lisp), which defaults to **"cm_display"**.

The resource manager defaults are as follows:

Class	Name	Description
Foreground	foreground	The foreground color of the GC to use.
Background	background	The background color of the GC to use.
borderWidth	BorderWidth	The width of the border around the window.
title	Title	The title used for the window. Defaults to the contents of **CMSR_x_default_title** (**CMSR:*X-DEFAULT-TITLE*** in Lisp), which defaults to "**CM Display**".
borderColor	BorderColor	The border color.
geometry	Geometry	The geometry with which to set up the window.

The length of the image buffer field to be transferred must be the same as the depth of the window. If the field length is longer, the high-order bits are lost. If the field length is shorter than the depth of the window, an error is signaled.

ERRORS

The Lisp version signals an error if *x-display* is not an X Display.

SEE ALSO

CMSR_create_init_cmfb_display

CMSR:create-init-symbolics-display

CMSR:create-init-symbolics-display-frame

CMSR_create_x_color_map

Allocates and returns an X11 color map filled with the specified color arrays.

SYNTAX

C Syntax

```
#include <cm/display.h>

colormap
  CMSR_create_x_color_map
          (x_display, visual, red_array, green_array, blue_array, size, cmap)

Display  *x_display;
Visual   *visual;
float    *red_array, *green_array, *blue_array;
int      size;
Colormap cmap;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CREATE_X_COLOR_MAP
&          (x_display, visual, red_array, green_array, blue_array, size, cmap)

INTEGER  x_display
INTEGER  visual
REAL     red_array(*), green_array(*), blue_array(*)
INTEGER  size
INTEGER  cmap
```

Lisp Syntax

```
CMSR:create-x-color-map (x-display visual
                         red-array green-array blue-array size
                         &optional cmap)
```

ARGUMENTS

x_display An X11 Display structure identifying the display for which the color map is to be created.

visual An X11 Visual structure identifying the visual for which the color map is to be created.

red_array, *green_array*, *blue_array*
Arrays of single floats containing the color values to be loaded into *cmap*.

size The length of the longest of *red_array*, *green_array*, and *blue_array*.

cmap An X11 color map structure to be filled with color values from *red_array*, *green_array*, and *blue_array*.

DESCRIPTION

CMSR_create_x_color_map returns an X color map on the specified *x_display* and *visual*, filled in with the specified arrays, *red_array*, *green_array*, and *blue_array*.

The color values in the arrays are single floats between 0.0 (off) and 1.0 (full intensity). The *size* argument specifies the length of the longest array. It may be shorter than the length of the display color map. The arrays may be different sizes; for example, an 8-bit true color display can use 2 bits of red, 3 bits of green, and 3 bits of blue.

If the *cmap* color map is not passed in, **CMSR_create_x_color_map** always allocates a color map for the specified *x_display* and *visual*. However, the color map is filled only if the *visual* supports writable color maps.

CMSR_create_x_color_map_named

Allocates and returns an X11 color map set to a standard Generic Display color map.

SYNTAX

C Syntax

```
#include <cm/display.h>

Colormap
  CMSR_create_x_color_map_named (x_display, visual, name, cmap)

Display  *x_display;
Visual   *visual;
char     *name;
Colormap cmap;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CREATE_X_COLOR_MAP_NAMED
&                                        (x_display, visual, name, cmap)

INTEGER          x_display
INTEGER          visual
CHARACTER*(*)    name
INTEGER          cmap
```

Lisp Syntax

```
CMSR:create-x-color-map-named (x-display visual name
                               &optional cmap)
```

ARGUMENTS

x_display	An X11 Display structure identifying the display for which the color map is to be created.

visual A X11 Visual structure identifying the visual for which the color map is to be created.

name The name of one of the standard Generic Display color maps. Valid values are:

- density
- random
- greyscale or grayscale
- rainbow

Capitalization in the names is ignored.

cmap An X11 color map structure to be filled with color values from *red_array*, *green_array*, and *blue_array*. If NULL or zero is passed in this argument, the color map structure will be allocated.

DESCRIPTION

CMSR_create_x_color_map_named returns an X color map on the specified *x_display* and *visual*, filled in with the standard Generic Display color map specified by *name*:

- *Density* is the default map. It is a scale running from dark blue to cyan to yellow to red over the length of the color map.
- *Greyscale* or *grayscale* loads the color map with a linear sequence of gray shades from black in the first index to white at the last.
- *Random* randomizes the entire red, green, and blue color maps.
- *Rainbow* puts a sinusoidal distribution of colors in the color table. The first index is set to black. The remaining positions on the color table range smoothly from red to green to blue and back to red. The intensity is constant throughout, only the hue changes.

If NULL in C or zero in Fortran is passed in *cmap*, **CMSR_create_x_color_map** always allocates a color map for the specified *x_display* and *visual*. However, the color map is filled only if the *visual* supports writable color maps.

CMSR_set_x_display_gc

Sets graphics context for currently selected X11 display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
 CMSR_set_x_display_gc (gc)

GC gc;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_X_DISPLAY_GC (gc)

INTEGER  gc
```

Lisp Syntax

```
CMSR:set-x-display-gc (gc)
```

ARGUMENTS

gc	An X11 graphics context object.

DESCRIPTION

If the current display is an X11 type display, **CMSR_set_x_display_gc** sets the graphics context to be used for output to the display.

ERRORS

An error is signaled if the current generic display is not an X11 display.

SEE ALSO

CMSR_x_display_gc

CMSR_x_display_display

CMSR_x_display_drawable

CMSR_x_workstation_display
CMSR_x_workstation_screen

Returns a pointer to the Xlib Display (Xlib Screen) structure identifying the currently selected Generic Display workstation.

SYNTAX

C Syntax

```
#include <cm/display.h>

Display *
  CMSR_x_workstation_display ()

Screen *
  CMSR_x_workstation_screen ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_X_WORKSTATION_DISPLAY ()

INTEGER FUNCTION CMSR_X_WORKSTATION_SCREEN ()
```

Lisp Syntax

```
CMSR:x-workstation-display ()

CMSR:x-workstation-screen ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_x_workstation_display returns a pointer to the X Window System Display data structure associated with the current Generic Display workstation. If there is no Generic Display workstation currently selected, **CMSR_x_workstation_display** returns NULL.

CMSR_x_workstation_screen returns a pointer to the X Window System Screen data structure associated with the current Generic Display workstation. If there is no Generic Display workstation currently selected, **CMSR_x_workstation_screen** returns NULL.

CMSR_x_workstation_font

Returns the current X11 font set for the current Generic Display workstation.

SYNTAX

C Syntax

```
#include <cm/display.h>

XFontStruct *
  CMSR_x_workstation_font ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_X_WORKSTATION_FONT ()
```

Lisp Syntax

```
CMSR:x-workstation-font ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_x_workstation_font returns an XFontStruct data structure identifying the X11 font set for the current Generic Display workstation.

CMSR_x_display_display

Returns a pointer to the current X11 Display structure.

SYNTAX

C Syntax

```
#include <cm/display.h>

Display *
  CMSR_x_display_display ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_X_DISPLAY_DISPLAY ()
```

Lisp Syntax

```
CMSR:x-display-display ()
```

ARGUMENTS

None.

DESCRIPTION

If the current display is an X11 type display, **CMSR_x_display_display** returns a pointer to the X11 display structure.

ERRORS

It is an error to call this routine if the current display is not an X11 display.

SEE ALSO

CMSR_x_display_drawable

CMSR_x_display_gc

CMSR_x_display_drawable

Returns the Drawable used for I/O with the current X11 display.

SYNTAX

C Syntax

```
#include <cm/display.h>

Drawable
  CMSR_x_display_drawable ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_X_DISPLAY_DRAWABLE ()
```

Lisp Syntax

```
CMSR:x-display-drawable ()
```

ARGUMENTS

None.

DESCRIPTION

If the current display is an X11 type display, **CMSR_x_display_drawable** returns the drawable used for I/O with the display. If the current display is not an X11 display, an error is signaled.

ERRORS

It is an error to call this routine if the current display is not an X11 display.

SEE ALSO

CMSR_x_display_display

CMSR_x_display_gc

CMSR_x_display_gc

Returns the graphics context used for I/O with the current X11 display.

SYNTAX

C Syntax

```
#include <cm/display.h>

GC
  CMSR_x_display_gc ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_X_DISPLAY_GC ()
```

Lisp Syntax

```
CMSR:x-display-gc ()
```

ARGUMENTS

None.

DESCRIPTION

If the current display is an X11 display, **CMSR_x_display_gc** returns the graphics context used for I/O with the display.

ERRORS

It is an error to call this routine if the current display is not an X11 display.

SEE ALSO

CMSR_x_display_display

CMSR_x_display_drawable

CMSR_x_visual_from_class

Returns the Visual structure supported by *screen* closest to the specified class and depth.

SYNTAX

C Syntax

```
#include <cm/display.h>

Visual *
  CMSR_x_visual_from_class (x_display, screen, class, depth)

Display  *x_display;
Screen   *screen;
int      class;
int      depth;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_X_VISUAL_FROM_CLASS
&                              (x_display, screen, class, depth)

INTEGER  x_display
INTEGER  screen
INTEGER  class
INTEGER  depth
```

Lisp Syntax

```
CMSR:x-visual-from-class (x-display
                          &optional screen (visual-class -1) depth)
```

ARGUMENTS

x_display	An X11 Display data structure specifying the display for which the visual is to be determined.

screen	An X11 Screen data structure specifying the screen on *x_display* for which the visual is to be determined.
class	The desired visual class (PseudoColor, StaticColor, DirectColor, TrueColor, GrayScale, or StaticGray) for the visual to be returned.
depth	The desired image depth (bits per pixel) for the visual to be returned.

DESCRIPTION

`CMSR_x_visual_from_class` returns the "best" visual on the specified *screen*. Best is defined as the closest match to the desired *class* and *depth* supported by the specified screen. If *screen* is NULL, the visual returned is the best match for all the screens on the display.

CMSR_set_x_window_title
CMSR_get_x_window_title

Sets (returns) the title to be used for X11 window created by Generic Display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_x_window_title (string)

char  *string;


char *
  CMSR_get_x_window_title ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_X_WINDOW_TITLE (string)

   CHARACTER*(*)  string


CHARACTER*(*) FUNCTION CMSR_GET_X_WINDOW_TITLE ()
```

Lisp Syntax

```
CMSR:set-x-window-title (string)

CMSR:get-x-window-title ()
```

ARGUMENTS

string	The string to be used as the title for Generic Display X11 windows created by the application.

DESCRIPTION

CMSR_set_x_window_title sets the character string to be used to label the X11 window opened by the Generic Display. The default title is "CM Display."

CMSR_get_x_window_title returns the current title string set for Generic Display X11 windows.

CMSR_set_x_resource_names CMSR_get_x_resource_name

Sets (returns) the application name and class to be used by the Generic Display system when reading resources.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_x_resource_names (app_name, app_class)

char  *app_name;
char  *app_class;


char *
  CMSR_get_x_resource_name ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_X_RESOURCE_NAMES (app_name, app_class)

CHARACTER*(*)  app_name
CHARACTER*(*)  app_class


CHARACTER*(*) CMSR_GET_X_RESOURCE_NAME ()
```

Lisp Syntax

```
CMSR:set-x-resource-names (app-name app-class)

CMSR:get-x-resource-name ()
```

ARGUMENTS

app_name The application name of the resource data base to be used by your program. By convention in X11 programs this is often arg[0]. The default is **cm_display**.

app_class The application class of the resource data base to be used by your program. By convention in X11 programs this is often the same as the application name but with initial capital letters. The default is **CM_Display**.

DESCRIPTION

CMSR_set_x_resource_names sets the *app_name* and *app_class* to be used by the Generic Display system when reading resources.

CMSR_get_x_resource_name returns the current Generic Display default X resource name as a character string.

The Generic Display Interface uses the X11 resource manager to set up default values. The default application resource class is **CM_Display**. The default application resource name is **cm_display** and it is stored in the variable **CMSR_x_resource_name** (**CMSR:*X-RESOURCE-NAME*** in Lisp).

CMSR_read_std_x_resources

Initializes a resource database for an X11 generic display.

SYNTAX

C Syntax

```
#include <cm/display.h>
void
  CMSR_read_std_x_resources (x_display)
Display  *x_display;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_READ_STD_X_RESOURCES (x_display)
INTEGER  x_display
```

Lisp Syntax

```
CMSR:read-std-x-resources (x–display)
```

ARGUMENTS

x_display	An X11 Display structure identifying the display from which the resources are to be read.

DESCRIPTION

CMSR_read_std_x_resources scans the X11 application defaults directory to initialize the resource database for the X11 generic display *x_display*. This routine creates a private database for the display and saves the application name and class to use as a base for reading later resources.

CMSR_set_x_app_defaults_dir sets the directory to be used as the X11 application defaults directory. By default this is **/usr/lib/X11/app-defaults/**.

CMSR_set_x_app_defaults_dir
CMSR_get_x_app_defaults_dir

Sets (returns) the directory/pathname to be used as the X11 application defaults directory.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_x_app_defaults_dir (string)

char  *string;

char *
  CMSR_get_x_app_defaults_dir ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_X_APP_DEFAULTS_DIR (string)

CHARACTER*(*)  string

CHARACTER*(*) FUNCTION CMSR_GET_X_APP_DEFAULTS_DIR ()
```

Lisp Syntax

```
CMSR:set-x-app-defaults-dir (string)

CMSR:get-x-app-defaults-dir ()
```

ARGUMENTS

string	A character string specifying the path name of the X11 application defaults directory.

DESCRIPTION

CMSR_set_x_app_defaults_dir sets the directory to be used as the X11 application defaults directory. This directory is scanned by **CMSR_read_std_x_resources** to initialize the resource database for an X11 generic display. By default this is **/usr/lib/X11/app-defaults/**.

CMSR_get_x_app_defaults_dir returns a character string containing the pathname of the current X11 application defaults directory.

CMSR_get_x_resource_class

Returns the current Generic Display default X resource class.

SYNTAX

C Syntax

```
#include <cm/display.h>

char *
  CMSR_get_x_resource_class ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

CHARACTER*(*) CMSR_GET_X_RESOURCE_CLASS ()
```

Lisp Syntax

```
CMSR:get-x-resource-class ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_get_x_resource_class returns the current Generic Display default X resource class as a string. This class may be used to access the resource database for a Generic Display X11 display.

The default class is **CM_Display**.

CMSR_get_x_resource_string
CMSR_get_x_resource_integer

Returns, as a character string (integer), the current value of the resource specified by *name* and *class*.

SYNTAX

C Syntax

```
#include <cm/display.h>

char *
  CMSR_get_x_resource_string (name, class, default_value)

int
  CMSR_get_x_resource_integer (name, class, default_value)

char  *name;
char  *class;
char  *default_value;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

CHARACTER*(*) FUNCTION CMSR_GET_X_RESOURCE_STRING
&                                    (name, class, default_value)

INTEGER FUNCTION CMSR_GET_X_RESOURCE_INTEGER
&                                    (name, class, default_value)

CHARACTER*(*)  name
CHARACTER*(*)  class
CHARACTER*(*)  default_value
```

Lisp Syntax

```
CMSR:get-x-resource-string (name class default)

   CMSR:get-x-resource-integer (name class default)
```

ARGUMENTS

name	The application name of the resource to be queried.
class	The application class of the resource to be queried.
default_value	The default value of the resource specified by *name* and *class*.

DESCRIPTION

CMSR_get_x_resource_string returns the specified X resource as a string from the private resource database for the current Generic Display X11 display. This database is created by **CMSR_read_std_x_resources**.

CMSR_get_x_resource_integer returns the specified X resource as an integer from the private resource database for the current Generic Display X11 display. This database is created by **CMSR_read_std_x_resources**.

If the resource requested by *app_name* and *app_class* is not set, the default value for that resource is returned.

2.8 CMFB Routines

As with the X11 routines in the previous section, you will not usually need to use these CMFB routines. The Generic Display Interface manages the interface to the CM framebuffer. However, the following routines give you direct access to, and information on, the CM framebuffer being used as the current generic display:

CMSR_create_cmfb_display

Returns a Generic Display display identifier for an existing CM framebuffer display.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_display_t
  CMSR_create_cmfb_display (display_id, buffer_id)

CMFB_display_id_t display_id;
CMFB_buffer_id_t  buffer_id;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CREATE_CMFB_DISPLAY
                                        (display_id, buffer_id)

INTEGER  display_id
INTEGER  buffer_id
```

Lisp Syntax

```
CMSR:create-cmfb-display (display-id &optional
                                    (buffer-id :current))
```

ARGUMENTS

display_id	A CM framebuffer display identifier for the framebuffer that is to be attached.
buffer-id	A keyword or named constant specifying the color buffer, or group of buffers, to which to write the color data. Valid values for *buffer_id* are:

C Values	Fortran Values	Lisp Keywords	# of Bits Transferred
`CMFB_red`	`CMFB_red`	`:red`	8
`CMFB_green`	`CMFB_green`	`:green`	8
`CMFB_blue`	`CMFB_blue`	`:blue`	8
`CMFB_overlay`	`CMFB_overlay`	`:overlay`	8
`CMFB_rgb`	`CMFB_rgb`	`:rgb`	24
`CMFB_all`	`CMFB_all`	`:all`	32
`CMFB_current`	`CMFB_current`	`:current`	8, 24, or 32 *
`NULL`	`0`	`nil`	8, 24, or 32 *

* The number of bits transferred depends on the image resolution of the current display.

DESCRIPTION

`CMSR_create_cmfb_display` returns a CM generic display structure initialized to the specified CM framebuffer display and buffer.

NOTE: The programmer must properly create and initialize the CM framebuffer display specified by *display_id* before calling **`CMSR_create_cmfb_display`**.

The *buffer_id* argument sets the current buffer for the display. Image data transferred to this display is written to *buffer_id* color buffer or buffers.

ERRORS

It is an error to call **`CMSR_create_cmfb_display`** if the *display_id* is not an initialized CM framebuffer display.

SEE ALSO

`CMSR_create_init_cmfb_display`

`CMSR_create_init_x_display`

`CMSR_create_x_display`

CMSR_create_init_cmfb_display

Attaches and initializes the specified framebuffer and returns a Generic Display ID.

SYNTAX

C Syntax

```
#include <cm/display.h>

CMSR_display_t
  CMSR_create_init_cmfb_display (display_location  bits_per_pixel)

char   *display_location;
int    bits_per_pixel;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CREATE_INIT_CMFB_DISPLAY
&                                        (display_location, bits_per_pixel)

CHARACTER*(*)   display_location
INTEGER         bits_per_pixel
```

Lisp Syntax

```
CMSR:create-init-cmfb-display (&optional display-location
                                         (bits-per-pixel 8))
```

ARGUMENTS

display_location	A string naming the framebuffer that is to be attached. If NULL, the default display is used.
bits_per_pixel	The number of bits of color data to be supported for each pixel in the display.

DESCRIPTION

CMSR_create_init_cmfb_display attaches and initializes the CM framebuffer named by *display_location*. A CM generic display structure identifying the framebuffer is returned.

The *display_location* argument is passed on to **CMFB_attach_display** and may be NULL to indicate you want to attach to the default framebuffer. The *bits_per_pixel* argument is passed onto **CMFB_initialize_display**.

ERRORS

If the framebuffer cannot be attached, an error is signaled in Lisp. In C, NULL is returned and the variable **CMFB_errno** holds the failure reason. The defined error codes for **CMFB_attach_display** are as follows:

CMFB_errno	Description
0	Display was successfully attached.
–1	No framebuffers were available. (Only if *location_string* is NULL.)
–2	*location_string* was not found in the configuration file.
–3	Could not reach the framebuffer over the I/O bus (may indicate a hardware problem).

SEE ALSO

CMSR_create_init_x_display

CMSR:create_init_symbolics_display

CMSR:create_init_symbolics_display_frame

CMSR_create_cmfb_display

CMSR_create_x_display

CMSR:create_symbolics_display

CMSR_set_cmfb_display_buffer_id
CMSR_cmfb_display_buffer_id

Sets (returns) the color buffer ID for a CM framebuffer display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_set_cmfb_display_buffer_id (buffer_id)

CMFB_buffer_id_t buffer_id;


CMFB_buffer_id_t
  CMSR_cmfb_display_buffer_id ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_CMFB_DISPLAY_BUFFER_ID (buffer_id)

CMFB_buffer_id_t buffer_id;

INTEGER FUNCTION CMSR_CMFB_DISPLAY_BUFFER_ID ()
```

Lisp Syntax

```
CMSR:set-cmfb-display-buffer-id ()

CMSR:cmfb-display-buffer-id ()
```

ARGUMENTS

buffer_id A keyword or named constant specifying the color buffer that is to become the current buffer for a CM framebuffer Generic Display. Valid values for *buffer_id* are:

C Values	Fortran Values	Lisp Keywords
`CMFB_green`	`CMFB_green`	`:green`
`CMFB_blue`	`CMFB_blue`	`:blue`

DESCRIPTION

If the current display is a CM framebuffer display in 8-bit-per-pixel mode, **`CMSR_set_cmfb_display_buffer_id`** sets the "current buffer" for the framebuffer to *buffer_id*. The current buffer is the color buffer in framebuffer display memory, either green or blue, currently selected for display. When the display system is in 8-bit-per-pixel mode, the current buffer is used as the index into all three color lookup tables.

`CMSR_cmfb_display_buffer_id` returns the *buffer_id* identifying the color buffer currently selected for I/O with the display. It is an error to call this routine with a display that is not a CM framebuffer display.

ERRORS

An error is signaled if **`CMSR_cmfb_display_buffer_id`** is called with a display that is not a CM framebuffer.

SEE ALSO

`CMSR_cmfb_display_display_id`

CMSR_cmfb_display_display_id

Returns the CMFB display ID of the CM framebuffer associated with the current generic display.

SYNTAX

C Syntax

```
#include <cm/display.h>
CMFB_display_id_t
  CMSR_cmfb_display_display_id ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_CMFB_DISPLAY_DISPLAY_ID ()
```

Lisp Syntax

```
CMSR:cmfb-display-display-id ()
```

ARGUMENTS

None.

DESCRIPTION

If the current display is a CM framebuffer display, **CMSR_cmfb_display_display_id** returns the display ID identifying the currently selected framebuffer display. It is an error to call this routine with a display that is not a CM framebuffer.

ERRORS

It is an error to call this routine if the current display is not a CM framebuffer display.

SEE ALSO

CMSR_cmfb_display_buffer_id

Chapter 3

Generic Text Routines

The Generic Text routines write text strings to the current Generic Display (either an X11 window or the CM framebuffer), or write text directly into an image buffer in CM memory. These routines are designed to allow you to add labels to your images, or, in conjunction with the Generic Mouse and Generic Display workstation routines, to create interactive displays.

This overview outlines the Generic Text system. For more detail, see the descriptions of the individual routines that follow.

3.1 Overview

3.1.1 Displaying and Drawing Text

Three Generic Text display routines write text directly into the current Generic Display:

- **`CMSR_display_text`** (*string, x, y*)
- **`CMSR_display_text_centered`** (*string, x, y*)
- **`CMSR_display_outline_text`** (*string, x, y*)

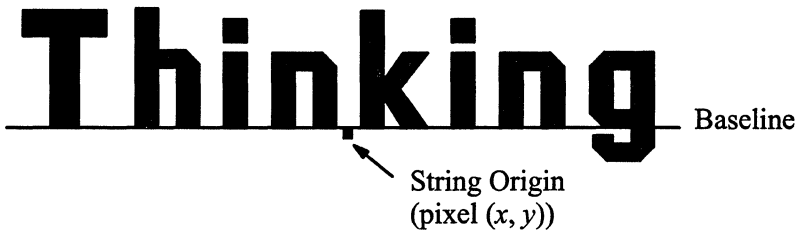
Three analogous drawing routines write text into an image buffer field in CM memory:

- **`CMSR_draw_text`** (*string, x, y, field, depth*)
- **`CMSR_draw_outline_text`** (*string, x, y, field, depth*)
- **`CMSR_draw_text_centered`** (*string, x, y, field, depth*)

These routines write the specified *string* of text to the location (x, y) specified in screen or image buffer coordinates. Remember that all the Generic Display and *Render routines assume a coordinate system with the origin (0,0) in the upper left corner, and in which x increases to the right, and y increases downwards.

`CMSR_display_text`, **`CMSR_display_outline_text`**, **`CMSR_draw_text`**, and **`CMSR_draw_outline_text`** position the left edge of the string baseline at (x, y):

`CMSR_display_text_centered` and **`CMSR_draw_text_centered`** position the center of the string baseline at (x, y):



The programmer is responsible for positioning the text so that the string is visible; characters (or portions of characters) that extend beyond the boundaries of the display or image buffer are clipped. Routines included with Generic Text allow you to determine the extent of a string so that it may be positioned exactly. See Section 3.1.3 below for more information on positioning text strings.

The string is rendered using the current value of three Generic Text parameters: text font, text colors, and text drawing mode. Defaults are provided so that you may begin using the text routines immediately:

- The default font is a 16-point label font, **`Think_Label`**, supplied by the Generic Text system. (This font does not depend on an X server, that is, it is available even when no generic workstation is selected.)
- The default background color is color 0, and the default foreground color is the highest color in the current color map.

- The default text drawing mode is used to render the text in the foreground color only.

The next section gives more information on setting these parameters.

CMSR_display_text, **CMSR_draw_text**, **CMSR_display_text_centered** and **CMSR_draw_text_centered** draw the specified string in the current text foreground color. **CMSR_display_outline_text**, and **CMSR_draw_outline_text** write the text in the current text foreground color with a one-pixel border in the current text background color.

3.1.2 Setting Text Parameters

Setting the Text Font

The text font determines the size and style of the characters displayed. You can set the current text font with **CMSR_set_font** (*font_name*). You can get the name of the font currently set with the information routine **CMSR_font_name**( ). The font name can always be one of two Generic Text fonts, **Think_Label** or **Think_Title**. If a Generic Display workstation is currently selected, the font name can also be any of the the fonts supported by that workstation specified in the X11R4 font name format.

Think_Label and **Think_Title** are built into the Generic Text system and can be used with any Generic Display. **Think_Label**, the default, is a 16-point constant-width serif font, and **Think_Title** is the same character set in a 24-point size.

If a Generic Display workstation is currently selected, you can also select any fonts available on the workstation by specifying the font name in the X11R4 format. You can list the fonts available on your workstation with the X11 shell command **xlsfonts**. In addition, an X11 interactive font selection tool, **xfontsel**, provides you with menus of the choices available for each font parameter and displays the font character set as you make your choices. See the description of **CMSR_set_font** for more information on specifying X11 fonts, and see your X Window System documentation for a complete explanation.

Setting Text Colors

The Generic Text system defines two text colors, foreground and background. You can set the colors with **CMSR_set_text_colors** (*foreground, background*) and query the current color settings with **CMSR_text_foreground_color**( ) and **CMSR_text_background_color**( ).

The colors are specified as indices into the color map of the currently selected Generic Display. The defaults are color 0 for the background, and the highest possible color on the color map for the foreground.

The use of the background and foreground colors depends on the text drawing mode, as explained below.

Text Drawing Modes

The text drawing mode determines how the text colors will be used to draw the characters. You can set the mode with **`CMSR_set_text_draw_mode`** (*mode*), and return the current mode with **`CMSR_text_draw_mode`**`( )`. The mode may be foreground only, foreground and background, or foreground xor'd into the display:

- The foreground only mode, **`CMSR_text_fg_only`** (1 in Fortran), draws the characters in the foreground text color over the existing display background. This is the default.
- Foreground and background mode, **`CMSR_text_fg_bg`** (2 in Fortran), draws the characters in the text foreground color and creates a rectangle in the text background color bounding the text string. The effect is to set the text string off from the display background by surrounding it with a box in the text background color.
- The foreground xor mode, **`CMSR_text_xor`** (3 in Fortran), draws the characters into the display in XOR mode using the foreground color. This means that any bit plane in the display corresponding to a 1 in the text foreground color is inverted. The effect is that the text pixel values directly change the color of the display pixels. This is useful for drawing and undrawing text; in foreground-xor mode, a second call to a text draw or display routine with the same string and coordinates erases the text displayed with the first call.

3.1.3 Positioning Text

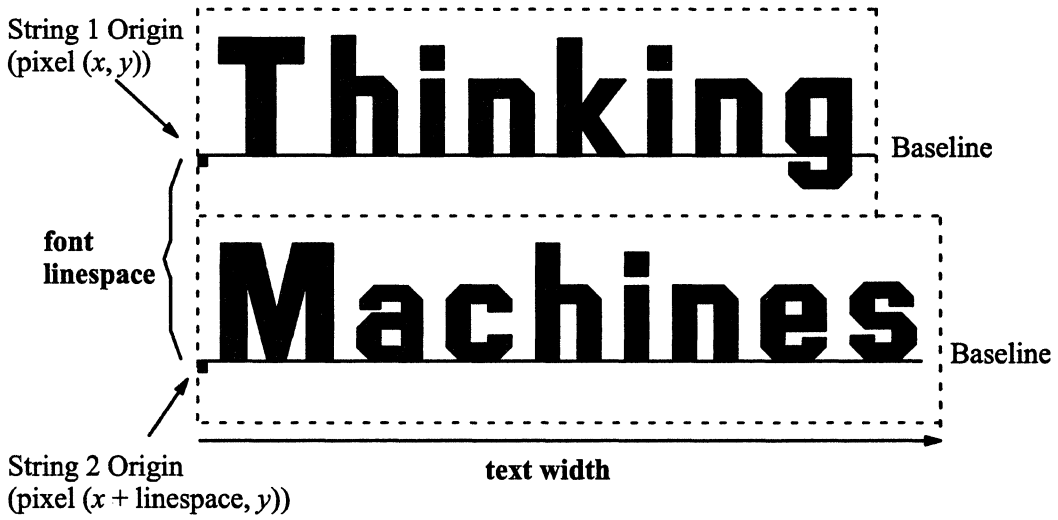
As mentioned above, the Generic Text display and drawing routines specify the location of the origin of the string in display or image buffer coordinates. You should usually position the text so that the entire string is visible. Any portion of the string that falls beyond the boundary of the display or image buffer is clipped.

If you have ample space in your image to display the text, there are two Generic Text routines, **`CMSR_font_linespace`** and **`CMSR_text_width`**, that make it easy to position

successive strings of text. If you must position text more precisely, either because of limited space or because you want to locate the string at a precise point in the image, you can use the more detailed methods for determining the logical, font, and actual extents of a string described below.

Using CMSR_font_linespace and CMSR_text_width

CMSR_font_linespace( ) returns the number of pixels that should ordinarily be used to space between lines in the current font. **CMSR_text_width**(*string*) returns the length in pixels of the specified string. This length includes the horizontal spacing to the left of the first, and to the right of the last, characters in the string. These dimensions are illustrated below.



To add a second line of text below one positioned at (*x*, *y*), locate the second at (*x*, (*y* + (**CMSR_font_linespace**( )))).

Similarly, to determine that a string created by **CMSR_display_text** or **CMSR_draw_text** at (*x*, *y*) will not run off the right edge of the screen, check to see that (*x* + **CMSR_text_width**(*string*)) is within the display or image buffer boundaries. If you are using centered text created with **CMSR_display_text_centered** or **CMSR_draw_text_centered**, you can determine the left edge of the string by finding (*x* – (**CMSR_text_width**(*string*)/2)), and the right edge of the string by finding (*x* + (**CMSR_text_width**(*string*)/2)).

The text width is also useful for adding strings of text horizontally. If you wish to butt a line of text up against the right end of a string created with **CMSR_display_text** or **CMSR_draw_text** at (*x*, *y*), locate the second string at ((*x* + **CMSR_text_width** (*string*)), *y*).

Using the Extents of a String

If you must squeeze text into a limited space or locate a string precisely in relation to an object in your image, you may need more precise measurements. Generic Text provides three different ways to measure a string's *extents* that are useful in different circumstances.

A string's extents are measured in four directions –left, right, bottom, and top– from the left edge of the string's baseline. Generic Text provides three different routines to return different measurements of a string's extents:

- **CMSR_text_logical_extents**(*string, extents_array*)
- **CMSR_text_actual_extents**(*string, extents_array*)
- **CMSR_font_extents**(*string, extents_array*)

You can then read specific values out of the extents array by passing it to the appropriate accessor routine:

- **CMSR_left_extent**(*extents_array*)
- **CMSR_right_extent**(*extents_array*)
- **CMSR_bottom_extent**(*extents_array*)
- **CMSR_top_extent**(*extents_array*)

Each of these different text metrics is explained briefly below. See the descriptions of the individual routines in this reference manual for more details.

Logical Text Extents

CMSR_text_logical_extents returns a string's logical extents. The logical extents are the dimensions defined by the font for the characters, including vertical space for line spacing and horizontal space (kerning) for character spacing:

These dimensions allow you to safely position adjacent strings without explicitly manipulating spacing. For example, if you are positioning text near the top of the screen, you can locate the baseline of the string at y + top extent and be assured that most characters will fit with an appropriate margin. A string's right logical extent is the same as the distance returned by **CMSR_text_width**.

Note, however, that parts of some characters may extend beyond the logical extent of the string. For example, the slant of an italic font often extends characters into the bounding box of previous and succeeding characters, and accented capital letters often extend slightly above the top extent. In practice, this is usually a concern only when specifying the origin of a line near the edge of the display space or of another object. In these cases, be sure to leave additional margin for the text so that characters that extend beyond the logical bounding box will not be clipped or obscured.

Actual Text Extents

CMSR_text_actual_extents returns a string's actual extents. The actual extents of a string are the dimensions of the text "ink" bounding box measured in pixels from the left edge of the string baseline. These dimensions allow you to determine the extent of the characters themselves in this particular string, not including any additional spacing. In contrast to the right logical extent, for example, which includes character spacing to the right of the last character in the string, the right actual extent measures only to the last pixel illuminated by the right-most character.

The actual extents are useful when you need to squeeze text into a limited space or place it at an exact point in the display. For example, the left extent is the number of pixels you can move the string origin to the left before obscuring the character. To place the edge of the left-most character itself, rather than the left edge of the string baseline, at a specific point *(x, y)* locate the string at (*x*, (*y* – *left–extent*)).

Note that the actual extents are specific to each string. The logical top extent, for example, will be the same for every string in the same font, but the actual top extent is determined by the height of the tallest character above the baseline in this string. When using actual extents, you must control the spacing for each string you display.

NOTE

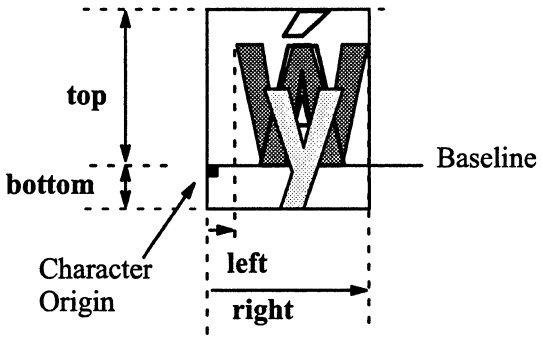
The actual extent is not computed for the Generic Display fonts **`Think_Label`** and **`Think_Title`**. When using these fonts **`CMSR_text_actual_extents`** returns the logical extent of the string and a warning will be generated:

```
Warning: Built-in fonts not tested for actual
extents.
```

Font Text Extents

CMSR_font_extents returns the maximum actual (ink) extent of the largest character in the font for each dimension. For example, left is the number of pixels from the character origin to the left edge of the widest character in the font; bottom is the number of pixels from the baseline to the bottom to the character that extends furthest below the baseline.

All measurements are in pixels from the left edge of the baseline:



You can use these extents to determine spacing that can accommodate the largest characters in the font, not just the largest in the specified string. This allows you to position text more tightly than the logical extents allow, but also, unlike the actual extents, to establish a consistent spacing that will accommodate any characters in the font. For example, the top font extent determines the minimum number of pixels above the baseline needed to fit the tallest character in the font without any interline spacing.

3.2 Generic Text Operations

This section contains the descriptions of the individual routines that make up the Generic Text system:

Text Display Routines

Text Parameter Routines

CMSR_display_text
CMSR_display_text_centered

Writes a string of text to the currently selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_display_text (string, x, y);
char *string;
int   x, y;

void
  CMSR_display_text_centered (string, x, y);
char *string;
int   x, y;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_DISPLAY_TEXT (string, x, y)
CHARACTER*(*)  string
INTEGER  x, y

SUBROUTINE CMSR_DISPLAY_TEXT_CENTERED (string, x, y)
CHARACTER*(*)  string
INTEGER  x, y
```

Lisp Syntax

```
CMSR:display-text (string, x, y)
CMSR:display-text-centered (string, x, y)
```

ARGUMENTS

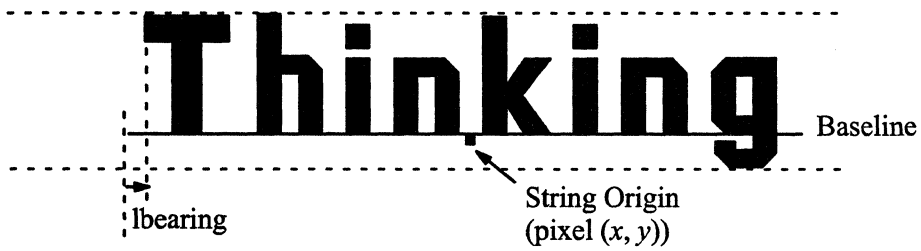
string	The string to be written to the display.
x, y	The display space coordinates at which to begin drawing *string*.

DESCRIPTION

CMSR_display_text and **CMSR_display_text_centered** write the *string* of text to the currently selected display at the screen coordinates specified by (x, y) using the current text font and color.

For **CMSR_display_text**, the coordinates (x, y) mark the left edge of the text string baseline. The string begins at the *lbearing* of the left-most character.

For **CMSR_display_text_centered**, the coordinates (x, y) mark the center of the string's baseline. The baseline begins at the *lbearing* of the left-most character. Note that the center point of the text may not be at the center of the baseline.



For both routines, the baseline marks the bottom of the character bodies; any character descenders extend below the baseline. Any portions of the string that extend beyond the edges of the display are clipped. Any newlines or other control characters in the string will be printed using whatever character occupies that position in the current font. Control characters are not interpreted in any way.

If the current display is an X11 window, **CMSR_display_text** and **CMSR_display_text_centered** render the text string directly into the display using X11 calls.

If the current display is a CM framebuffer display, **CMSR_display_text** and **CMSR_display_text_centered** read the image from the display into a temporary field in CM memory, combine the text array with the image, and write the image back to the display. Another routine, **CMSR_draw_text** is provided to draw text directly into an image buffer field in CM memory.

NOTE: Text is not preserved during display updates. New image data written to the display by **CMSR_write_to_display** will overwrite the text.

SEE ALSO

CMSR_draw_text (*string, x, y, field, depth*)

CMSR_draw_text_centered (*string, x, y, field, depth*)

CMSR_display_outline_text

Writes a string of outlined text to the currently selected display.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_display_outline_text (string, x, y);

char *string;
int   x, y;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_DISPLAY_OUTLINE_TEXT (string, x, y)

CHARACTER*(*)  string
INTEGER  x, y
```

Lisp Syntax

```
CMSR:display-outline-text (string, x, y)
```

ARGUMENTS

string	The string to be written to the display.
x, y	The display space coordinates at which to begin drawing *string*.

DESCRIPTION

CMSR_display_text writes the *string* of text to the currently selected display at the screen coordinates specified by (x, y) using the current text font. The string is written in

the current text foreground color surrounded by a one-pixel border in the current text background color.

The coordinates (x, y) mark the left edge of the text string baseline. The string begins at the *lbearing* of the left-most character.

The baseline marks the bottom of the character bodies; any character descenders extend below the baseline. Any portions of the string that extend beyond the edges of the display are clipped. Any newlines or other control characters in the string will be printed using whatever character occupies that position in the current font. Control characters are not interpreted in any way.

If the current display is an X11 window, **`CMSR_display_outline_text`** renders the text string directly into the display using X11 calls.

If the current display is a CM framebuffer display, **`CMSR_display_outline_text`** reads the image from the display into a temporary field in CM memory, combines the text array with the image, and writes the image back to the display. Another routine, **`CMSR_draw_outline_text`** is provided to draw outlined text directly into an image buffer field in CM memory.

NOTE: Text is not preserved during display updates. New image data written to the display by **`CMSR_write_to_display`** will overwrite the text.

SEE ALSO

`CMSR_draw_text` (*string, x, y, field, depth*)

`CMSR_draw_text_centered` (*string, x, y, field, depth*)

CMSR_draw_text
CMSR_draw_text_centered

Draws a string of text into an image buffer or other 2D field in CM memory.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_draw_text (string, x, y, field, depth) ;
char  *string;
int   x, y;
CM_field_id_t  field;
unsigned int   depth;

void
  CMSR_draw_text_centered (string, x, y, field, depth) ;
char  *string;
int   x, y;
CM_field_id_t  field;
int   depth;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_DRAW_TEXT (string, x, y, field, depth)
CHARACTER*(*)  string
INTEGER  x, y
INTEGER  field
INTEGER  depth

SUBROUTINE CMSR_DRAW_TEXT_CENTERED (string, x, y, field, depth)
CHARACTER*(*)  string(*)
INTEGER  x
INTEGER  y
INTEGER  field
INTEGER  depth
```

Lisp Syntax

CMSR:draw-text (*string, x, y, field, depth*)

CMSR:draw-text-centered (*string, x, y, field, depth*)

ARGUMENTS

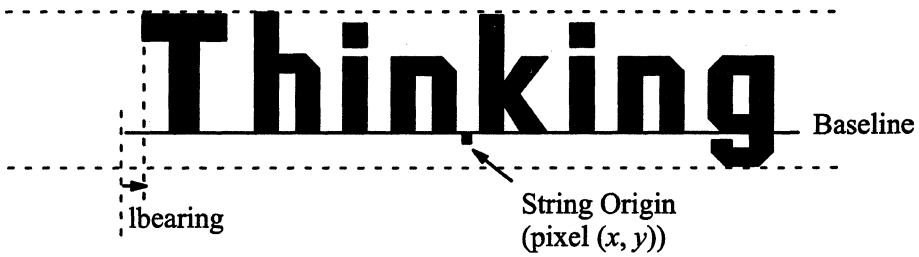
string	The characters to be drawn into *field*.
x, y	The image-buffer coordinates at which to begin drawing *string*.
field	The 2D Paris field in CM memory into which to write the string. The field must be part of a 2D geometry.
depth	The length of the *field* (in bits).

DESCRIPTION

CMSR_draw_text and **CMSR_draw_text_centered** write a string of text into the specified field in CM memory at the image buffer coordinates specified by (*x*, *y*) using the current text font and color.

For **CMSR_draw_text**, the coordinates (*x*, *y*) mark the left edge of the baseline. The string begins at the *lbearing* of the left-most character.

For **CMSR_draw_text_centered**, the coordinates (*x*, *y*) mark the center of the string's baseline. The baseline begins at the *lbearing* of the left-most character; note that the center point of the text may not be at the center of the baseline.



For both routines, the baseline marks the bottom of the character bodies; any character descenders extend below the baseline. Any portions of the string that extend beyond the dimensions of the image buffer geometry are clipped. Any newlines or other control characters in the string will be printed using whatever character occupies that position in the current font. Control characters are not interpreted in any way.

If the current generic display is an X11 window, **CMSR_draw_text** or **CMSR_draw_text_centered** will be slower than **CMSR_display_text**, since the text bitmap must be sent to the CM, and then the entire display bitmap sent to the display device. However, if the image buffer field is to be displayed several times, it is best to draw the text directly into the field, and then repeatedly write the display to the X window with **CMSR_write_to_display**.

NOTE: Text drawn to the image buffer is not preserved during image updates. New image data written to the image buffer will overwrite the text drawn by **CMSR_draw_text**.

SEE ALSO

CMSR_display_text (*string, x, y*)

CMSR_display_text_centered (*string, x, y*)

CMSR_draw_outline_text

Draws a string of text into an image buffer or other 2D field in CM memory.

SYNTAX

C Syntax

```
#include <cm/display.h>

void
  CMSR_draw_outline_text (string, x, y, field, depth);
char  *string;
int   x, y;
CM_field_id_t  field;
unsigned int   depth;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_DRAW_OUTLINE_TEXT (string, x, y, field, depth)
CHARACTER*(*)  string
INTEGER  x, y
INTEGER  field
INTEGER  depth
```

Lisp Syntax

```
CMSR:draw-outline-text (string, x, y, field, depth)
```

ARGUMENTS

string	The characters to be drawn into *field*.
x, y	The image-buffer coordinates at which to begin drawing *string*.
field	The 2D Paris field in CM memory into which to write the string. The field must be part of a 2D geometry.
depth	The length of the *field* (in bits).

DESCRIPTION

CMSR_draw_outline_text writes a string of text into the specified field in CM memory at the image buffer coordinates specified by (*x*, *y*). The string is drawn in the current font and the current text foreground color surrounded by a one-pixel border in the current text background color.

The coordinates (*x*, *y*) mark the left edge of the baseline. The string begins at the *lbearing* of the left-most character.

The baseline marks the bottom of the character bodies; any character descenders extend below the baseline. Any portions of the string that extend beyond the dimensions of the image buffer geometry are clipped. Any newlines or other control characters in the string will be printed using whatever character occupies that position in the current font. Control characters are not interpreted in any way.

If the current generic display is an X11 window, **CMSR_draw_outline_text** will be slower than **CMSR_display_outline_text**, since the text bitmap must be sent to the CM, and then the entire display bitmap sent to the display device. However, if the image buffer field is to be displayed several times, it is best to draw the text directly into the field, and then repeatedly write the display to the X window with **CMSR_write_to_display**.

NOTE: Text drawn to the image buffer is not preserved during image updates. New image data written to the image buffer will overwrite the text drawn by **CMSR_draw_text**.

SEE ALSO

CMSR_display_text (*string, x, y*)

CMSR_display_text_centered (*string, x, y*)

CMSR_set_font
CMSR_font_name

Sets (returns) the current font for the current generic workstation.

SYNTAX

C Syntax

```
#include <cm/display.h>
int
  CMSR_set_font (font_name)
char * font_name;

char *
  CMSR_font_name ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_SET_FONT (font_name)
CHARACTER*(*) font_name(*)

CHARACTER*(*) FUNCTION CMSR_FONT_NAME ()
```

Lisp Syntax

```
CMSR:set-font (font-name)
CMSR:font-name ()
```

ARGUMENTS

font_name The name of the font to be set as the current font; may be any of the following:

- **Think_Label**
  A 16-point label font, the default. This font is equivalent to the X11R4 font:

–sony–fixed–medium–r–normal—
16–150–75–75–c–80–iso8859–1

- **Think_Title**
  A 24-point title font. This font is equivalent to the X11R4 font:
  –sony–fixed–medium–r–normal—
  24–230–75–75–c–120–iso8859–1

- An X11R4 font name in the same format as the sony fonts given above.

The specific X11 fonts available to you depends on the fonts supported by the currently selected Generic Display workstation. **Think_Label** and **Think_Title** are always available.

DESCRIPTION

CMSR_set_font sets the current font to be used by **CMSR_draw_text**, **CMSR_display_text**, **CMSR_draw_text_centered**, and **CMSR_display_text_centered**.

CMSR_font_name returns the name of the current Generic Text font. Currently the font name is **Think_label**, **Think_Title**, or one of the X11R4 font names.

Two constant-width fonts, **Think_Label** and **Think_Title**, are provided by the Generic Text software. These fonts are always available and work even when no Generic Display workstation is selected.

If a Generic Display workstation is currently selected, you may also select any of the X11 fonts available on it by specifying the X11R4 font name. If no workstation is selected, or the specified font is not found on the current workstation, **CMSR_set_font** returns 0 and the font is unchanged. If the font is set successfully, **CMSR_set_font** returns a non-zero value.

NOTE: The X11 fonts are a property of the currently selected workstation. If the current workstation changes, the font also changes to the current font defined on the new workstation.

The X11 font names are a string of up to 15 parameters. Examples are given in the description of **Think_Label** and **Think_Title** in the Arguments section above. You can list the fonts available on your workstation with the X11 shell command **xlsfonts**. The X Window System also provides an interactive font selection tool,

xfontsel, which can be run from an X11 window. This tool provides you with menus of the choices available for each font parameter and displays the font character set as you make your choices. For more information on the X11 fonts, see your X11 documentation.

ERRORS

If an error occurs, **CMSR_set_font** returns 0 and the font is unchanged.

CMSR_set_text_draw_mode
CMSR_text_draw_mode

Sets (returns) the current text drawing mode for the current generic workstation.

SYNTAX

C Syntax

```
#include <cm/display.h>
void
  CMSR_set_text_draw_mode (mode);
CMSR_text_draw_mode_t   *mode;

CMSR_text_draw_mode_t
  CMSR_text_draw_mode ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_SET_TEXT_DRAW_MODE (mode)
INTEGER  mode(*)

INTEGER FUNCTION CMSR_TEXT_DRAW_MODE ()
```

Lisp Syntax

```
CMSR:set-text-draw-mode (mode)
CMSR:text-draw-mode (mode)
```

ARGUMENTS

mode Valid values are as follows:

- **CMSR_text_fg_only**
  The default mode. Bits that are set in the font are written into the destination with the text foreground color set by **CMSR_set_text_colors**. Bits that are clear in the font

(background bits) are unmodified in the destination. The effect is that the text appears over the existing display background.

- **CMSR_text_fg_bg**
  Foreground bits in the font are written in the foreground color set by **CMSR_set_text_colors**, and background bits in the font are written in the background color set by **CMSR_set_text_colors**. This creates a rectangle in the text background color around the text string, which is displayed in the text foreground color.

- **CMSR_text_xor**
  The foreground bits in the font are drawn into the destination foreground color in XOR mode. This means that any bit planes which are 1 in the font are inverted in the display. The effect is that the text itself changes the color of whatever was underneath it. This is useful for drawing and undrawing text, since if the same text is drawn twice with the same mode and color it will disappear.

DESCRIPTION

CMSR_set_text_draw_mode sets the current text drawing mode for the current display. Text written to the display by **CMSR_display_text**, **CMSR_display_text_centered**, **CMSR_draw_text**, and **CMSR_draw_text_centered** will be drawn using the method specified by *mode*.

CMSR_text_draw_mode returns the current text drawing mode of the current display.

The value returned will be one of the following:

- **CMSR_text_fg_only** (1 in Fortran)
- **CMSR_text_fg_bg** (2 in Fortran)
- **CMSR_text_xor** (3 in Fortran)

The meaning of these drawing modes is explained the the Arguments section above.

NOTE: The text drawing mode is a property of the current generic workstation. If you change the current workstation, the text draw mode also changes to the current text draw mode of the new workstation.

CMSR_set_text_colors

Sets the current foreground and background colors for text drawing.

SYNTAX

C Syntax

```
#include <cm/display.h>
void
  CMSR_set_text_colors (foreground, background)
unsigned int  foreground, background
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_SET_TEXT_COLORS (foreground, background)
INTEGER  foreground, background
```

Lisp Syntax

```
CMSR:set-text-colors (foreground, background)
```

ARGUMENTS

foreground, background

The color map indices of the foreground and background colors to be used to draw text. The defaults are color 0 for the background, and the highest possible color in the current color map for the foreground.

Note that the color values specified here are not RGB values but indices into the display's color map. The colors actually displayed depend on the colors currently set at these indices in the current display's color map.

DESCRIPTION

CMSR_set_text_colors sets the current foreground and background colors for text drawing. The current text drawing mode, set with **CMSR_set_text_draw_mode**, determines how these colors are used to display the text.

NOTE: The text colors are a property of the current generic workstation. If you change the selected workstation, the text colors also change to the current text colors of the new workstation.

SEE ALSO

CMSR_text_foreground_color( )

CMSR_text_background_color( )

CMSR_text_foreground_color
CMSR_text_background_color

Returns the current text foreground (background) color for the current display.

SYNTAX

C Syntax

```
#include <cm/display.h>

unsigned int
  CMSR_text_foreground_color ();


unsigned int
  CMSR_text_background_color ();
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_TEXT_FOREGROUND_COLOR ()


INTEGER FUNCTION CMSR_TEXT_BACKGROUND_COLOR ()
```

Lisp Syntax

```
CMSR:text-foreground-color ()

CMSR:text-background-color ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_text_foreground_color returns the index of the color map element for current Generic Text foreground color.

CMSR_text_background_color returns the index of the color map element for the current Generic Text background color.

The current foreground and background colors for text drawing are set with **CMSR_set_text_colors**. The use of these colors in rendering text is controlled by **CMSR_set_text_draw_mode**.

NOTE: The Generic Text colors are a property of the current generic workstation. If you change the selected workstation, the text colors also change to the current text colors of the new workstation. Note also that these values are indices into the current display's color map, not RGB values. The colors actually displayed depend on the colors set at these indices in the current color map for the display.

CMSR_font_linespace

Returns the standard interline spacing defined for the current font.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_font_linespace ();
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_FONT_LINESPACE ()
```

Lisp Syntax

```
CMSR:font-linespace ()
```

ARGUMENTS

None.

DESCRIPTION

`CMSR_font_linespace` returns the number of pixels that should be used to space between lines of text in the current font according to the font definition. The distance is measured from baseline to baseline (see the figure below).

For example, to position two lines with proper vertical spacing, draw the first at (x, y) and the second at (x, (y + *font_linespace*)).

CMSR_text_width

Returns the horizontal length, in pixels, of a specified string in the current font.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_text_width (string);

char *string;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_TEXT_WIDTH (string)
```

Lisp Syntax

```
CMSR:text-width (string)
```

ARGUMENTS

string	The string for which the width is to be determined.

DESCRIPTION

CMSR_text_width returns the width, in pixels, of the specified *string* in the current font. This length includes the horizontal spacing to the left of the first, and to the right of the last, characters in the string, as illustrated below.

To determine that a string created by **CMSR_display_text** or **CMSR_draw_text** at (*x*, *y*) will not run off the right edge of the screen, check to see that (*y* + **CMSR_text_width**(*string*)) is within the display or image buffer boundaries. If you are using centered text created with **CMSR_display_text_centered** or **CMSR_draw_text_ centered**, you can determine the left edge of the string by finding (*y* – (**CMSR_text_width**(*string*)/2)), and the right edge of the string by finding (*y* + (**CMSR_text_width**(*string*)/2)).

The text width is also useful for adding strings of text horizontally. If you wish to butt a line of text up against the right end of a string created with **CMSR_display_text** or **CMSR_draw_text** at (*x*, *y*), locate the second string at (*x* + **CMSR_text_width** (*string*), *y*).

CMSR_text_actual_extents

Returns the text "ink" bounding box of a specified string in the current font.

SYNTAX

C Syntax

```
#include <cm/display.h>

int *
  CMSR_text_actual_extents (string, extents_array)

char  *string;
int   extents_array[4];
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_TEXT_EXTENTS (string, extents_array)

CHARACTER*(*)  string(*)
INTEGER        extents_array(4)
```

Lisp Syntax

```
CMSR:draw-text-extents (string, &optional extents–array)
```

ARGUMENTS

string — The string for which the extents are to be determined.

extents_array — The left, right, bottom, and top extents of the text bounding box of *string*.

To read these values, pass *extents_array* to the appropriate accessor routine:

- **CMSR_left_extent**
- **CMSR_right_extent**

- `CMSR_bottom_extent`
- `CMSR_top_extent`

DESCRIPTION

`CMSR_text_actual_extents` returns the extents of the text "ink" bounding box, referenced to the left edge of the baseline. The "ink" bounding box defines the actual extents of the characters not including any character or line spacing. In a graphics display this means the furthest pixel from the text origin that is turned on (foreground). The actual character extents are useful when you need to position a text string exactly, for example, to place a label in an image, or to adjust to a limited space by reducing the standard interline spacing.

NOTE

The actual extent is not computed for the Generic Display fonts **`Think_Label`** and **`Think_Title`**. When using these fonts **`CMSR_text_actual_extents`** returns the logical extent of the string and a warning will be generated:

```
Warning: Built-in fonts not tested for actual
extents.
```

All measurements are in pixels from the left edge of the baseline:

Values are returned in four directions

- *Left* is the distance, in pixels, from the left edge of the baseline to the left edge of the first character. This measurement is the same as the *lbearing* of the left-most character.
- *Right* is the distance, in pixels, from the left edge of the baseline to the right edge of the right-most character in the string.
- *Bottom* is the number of pixels from the baseline to the bottom of the character that extends furthest below the line. This corresponds to the maximum descent of the enclosed character string.
- *Top* is the number of pixels from the baseline to the top of the character that extends furthest above the line. This corresponds to the maximum ascent of the enclosed character string.

SEE ALSO

CMSR_text_logical_extents(*string, extents_array*)

CMSR_font_extents(*extents_array*)

CMSR_text_logical_extents

Returns the logical text bounding box of a specified string in the current font.

SYNTAX

C Syntax

```
#include <cm/display.h>

int *
  CMSR_text_logical_extents (string, extents_array);

char  *string;
int   extents_array[4];
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_TEXT_EXTENTS (string, extents_array)

CHARACTER*(*)  string(*)
INTEGER        extents_array(4)
```

Lisp Syntax

```
CMSR:draw-text-extents (string, &optional extents-array)
```

ARGUMENTS

string — The string for which the extents are to be determined.

extents_array — The left, right, bottom, and top extents of the logical text bounding box of *string*.

To read these values, pass *extents_array* to the appropriate accessor routine:

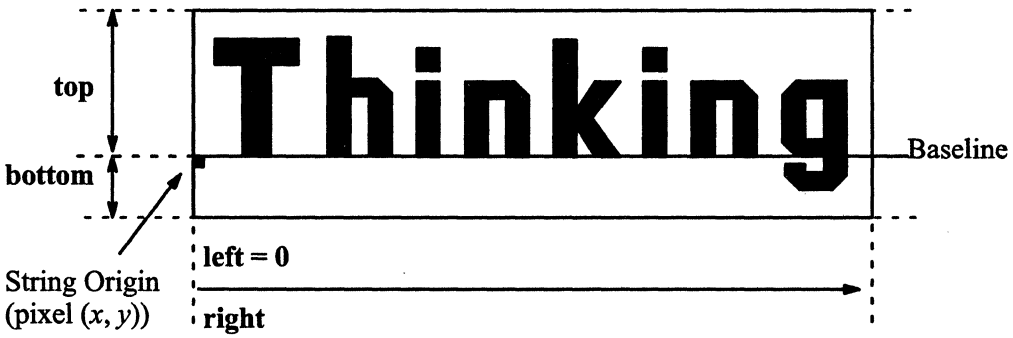
- `CMSR_left_extent`
- `CMSR_right_extent`

- `CMSR_bottom_extent`
- `CMSR_top_extent`

DESCRIPTION

`CMSR_text_actual_extents` returns the extents of the text logical bounding box, referenced to the left edge of the baseline. The logical text bounding box defines the extents of the string including the full character width and the font ascent and descent. Use these extents to position pieces of text next to one another. For example, the right extent can be used to determine the baseline *x* coordinate for the next piece of text to the right.

All measurements are in pixels from the left edge of the baseline:



Values are returned in four directions

- *Left*: Since the left edge of the logical bounding box is positioned at the left edge of the string's baseline, the logical left extent is always 0.
- *Right* is the distance, in pixels, from the left edge of the baseline to the right edge of the width of the right-most character in the string. Unlike the actual extent, this distance includes the character spacing.
- *Bottom* is the logical extent of the font below the baseline in pixels. This distance is the vertical space defined for the font to allow for character descenders and line spacing between strings.
- *Top* is the logical extent of the font above the baseline in pixels. This distance is the vertical space defined for the font to allow for height of the characters and line spacing between strings. Note that this is the distance set for the current font and is independent of the particular string specified.

SEE ALSO

CMSR_font_extents(*extents_array*)

CMSR_text_actual_extents(*string, extents_array*)

CMSR_font_extents

Returns the maximum character extents in the current font.

SYNTAX

C Syntax

```
#include <cm/display.h>

int *
  CMSR_font_extents (extents_array);

int   extents_array[4];
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_FONT_EXTENTS (extents_array)

INTEGER        extents_array(4)
```

Lisp Syntax

```
CMSR:font-extents (&optional extents-array)
```

ARGUMENTS

extents_array The maximum left, right, bottom, and top extents for any character in the current font.

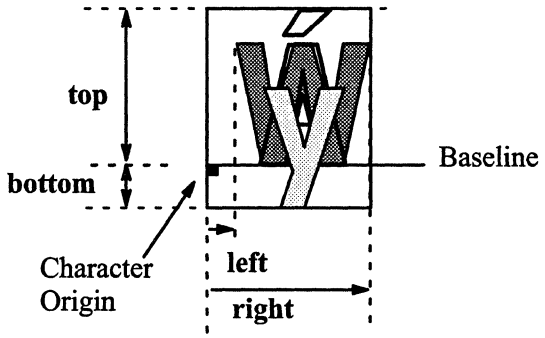
To read these values, pass *extents_array* to the appropriate accessor routine:

- `CMSR_left_extent`
- `CMSR_right_extent`
- `CMSR_bottom_extent`
- `CMSR_top_extent`

DESCRIPTION

CMSR_font_extents returns the maximum actual (ink) extent of any character in the font. Use these extents to determine spacing that can accommodate the largest characters in the font, not just the largest in the specified string.

All measurements are in pixels from the left edge of the baseline:



Values are returned in four directions

- *Left* is the distance, in pixels, from the left edge of the baseline to the left edge of the character in the current font that extends furthest to the left. This measurement is the same as the *lbearing* of the widest character in the font.
- *Right* is the distance, in pixels, from the left edge of the baseline to the right edge of the character in the current font that extends furthest to the right. This measurement is the same as the *rbearing* of the widest character in the font.
- *Bottom* is the number of pixels from the baseline to the bottom of the character in the current font that extends furthest below the line. This corresponds to the maximum descent of any character in the current font.
- *Top* is the number of pixels from the baseline to the top of the character in the current font that extends furthest above the line. This corresponds to the maximum ascent of any character in the current font.

SEE ALSO

CMSR_text_actual_extents

CMSR_text_logical_extents

CMSR_bottom_extent

Returns the text bottom extent from *extents_array*.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_bottom_extent (extents_array)

int   extents_array[4];
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_BOTTOM_EXTENT (extents_array)

INTEGER  extents_array(4)
```

Lisp Syntax

```
CMSR:bottom-extent (extents–array)
```

ARGUMENTS

extents_array A structure containing the left, right, bottom, and top extents of *string*. This structure is created for a string by one of the following routines:

- `CMSR_text_actual_extents`
- `CMSR_text_logical_extents`
- `CMSR_font_extents`

DESCRIPTION

CMSR_bottom_extent accepts an *extents_array* structure created by **CMSR_text_actual_extents**, **CMSR_text_logical_extents**, or **CMSR_font_extents** and returns the text bottom extent. These routines determine the top, bottom, right, and left extents for a specified string in the current font and load them into the *extents_array*. The meaning of the bottom extent depends on which of these extent routines created the *extents_array* passed to **CMSR_bottom_extent**.

Actual Extent

If the *extents_array* was created by **CMSR_text_actual_extents**, **CMSR_bottom_extent** returns the number of pixels from the string's baseline to the bottom of the character in the *string* that extends furthest below the line. This corresponds to the maximum actual character descent of the string passed to **CMSR_text_actual_extents**.

Font Extent

If the *extents_array* was created by **CMSR_font_extents**, **CMSR_bottom_extent** returns the number of pixels from the text baseline to the bottom of the character in the current *font* that extends furthest below the line. This corresponds to the maximum actual descent of any character in the current font.

Logical Extent

If the *extents_array* was created by **CMSR_text_logical_extents**, **CMSR_bottom_extent** returns the number of pixels required below the text baseline for the longest descender in the font plus the interline spacing.

This value is independent of the specific string passed to **CMSR_text_logical_extents**.

SEE ALSO

CMSR_left_extent(*extents_array*)

CMSR_right_extent(*extents_array*)

CMSR_top_extent(*extents_array*)

CMSR_top_extent

Returns the text top extent from an *extents_array*.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
   CMSR_top_extent (extents_array);

int    extents_array[4];
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_TOP_EXTENT (extents_array)

INTEGER  extents_array(4)
```

Lisp Syntax

```
CMSR:top-extent (extents–array)
```

ARGUMENTS

extents_array A structure containing the left, right, bottom, and top extents of *string*. This structure is created for a string by one of the following routines:

- **CMSR_text_actual_extents**
- **CMSR_text_logical_extents**
- **CMSR_font_extents**

DESCRIPTION

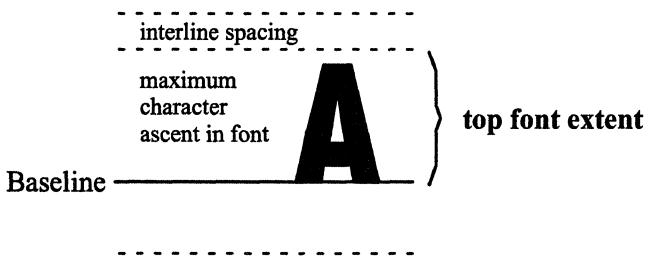
CMSR_top_extent accepts an *extents_array* structure created by **CMSR_text_actual_extents**, **CMSR_text_logical_extents**, or **CMSR_font_extents** and returns the text top extent. These routines determine the top, bottom, right, and left extents for a specified string in the current font and load them into the *extents_array*. The meaning of the top extent depends on which of these extent routines created the *extents_array* passed to **CMSR_bottom_extent**.

Actual Extent

If the *extents_array* was created by **CMSR_text_actual_extents**, **CMSR_top_extent** returns the number of pixels from the string's baseline to the top of the character in the *string* that extends furthest above the line. This corresponds to the maximum actual character ascent of the string passed to **CMSR_text_actual_extents**.

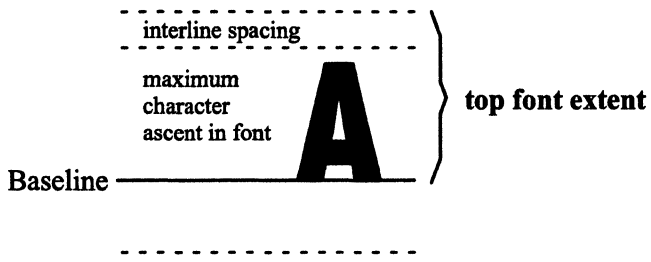
Font Extent

If the *extents_array* was created by **CMSR_font_extents**, **CMSR_top_extent** returns the number of pixels from the text baseline to the top of the character in the current *font* that extends furthest above the line. This corresponds to the maximum actual ascent of any character in the current font.



Logical Extent

If the *extents_array* was created by **CMSR_text_logical_extents**, **CMSR_top_extent** returns the number of pixels required above the text baseline for the highest character ascent in the font plus the interline spacing.



This value is independent of the specific string passed to **CMSR_text_logical_extents**.

SEE ALSO

CMSR_left_extent(*extents_array*)

CMSR_right_extent(*extents_array*)

CMSR_bottom_extent(*extents_array*)

CMSR_right_extent

Returns the text right extent from an *extents_array*.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
   CMSR_right_extent (extents_array);

int    extents_array[4];
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_RIGHT_EXTENT (extents_array)

INTEGER  extents_array(4)
```

Lisp Syntax

```
CMSR:right-extent (extents-array)
```

ARGUMENTS

extents_array — A structure containing the left, right, bottom, and top extents of *string*. This structure is created for a string by one of the following routines:

- **CMSR_text_actual_extents**
- **CMSR_text_logical_extents**
- **CMSR_font_extents**

DESCRIPTION

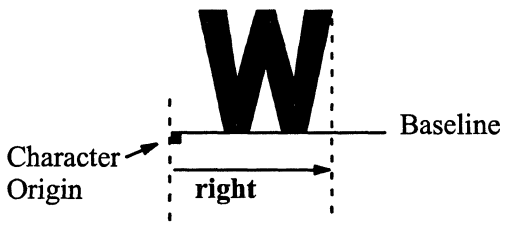
CMSR_right_extent accepts an *extents_array* structure created by **CMSR_text_actual_extents**, **CMSR_text_logical_extents**, or **CMSR_font_extents** and returns the text right extent. These routines determine the top, bottom, right, and left extents for a specified string in the current font and load them into the *extents_array*. The meaning of the right extent depends on which of these extent routines created the *extents_array* passed to **CMSR_bottom_extent**.

Actual Extent

If the *extents_array* was created by **CMSR_text_actual_extents**, **CMSR_right_extent** returns the number of pixels from the left edge of the string's baseline to the right edge of the right-most character in the *string*. This does not include any horizontal spacing beyond the last character in the string.

Font Extent

If the *extents_array* was created by **CMSR_font_extents**, **CMSR_right_extent** returns the distance, in pixels, from the character origin to the right edge of the character in the current font that extends furthest to the right. This measurement is the same as the *rbearing* of the widest character in the font.



Logical Extent

If the *extents_array* was created by **CMSR_text_logical_extents**, **CMSR_right_extent** returns the number of pixels from the left edge of the string's baseline to the right edge of the string bounding box. This is the distance to the right edge of the right-most character in the *string* plus the character's horizontal spacing.

SEE ALSO

CMSR_left_extent(*extents_array*)

CMSR_bottom_extent(*extents_array*)

CMSR_top_extent(*extents_array*)

CMSR_left_extent

Returns the text left extent from an *extents_array*.

SYNTAX

C Syntax

```
#include <cm/display.h>

int
  CMSR_left_extent (extents_array);

int   extents_array[4];
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_LEFT_EXTENT (extents_array)

INTEGER  extents_array(4)
```

Lisp Syntax

```
CMSR:left-extent (extents–array)
```

ARGUMENTS

extents_array A structure containing the left, right, bottom, and top extents of *string*. This structure is created for a string by one of the following routines:

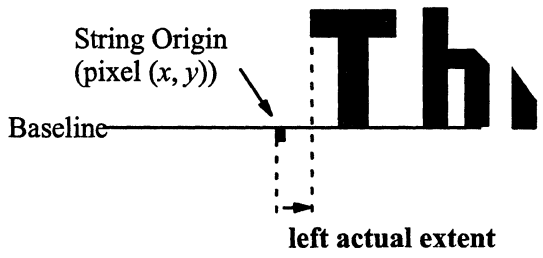
- `CMSR_text_actual_extents`
- `CMSR_text_logical_extents`
- `CMSR_font_extents`

DESCRIPTION

CMSR_left_extent accepts an *extents_array* structure created by **CMSR_text_actual_extents**, **CMSR_text_logical_extents**, or **CMSR_font_extents** and returns the text left extent. These routines determine the top, bottom, right, and left extents for a specified string in the current font and load them into the *extents_array*. The meaning of the left extent depends on which of these extent routines created the *extents_array* passed to **CMSR_bottom_extent**.

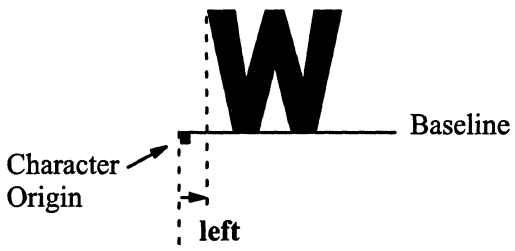
Actual Extent

If the *extents_array* was created by **CMSR_text_actual_extents**, **CMSR_left_extent** returns the number of pixels from the left edge of the string's baseline to the left edge of the left-most character in the *string*. This measurement is the same as the *lbearing* of the left-most character, that is, the number of pixels allocated for the left spacing of this character in the current font.



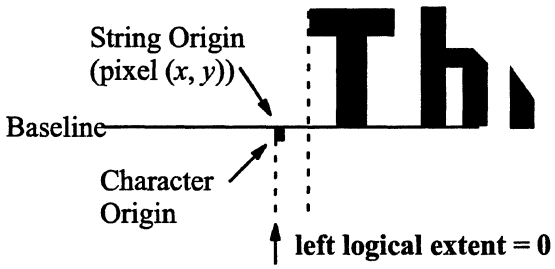
Font Extent

If the *extents_array* was created by **CMSR_font_extents**, **CMSR_left_extent** returns the number of pixels from the left edge of the character origin to the left edge of the character in the current font that extends furthest to the left. This measurement is the same as the *lbearing* of the widest character in the font.



Logical Extent

If the *extents_array* was created by **CMSR_text_logical_extents**, **CMSR_left_extent** always returns 0. Since the character origin of the left-most character of the string determines the left edge of the string's baseline, the logical left extent is always 0.



SEE ALSO

CMSR_right_extent(*extents_array*)

CMSR_bottom_extent(*extents_array*)

CMSR_top_extent(*extents_array*)

Chapter 4

Mouse Interface Routines

The Generic Display mouse interface is a mouse interaction system that is independent of the display type and the mouse host system type. It is based on the Generic Display Interface for the display operations and the X Window System mouse functions for the mouse interaction.

Two levels of interface are provided: a high-level interface, which provides automatic mouse tracking and higher-level selection routines; and a low-level interface, which allows the application complete control over the cursor and the mouse state.

4.1 Overview

4.1.1 Selecting a Generic Display Workstation and Display

Before using the mouse interface routines described in this section, you must create and select a Generic Display workstation and display. The easiest way to do this is by calling **`CMSR_select_workstation_menu`**. For detailed information on this and the other Generic Display workstation and display routines, see Chapter 2 of this manual.

The generic display can be either an X11 window or a CM framebuffer. The currently selected display is the display space for the Generic Display routines that read and write images.

The Generic Display workstation must be an X11 server. If you select an X11 server as the display, it will also be used as the workstation. The currently selected workstation provides X11 resources and the physical mouse to support the Generic Display system's text and mouse routines. The Generic Display mouse routines use the current workstation's resources to handle cursor tracking and interaction.

4.1.2 High-Level Mouse Routines

Three Generic Display routines supply the basic functionality of the mouse interface at a high level:

- **CMSR_get_mouse_point**
- **CMSR_get_mouse_line**
- **CMSR_get_mouse_rectangle**

Each of these routines automatically *grabs* the workstation mouse and tracks the mouse with a cursor on the currently selected display. When a button event is triggered on the mouse, these routines return a Generic Display **CMSR_mouse_point_t** data structure containing information on the cursor location at the time of the event and the button that caused it, and then release the workstation mouse.

NOTE: You must allocate the point data structure by calling **CMSR_allocate_mouse_point**.

When the workstation mouse is grabbed by one of these routines, the X11 cursor on the workstation screen disappears and a Generic Display Interface cursor appears on the currently selected display. This cursor is now controlled by the workstation mouse and mouse events are related to the current generic display. The user can use the mouse to move the cursor on the display and signal input by pressing and releasing the mouse buttons.

Note that if the current generic display is an X11 window

- the cursor is confined to that window until the Generic Display Interface releases the mouse.
- an error is generated if the window is iconified when you attempt to grab the mouse.

CMSR_get_mouse_point returns the location of the cursor when the user presses a mouse button and identifies the button. **CMSR_get_mouse_line** allows the user to define two points, the start and end points of a line, and returns the button information. Similarly, **CMSR_get_mouse_rectangle** allows the user to define a rectangle, and returns the button information. During their operations **CMSR_get_mouse_line** and **CMSR_get_mouse_rectangle** draw rubber band lines that follow the cursor to help the user position the line or rectangle in the display.

Your application can then read the coordinate and button information in the **CMSR_mouse_point_t** data structure with a set of accessor routines:

- **CMSR_mouse_point_x**

- `CMSR_mouse_point_y`
- `CMSR_mouse_point_buttons`
- `CMSR_mouse_point_pressed`
- `CMSR_mouse_point_released`

Routines are also provided that allow you to allocate, deallocate, and set the initial cursor position in a mouse structure:

- `CMSR_allocate_mouse_point`
- `CMSR_deallocate_mouse_point`
- `CMSR_set_mouse_point_location`

4.1.3 Low-Level Mouse Routines

A set of lower-level mouse interaction routines is also provided to give you greater control over the interaction between the mouse and display cursor:

- `CMSR_grab_mouse`
- `CMSR_release_mouse`
- `CMSR_current_mouse_point`
- `CMSR_current_mouse_delta`
- `CMSR_track_mouse`

Should your application require it, these routines allow you to explicitly manage grabbing, tracking, and releasing the mouse. For example, these routines make it possible for your application to respond to a series of button or motion events without releasing the mouse after each one.

4.1.4 Cursor Routines

Finally, a set of routines is provided to define the appearance and behavior of the generic display cursor and to return information about it:

- `CMSR_move_cursor`
- `CMSR_set_cursor_visibility`
- `CMSR_set_mouse_motion_threshold`

- **CMSR_set_mouse_motion_multiple**
- **CMSR_set_cursor_bitmap**
- **CMSR_set_cursor_named**
- **CMSR_closest_cursor_size**
- **CMSR_cursor_width**
- **CMSR_cursor_height**
- **CMSR_cursor_hot_x**
- **CMSR_cursor_hot_y**
- **CMSR_cursor_x**
- **CMSR_cursor_y**

The rest of this chapter contains detailed descriptions of these routines.

4.2 Point and Area Selection Routines

These routines are the highest-level interface to the Generic Display mouse support. For most applications you will need only these routines:

CMSR_get_mouse_point

Returns location of generic display cursor when button is pressed.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

CMSR_mouse_point_t
  CMSR_get_mouse_point (point);

CMSR_mouse_point_t point;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_GET_MOUSE_POINT (point)

INTEGER  point
```

Lisp Syntax

```
CMSR:get-mouse-point (&optional point)
```

ARGUMENTS

point

A **CMSR_mouse_point_t** structure representing the state of the mouse It includes both the mouse's location and which buttons were pressed.

NOTE: You must allocate *point* by calling **CMSR_allocate_mouse_point**.

In Fortran, *point* is returned as a integer. This integer may be passed to the Generic Display routines, described below, that access point information.

DESCRIPTION

CMSR_get_mouse_point grabs the currently selected workstation's mouse (if it is not already grabbed) and causes the mouse to track a cursor on the currently selected display until a button is pressed. When a button is pressed **CMSR_get_mouse_point** returns a **CMSR_mouse_point_t** data structure containing the location of the display cursor and the button that was pressed.

The application can access the information in *point* through the Generic Display mouse point routines:

- **CMSR_mouse_point_x**
- **CMSR_mouse_point_y**
- **CMSR_mouse_point_buttons**
- **CMSR_mouse_point_pressed**
- **CMSR_mouse_point_released**
- **CMSR_mouse_point_timestamp**

When the workstation mouse is grabbed, the cursor disappears from the currently selected Generic Display workstation and appears on the currently selected generic display.

If the workstation and display are an X11 server, the cursor is confined to the generic display window on the workstation screen until a button is pressed. The coordinates returned in *point* are the physical location of the cursor relative to that window, *not* to the screen as a whole. The coordinates do not reflect any Generic Display image offsets that may be set.

If the display is a CM framebuffer, the cursor is removed from the workstation screen and displayed on the framebuffer monitor until a button is pressed. The coordinates returned in *point* are the physical location of the cursor relative to the monitor screen and do not reflect any Generic Display image offsets that may be set.

CMSR_get_mouse_point returns the button pressed in *point* as one of the X11 button constants **Button1** through **Button5**.

You may read the button returned with **CMSR_mouse_point_pressed**.

CMSR_get_mouse_point works whether or not the mouse has been explicitly grabbed with **CMSR_grab_mouse**.

ERRORS

This routine signals an error if there is no selected workstation or selected display, or if you change the selected workstation or display while the cursor is grabbed.

If the display is an X11 window and the window is iconified when **CMSR_get_mouse_point** attempts to grab the mouse, the following error message is generated:

```
Warning: Unexpected grab status: 3. Pointer not grabbed.
```

CMSR_get_mouse_line
CMSR_get_mouse_rectangle

Returns points defining a line (rectangle) set by user with Generic Display cursor.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

void
  CMSR_get_mouse_line (p1, p2, anchorp);

CMSR_mouse_point_t   p1, p2;
int                  anchorp;

void
  CMSR_get_mouse_rectangle (p1, p2, anchorp);

CMSR_mouse_point_t   p1, p2;
int                  anchorp;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_GET_MOUSE_LINE (p1, p2, anchorp)

INTEGER  p1, p2
LOGICAL  anchorp

SUBROUTINE CMSR_GET_MOUSE_RECTANGLE (p1, p2, anchorp)
INTEGER  p1, p2
LOGICAL  anchorp
```

Lisp Syntax

```
CMSR:get-mouse-line (p1, p2, &optional anchorp)
CMSR:get-mouse-rectangle (p1, p2, &optional anchorp)
```

ARGUMENTS

p1

A **CMSR_mouse_point_t** structure containing the coordinates of the start point of the line or rectangle and the state of the mouse buttons when the point was defined.

NOTE: You must allocate *p1* by calling **CMSR_allocate_mouse_point**. In Fortran, the point structure is returned as a integer. This integer may be passed to the Generic Display routines, described below, that access point information.

p2

A **CMSR_mouse_point_t** structure containing the coordinates of the end point of the line or rectangle and the state of the mouse buttons when the point was defined.

NOTE: You must allocate *p2* by calling **CMSR_allocate_mouse_point**. In Fortran, the point structure is returned as a integer. This integer may be passed to the Generic Display routines, described below, that access point information.

anchorp

A predicate specifying how the anchor point of the line or rectangle is to be defined.

If *anchorp* is zero (nil in lisp), the start point is the cursor location when a mouse button is first pressed after the routine is called, and the end point is the cursor location when the button is released.

If *anchorp* is nonzero (non-nil in lisp), the start point is defined by the current contents of *p1* when the routine is called, and the end point is the location of the cursor at the first button press.

DESCRIPTION

CMSR_get_mouse_line and **CMSR_get_mouse_rectangle** return two points defining a line or rectangle, respectively, which have been set by the user with the Generic Display cursor.

Both routines work whether or not the mouse has been explicitly grabbed by calling **CMSR_grab_mouse**.

CMSR_get_mouse_line grabs the currently selected workstation's mouse (if it is not already grabbed) and causes the mouse to track a cursor on the currently selected display until a line is defined. A rubber-band line is drawn to help you place the points and is undrawn when the routines return.

If *anchorp* is set, the start point of the line is defined by the contents of *p1* when the routine is called, and a rubber-band line is drawn between this point and the position of the cursor until a button is pressed. When a button is pressed, **CMSR_get_mouse_line** returns the anchor point in *p1,* and returns the last cursor location and the button that was pressed in *p2.*

You can use **CMSR_set_mouse_point_location** to set *p1* to an initial value.

If *anchorp* is not set, the start point of the line is not defined until the first button press after the routine is called. When a button is pressed the current position of the cursor becomes the start point of the line, and a rubber-band line is then drawn between this point and the changing position of the cursor until a button is released. When a button is released, **CMSR_get_mouse_line** returns the start point and the button that was pressed in *p1,* and the last cursor location and the button that was released in *p2.*

Similarly, **CMSR_get_mouse_rectangle** grabs the currently selected workstation's mouse (if it is not already grabbed) and causes the mouse to track a cursor on the currently selected display until a rectangle is defined.

If *anchorp* is set, the start point of the rectangle is defined by the contents of *p1* when the routine is called and rubber-band lines are drawn defining a rectangle between this point and the position of the cursor until a button is pressed. When a button is pressed, **CMSR_get_mouse_rectangle** returns the anchor point in *p1,* and returns the last cursor location and the button that was pressed in *p2.*

If *anchorp* is not set, the start point of the rectangle is not defined until the first button press after the routine is called. When a button is pressed the current position of the cursor becomes the start point of the rectangle, and rubber-band lines defining a rectangle are then drawn between this point and the changing position of the cursor until a button is released. When a button is released, **CMSR_get_mouse_rectangle** returns the start point and the button that was pressed in *p1,* and the last cursor location and the button that was released in *p2.*

If **CMSR_get_mouse_line** or **CMSR_get_mouse_rectangle** is interrupted in Lisp, the mouse is automatically ungrabbed for the duration of the interrupt, and any rubber-band lines are undrawn.

The application can access the information in *p1* and *p2* through the Generic Display mouse point routines:

- **CMSR_mouse_point_x**
- **CMSR_mouse_point_y**
- **CMSR_mouse_point_buttons**

- **CMSR_mouse_point_pressed**
- **CMSR_mouse_point_released**
- **CMSR_mouse_point_timestamp**

When the workstation mouse is grabbed, the cursor disappears from the workstation screen and appears on the currently selected generic display. If the workstation and display are an X11 server, the cursor is confined to the generic display window on the workstation screen until the routine returns. If the display is a CM framebuffer, the cursor is removed from the workstation screen and displayed on the framebuffer monitor until the routine returns.

The coordinates returned in *p1* and *p2* are the physical location of the cursor relative to the display window or framebuffer monitor screen. These coordinates do not reflect any Generic Display image offsets that may be set.

The button pressed or released is returned in *p1* or *p2* as one of the X11 button constants **Button1** through **Button5**.

ERRORS

This routine signals an error if there is no selected workstation or selected display, or if you change the selected workstation or display while the cursor is grabbed.

If the display is an X11 window and the window is iconified when **CMSR_get_mouse_line** or **CMSR_get_mouse_rectangle** attempts to grab the mouse, the following error message is generated:

```
Warning: Unexpected grab status: 3. Pointer not grabbed.
```

CMSR_mouse_pan_and_zoom

Uses mouse to interactively pan and zoom CM framebuffer Generic Display.

SYNTAX

C Syntax

```
#include <cm/display.h>
void
  CMSR_mouse_pan_and_zoom ();
```

Fortran Syntax

```
INCLUDE include '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_MOUSE_PAN_AND_ZOOM ()
```

Lisp Syntax

```
CMSR:mouse-pan-and-zoom ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_mouse_pan_and_zoom allows the user to interactively pan and zoom a CM framebuffer. The currently selected Generic Display workstation must have a CM framebuffer as its selected display.

When **CMSR_mouse_pan_and_zoom** is called, the routine grabs the mouse and establishes the following operations:

- If no mouse buttons are pressed, moving the mouse left, right, up, or down will pan over the image on the framebuffer in the same direction.
- If you press the left mouse button, moving the mouse away from you will increase the framebuffer zoom, enlarging the image; moving the mouse towards you will decrease the framebuffer zoom.

- If you press the middle mouse button, the framebuffer returns to the state it was in before **CMSR_mouse_pan_and_zoom** was called.
- If you press the right mouse button, the routine returns, leaving the framebuffer with the current pan and zoom settings.

ERRORS

If you call **CMSR_mouse_pan_and_zoom** when the currently selected display is not a CM framebuffer, it will print an error message and exit.

If the display is an X11 window and the window is iconified when **CMSR_mouse_pan_and_zoom** attempts to grab the mouse, the following error message is generated:

```
Warning: Unexpected grab status: 3. Pointer not grabbed.
```

4.3 Mouse Points

These routines allocate, deallocate, and return information from a Generic Display mouse point data structure. The mouse point data structure is used to return information about mouse events to the the Generic Display Interface. You can use this information to make your application respond to events on the current generic display.

CMSR_allocate_mouse_point
CMSR_deallocate_mouse_point

Allocates (deallocates) `CMSR_mouse_point_t` data structure.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

CMSR_mouse_point_t
 CMSR_allocate_mouse_point ();
void
 CMSR_deallocate_mouse_point (point);
CMSR_mouse_point_t point;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_ALLOCATE_MOUSE_POINT ()
SUBROUTINE CMSR_DEALLOCATE_MOUSE_POINT (point)
INTEGER point
```

Lisp Syntax

```
CMSR:allocate-mouse-point ()
CMSR:deallocate-mouse-point (point)
```

ARGUMENTS

point	A `CMSR_mouse_point_t` structure representing the state of the mouse. It includes both the mouse's location and the buttons that were pressed.

DESCRIPTION

CMSR_allocate_mouse_point returns a **CMSR_mouse_point_t** point structure. This structure contains mouse point coordinates, mouse button event information, and a timestamp.

CMSR_deallocate_mouse_point frees all memory associated with the mouse point structure.

The mouse point structure is used to return information about the Generic Display cursor by:

- **CMSR_get_mouse_point**
- **CMSR_get_mouse_line**
- **CMSR_get_mouse_rectangle**
- **CMSR_current_mouse_point**
- **CMSR_current_mouse_delta**
- **CMSR_track_mouse**

You can read information from a **CMSR_mouse_point_t** structure with the Generic Display mouse point routines:

- **CMSR_mouse_point_x**
- **CMSR_mouse_point_y**
- **CMSR_mouse_point_buttons**
- **CMSR_mouse_point_pressed**
- **CMSR_mouse_point_released**
- **CMSR_mouse_point_timestamp**

CMSR_set_mouse_point_location

Sets the mouse point coordinates in the specified *point* data structure.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
void
  CMSR_set_mouse_point_location (point, x, y)
CMSR_mouse_point_t point;
int x;
int y;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_SET_MOUSE_POINT_LOCATION (point, x, y)
INTEGER point
INTEGER x
INTEGER y
```

Lisp Syntax

```
CMSR:set-mouse-point-location (point x y)
```

ARGUMENTS

point	A **CMSR_mouse_point_t** structure representing the state of the mouse. It includes both the mouse's location and the buttons that were pressed. **NOTE:** You must allocate *point* by calling **CMSR_allocate_mouse_point**.
x, y	The coordinates of the mouse location to be set in the *point* structure; *x* and *y* are referenced to the physical display window or CM framebuffer screen, and ignore any Generic Display image offset that may be set.

DESCRIPTION

CMSR_set_mouse_point_location sets the mouse point coordinates in the *point* structure.

CMSR_set_mouse_point_location does not change the location of the cursor directly; these coordinates take effect only when another routine references the *point* structure.

CMSR_mouse_point_x
CMSR_mouse_point_y

Returns the current mouse point *x* (*y*) coordinate from the specified *point* data structure.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
int
 CMSR_mouse_point_x (point)
CMSR_mouse_point_t point;

int
 CMSR_mouse_point_y (point)
CMSR_mouse_point_t point;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_MOUSE_POINT_X (point)
INTEGER  point

INTEGER FUNCTION CMSR_MOUSE_POINT_Y (point)
INTEGER  point
```

Lisp Syntax

```
CMSR:mouse-point-x (point)
CMSR:mouse-point-y (point)
```

ARGUMENTS

point — A **CMSR_mouse_point_t** structure representing the state of the mouse. Upon return from one of the routines listed below, it contains both the mouse's location and the buttons that were pressed.

DESCRIPTION

CMSR_mouse_point_x returns the current mouse point *x* coordinate from *point.*

CMSR_mouse_point_y returns the current mouse point *y* coordinate from *point.*

The mouse point structure describing the state of a mouse is returned by one of the following routines:

- **CMSR_get_mouse_point**
- **CMSR_get_mouse_line**
- **CMSR_get_mouse_rectangle**
- **CMSR_current_mouse_point**
- **CMSR_current_mouse_delta**
- **CMSR_track_mouse**

The coordinates, *x* and *y*, are the current cursor location when the routine that returned *point* was called, not the current location when **CMSR_mouse_point_x** or **CMSR_mouse_point_y** is called. The coordinates are relative to the physical generic display window or CM framebuffer monitor screen, and ignore any Generic Display image offset that may be set.

CMSR_mouse_point_pressed
CMSR_mouse_point_released

Returns the button constant from the specified *point* data structure for the last button pressed (released).

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
int
  CMSR_mouse_point_pressed (point)
CMSR_mouse_point_t point;

int
  CMSR_mouse_point_released (point)
CMSR_mouse_point_t point;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_MOUSE_POINT_PRESSED   (point)
INTEGER  point

INTEGER FUNCTION CMSR_MOUSE_POINT_RELEASED  (point)
INTEGER  point
```

Lisp Syntax

```
CMSR:mouse-point-pressed (point)
CMSR:mouse-point-released (point)
```

ARGUMENTS

point — A **CMSR_mouse_point_t** structure representing the state of the mouse. Upon return from one of the routines listed below, it contains both the mouse's location and the buttons that were pressed.

DESCRIPTION

CMSR_mouse_point_pressed and **CMSR_mouse_point_released** return the current button constant stored in the pressed or released field, respectively, of the *point* structure. If no constant is registered, these routines return NULL.

The mouse point structure describing the state of a mouse is returned by one of the following routines:

- **CMSR_get_mouse_point**
- **CMSR_get_mouse_line**
- **CMSR_get_mouse_rectangle**
- **CMSR_current_mouse_point**
- **CMSR_current_mouse_delta**
- **CMSR_track_mouse**

The structure contains a constant identifying the button that caused the latest event when the routine that returned *point* was called, not the latest event when **CMSR_mouse_point_pressed** or **CMSR_mouse_point_released** is called.

The button that caused the latest event appears in pressed or released, depending on its state.The constant is one of the X11 button constants **Button1** through **Button5**. In Lisp, the constants are **:button-1** through **:button-5**.

For example, calling **CMSR_mouse_point_released** with the *point* structure returned by **CMSR_get_mouse_rectangle**, returns the button constant representing the button that ended the operation, that is, the last button to go up, terminating the drag.

CMSR_mouse_point_buttons

Returns all the active buttons and modifiers from the specified *point* data structure.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
int
  CMSR_mouse_point_buttons (point)
CMSR_mouse_point_t point;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_MOUSE_POINT_BUTTONS (point)
INTEGER point
```

Lisp Syntax

```
CMSR:mouse-point-buttons (point)
```

ARGUMENTS

point	A **CMSR_mouse_point_t** structure representing the state of the mouse. It includes both the mouse's location and which buttons were pressed.

DESCRIPTION

CMSR_mouse_point_buttons returns an integer which has some bits set according to what modifiers (**ShiftMask**, **ControlMask**, **Mod1Mask**, etc.) and mouse buttons (**Button1Mask** through **Button5Mask**) were active when the the function that returned the *point* structure was called.

The mouse point structure describing the state of a mouse is returned by one of the following routines:

- **CMSR_get_mouse_point**
- **CMSR_get_mouse_line**
- **CMSR_get_mouse_rectangle**
- **CMSR_current_mouse_point**
- **CMSR_current_mouse_delta**
- **CMSR_track_mouse**

The significant event varies depending on the function. For example, **CMSR_get_mouse_point** sets the buttons according to the state when the button that activated the point selection was pressed. **CMSR_get_mouse_line** and **CMSR_get_mouse_rectangle** set the buttons of the two mouse point structures (*p1* and *p2*) to be the states of the buttons and modifiers at the beginning of the drag and at the end.

In C, the integer returned by **CMSR_mouse_point_buttons** is a bit-wise *or* of *Button1Mask* through *Button5Mask*, and *ShiftMask* and *ControlMask* and *LockMask* and *Mod1Mask* through *Mod5Mask*.

In Lisp, the masks are computed using **(xlib:make-state-mask &rest** *keys*), where *key* is one of *:button–1* through *:button–5* or *:shift* or *:lock* or *:control* or *:mod–1* through *:mod–5*.

To use these from Fortran, bitwise *or* the returned integer with the constants as defined in **display-cmf.h**:

```
integer ShiftMask
       parameter(ShiftMask=1)

integer LockMask
       parameter(LockMask=2)

integer ControlMask
       parameter(ControlMask=4)

integer Mod1Mask
       parameter(Mod1Mask=8)

integer Mod2Mask
       parameter(Mod2Mask=16)

integer Mod3Mask
       parameter(Mod3Mask=32)

integer Mod4Mask
       parameter(Mod4Mask=64)
```

```
integer Mod5Mask
       parameter(Mod5Mask=128)

integer Button1Mask
       parameter(Button1Mask=256)

integer Button2Mask
      parameter(Button2Mask=512)

integer Button3Mask
       parameter(Button3Mask=1024)

integer Button4Mask
       parameter(Button4Mask=2048)

integer Button5Mask
       parameter(Button5Mask=4096)

integer Button1
       parameter(Button1=1)

integer Button2
       parameter(Button2=2)

integer Button3
       parameter(Button3=3)

integer Button4
       parameter(Button4=4)

integer Button5
       parameter(Button5=5)
```

CMSR_mouse_point_timestamp

Returns the timestamp for the last mouse event in the specified *point* data structure

---

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
int
  CMSR_mouse_point_timestamp (point)
CMSR_mouse_point_t point;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_MOUSE_POINT_TIMESTAMP (point)
INTEGER point
```

Lisp Syntax

```
CMSR:mouse-point-timestamp (point)
```

ARGUMENTS

point	A **CMSR_mouse_point_t** structure representing the state of the mouse upon return from one of the routines listed below.

DESCRIPTION

CMSR_mouse_point_timestamp returns the X Window System's timestamp for the last mouse event recorded in the *point* structure.

The mouse point structure describing the state of a mouse is returned by one of the following routines:

- **CMSR_get_mouse_point**
- **CMSR_get_mouse_line**

- `CMSR_get_mouse_rectangle`
- `CMSR_current_mouse_point`
- `CMSR_current_mouse_delta`
- `CMSR_track_mouse`

The timestamp in *point* reflects the time of the last mouse event when the routine that returned *point* was called, not the time of the last event when `CMSR_mouse_point_timestamp` is called.

4.4 Grabbing Routines

These routines explicitly *grab* and release control of the mouse associated with the current generic workstation. When the mouse is grabbed, the mouse tracks a cursor on the current generic display and returns events associated with it. Releasing the mouse returns control to the physical workstation supporting the X11 server.

The cursor is automatically grabbed by the high-level routines **CMSR_get_mouse_point**, **CMSR_get_mouse_line**, **CMSR_get_mouse_rectangle**, and **CMSR_mouse_pan_and_zoom**. When using these routines you do not need to manage grabbing or releasing the mouse.

You must explicitly grab the mouse before using the low-level routines **CMSR_current_mouse_point**, **CMSR_current_mouse_delta**, and **CMSR_track_mouse**.

The grabbing routines described in this section are:

CMSR_grab_mouse
CMSR_release_mouse

Grabs (releases) the physical mouse of Generic Display workstation for display window.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
void
  CMSR_grab_mouse ()
void
  CMSR_release_mouse ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_GRAB_MOUSE ()
SUBROUTINE CMSR_RELEASE_MOUSE ()
```

Lisp Syntax

```
CMSR:grab-mouse ()
CMSR:release-mouse ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_grab_mouse causes the physical mouse associated with currently selected Generic Display workstation to control a cursor on the currently selected generic display. When **CMSR_grab_mouse** is called, the cursor disappears from the currently selected Generic Display workstation screen and appears on the currently selected generic display at the current display cursor location. The selected display window becomes the source window for mouse button or motion events and you can poll the events with the Generic Display mouse point routines described in Section 4.3.

CMSR_release_mouse returns control of the mouse to the workstation server. The cursor disappears from the display and reappears on the workstation, and control of the workstation cursor returns to the mouse.

CMSR_grab_mouse must be called before using the low-level Generic Display mouse routines **CMSR_current_mouse_point**, **CMSR_current_mouse_delta**, or **CMSR_track_mouse**.

However, the high-level mouse routines, **CMSR_get_mouse_point**, **CMSR_get_mouse_line**, and **CMSR_get_mouse_rectangle**, grab the mouse automatically while performing their operations on the display. These routines work whether or not the mouse is grabbed explicitly with **CMSR_grab_mouse**.

If the current display is a CM framebuffer when **CMSR_grab_mouse** is called, the cursor is removed from the current workstation screen and displayed on the framebuffer monitor until the mouse is released. While the mouse is grabbed, the application must control the display cursor by updating the cursor in relation to the workstation mouse with **CMSR_current_mouse_point**, **CMSR_current_mouse_delta**, or **CMSR_track_mouse**, or by explicitly positioning the display cursor with **CMSR_move_cursor**. The application must maintain this control until **CMSR_release_mouse** is called.

If the display is an X11 window when **CMSR_grab_mouse** is called, the cursor is confined to the generic display window until the mouse is released. The display cursor will automatically track the workstation mouse as usual; the application does not need to control the cursor position as on the CM framebuffer. However, since no harm is ever done by calling **CMSR_current_mouse_point**, it is recommended that it always be called in the application's inner loop regardless of display type.

Note to Lisp users: If **CMSR:grab-mouse** is interrupted in Lisp, the mouse is automatically ungrabbed for the duration of the interrupt.

ERRORS

It is an error to change the selected display or workstation while the mouse is grabbed.

If the display is an X11 window and the window is iconified when you call **CMSR_grab_mouse**, the following error message is generated:

```
Warning: Unexpected grab status: 3. Pointer not grabbed.
```

CMSR_mouse_grabbed_p

Returns the current status of the mouse interface.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
int
  CMSR_mouse_grabbed_p ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
LOGICAL FUNCTION CMSR_MOUSE_GRABBED_P ()
```

Lisp Syntax

```
CMSR:mouse-grabbed-p ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_mouse_grabbed_p returns the current status of the mouse interface.

If the currently selected workstation's mouse is grabbed, **CMSR_mouse_grabbed_p** returns true (.TRUE. in Fortran, non-NULL in C, non-nil in Lisp). If the mouse is not grabbed, **CMSR_mouse_grabbed_p** returns false (.FALSE. in Fortran, NULL in C, nil in Lisp). If there is no workstation selected, **CMSR_mouse_grabbed_p** always returns false.

The mouse is grabbed explicitly by **CMSR_grab_mouse** and internally by **CMSR_get_mouse_point**, **CMSR_get_mouse_line**, and **CMSR_get_mouse_rectangle** during their operations.

4.5 Low–Level Mouse Routines

These low-level mouse routines are called by the high-level routines to return information about the location of the mouse cursor. Ordinarily, you will not need to use them. However, if you want more explicit control of the mouse, you can use these routines to poll and track the mouse.

You must call **CMSR_grab_mouse** to explicitly grab the mouse before calling these routines.

CMSR_current_mouse_point

Updates generic display cursor and returns the mouse button state and the change in mouse coordinates since last update.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

CMSR_mouse_point_t
  CMSR_current_mouse_point (point, wait_p, compress_motion_p)

CMSR_mouse_point_t   point;
int                  wait_p, compress_motion_p;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CURRENT_MOUSE_POINT
&                                (point, wait_p, compress_motion_p)
INTEGER  point
LOGICAL  wait_p, compress_motion_p
```

Lisp Syntax

```
CMSR:current-mouse-point
                 (&optional point, (wait_p t), compress-motion-p)
```

ARGUMENTS

point

A **CMSR_mouse_point_t** structure representing the state of the mouse. It includes both the mouse's location and the buttons that were pressed.

The mouse point structure is allocated by calling **CMSR_allocate_mouse_point**. In Fortran, *point* is returned as a integer. This integer may be passed to the Generic Display routines, described below, that access point information.

wait_p A predicate specifying whether the routine should immediately return the current position and state of the mouse or wait until the next motion or button event.

compress_motion_p
A predicate specifying whether the routine should return the current position or the position at the first button event.

DESCRIPTION

CMSR_current_mouse_point updates the display cursor position and returns, in the passed structure, the current *x* and *y* coordinates of the mouse, along with its buttons.

NOTE: This routine works only when the mouse has been explicitly grabbed by calling **CMSR_grab_mouse**.

You can access the coordinate information in *point* through the Generic Display mouse point routines **CMSR_mouse_point_x** and **CMSR_mouse_point_y**.

If *wait_p* is true (.TRUE.in Fortran, non-NULL in C, non-nil in Lisp), then the routine waits until a mouse event (motion or button) occurs before returning. If *wait_p* is false (.FALSE. in Fortran, NULL in C, nil in Lisp), then the current state of the mouse is returned.

If *compress_motion_p* is true (.TRUE.in Fortran, non-NULL in C), the routine records a motion event only when the cursor starts or stops moving. If *compress_motion_p* is false (.FALSE. in Fortran, NULL in C), the routine records motion events continuously as the cursor is moved. Setting *compress_motion_p* to false is useful when it is important for the application not to lose any button presses.

If the routine returns because of a button press while *compress_motion_p* is false, the coordinates in the *point* structure will be the coordinates of the button press on the display.

If *wait_p* and *compress_motion_p* are both false, then the position at the earliest queued button press (if any) is returned, rather than reading to the end of the queue. This happens only if it is supported by the hardware.

ERRORS

It is an error to change the selected display or workstation while the mouse is grabbed.

CMSR_current_mouse_delta

Updates generic display cursor and returns the mouse button state and the change in mouse coordinates since last update.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

CMSR_mouse_point_t
  CMSR_current_mouse_delta (point, wait_p, compress_motion_p)

CMSR_mouse_point_t   point;
int                  wait_p, compress_motion_p;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CURRENT_MOUSE_DELTA
&                                (point, wait_p, compress_motion_p)

INTEGER  point
LOGICAL  wait_p, compress_motion_p
```

Lisp Syntax

```
CMSR:current-mouse-delta
                 (&optional point, (wait-p t), compress-motion-p)
```

ARGUMENTS

point

A **CMSR_mouse_point_t** structure representing the state of the mouse. It includes both the mouse's location and the buttons that were pressed.

The mouse point structure is allocated by calling **CMSR_allocate_mouse_point**. In Fortran, *point* is returned as a integer. This integer may be passed to the Generic Display routines, described below, that access point information.

wait_p A predicate specifying whether the routine should immediately return the current position and state of the mouse or wait until the next motion or button event.

compress_motion_p
A predicate specifying whether the routine should return the current position or the position at the first button event.

DESCRIPTION

CMSR_current_mouse_delta updates the display cursor and returns, in the passed structure, the delta *x* and *y* coordinates of the mouse since the last call to **CMSR_current_mouse_delta** or **CMSR_current_mouse_point**, along with the current mouse button state.

NOTE: This routine works only when the mouse has been explicitly grabbed by calling **CMSR_grab_mouse**.

You can read information from the mouse point structure with the Generic Display mouse point routines:

- **CMSR_mouse_point_x**
- **CMSR_mouse_point_y**
- **CMSR_mouse_point_buttons**
- **CMSR_mouse_point_pressed**
- **CMSR_mouse_point_released**

If *wait_p* is false (.FALSE. in Fortran, NULL in C, nil in Lisp), then the current state of the mouse is returned. If *wait_p* is true (.TRUE.in Fortran, non-NULL in C, non-nil in Lisp), then the routine waits until a mouse event (motion or button) occurs before returning.

If *compress_motion_p* is true (.TRUE.in Fortran, non-NULL in C), the routine records a motion event only when the cursor starts or stops moving. If *compress_motion_p* is false (.FALSE. in Fortran, NULL in C), the routine records motion events continuously as the cursor is moved. Setting *compress_motion_p* to false is useful when it is important for the application not to lose any button presses.

If the routine returns because of a button press (*compress_motion_p* = false), the coordinates in the point will be the coordinates of the button press on the display.

If *wait_p* and *compress_motion_p* are both false, then the position at the earliest queued button press (if any) is returned, rather than reading to the end of the queue. This happens only if it is supported by the hardware.

ERRORS

It is an error to change the selected display or workstation while the mouse is grabbed.

CMSR_track_mouse

Updates the Generic Display cursor and returns the current mouse points without removing button events from the event queue.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
CMSR_mouse_point_t
  CMSR_track_mouse (point)
CMSR_mouse_point_t  point;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_TRACK_MOUSE (point)
INTEGER  point
```

Lisp Syntax

```
CMSR:track-mouse (&optional point)
```

ARGUMENTS

point

A **CMSR_mouse_point_t** structure representing the state of the mouse. It includes both the mouse's location and the buttons that were pressed.

The mouse point structure is allocated by calling **CMSR_allocate_mouse_point**. In Fortran, *point* is returned as a integer. This integer may be passed to the Generic Display routines, described below, that access point information.

DESCRIPTION

CMSR_track_mouse updates the mouse pointer on the display and returns its current coordinates, just like **CMSR_current_mouse_point** or **CMSR_current_mouse_delta**. However it *never* removes any button events from the queue; if a button is pressed or released, that event will be skipped and will remain on the event queue to be read later by **CMSR_current_mouse_point** or **CMSR_current_mouse_delta** or one of the higher-level point and area selection routines or mouse point routines. This is useful if the application wants the mouse to track properly while it is computing, but it doesn't want to handle buttons until it is finished.

NOTE: This routine works only when the mouse has been explicitly grabbed by calling **CMSR_grab_mouse**.

You can read information from the mouse point structure with the Generic Display mouse point routines:

- **CMSR_mouse_point_x**
- **CMSR_mouse_point_y**
- **CMSR_mouse_point_buttons**
- **CMSR_mouse_point_pressed**
- **CMSR_mouse_point_released**

4.6 Cursor Routines

These routines control the appearance and behavior of the Generic Display Interface cursor. This cursor appears when the workstation mouse is grabbed explicitly by **CMSR_grab_mouse** or internally by **CMSR_get_mouse_point**, **CMSR_get_mouse_line**, and **CMSR_get_mouse_ rectangle** during their operations.

NOTE: The cursor is a property of the currently selected generic workstation. If you change the selected workstation, the cursor attributes change to the current setting for the new workstation.

CMSR_move_cursor

Moves the generic display cursor to specified display coordinates.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

void
  CMSR_move_cursor (x, y)

int x;
int y;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_MOVE_CURSOR (x, y)

INTEGER  x
INTEGER  y
```

Lisp Syntax

```
CMSR:move-cursor (x, y)
```

ARGUMENTS

x, y	The *x* and *y* coordinates of the point to which the cursor is to move. The coordinates are in pixels relative to the current display window or, if the current display is a CM framebuffer, to the framebuffer monitor screen. Any Generic Display image offsets that may be set are *not* applied to these coordinates.

DESCRIPTION

CMSR_move_cursor immediately moves the display cursor to the specified point on the display, as though the user had moved the mouse there.

CMSR_move_cursor only works when the mouse is grabbed. If the mouse is not grabbed, this routine merely updates the current cursor position in the Generic Display software; the visible display cursor is not affected. When the mouse is subsequently grabbed, by **CMSR_grab_mouse** or one of the high-level mouse operations **CMSR_get_mouse_point**, **CMSR_get_mouse_line**, or **CMSR_get_mouse_rectangle**, the cursor moves to the location specified.

CMSR_set_cursor_visibility

Makes the mouse cursor visible or invisible.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

void
  CMSR_set_cursor_visibility (visiblep)

int visiblep;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_CURSOR_VISIBILITY (visiblep)

LOGICAL  visiblep;
```

Lisp Syntax

```
CMSR:set-cursor-visibility (visiblep)
```

ARGUMENTS

visiblep	A predicate specifying whether or not the cursor is to be visible.

DESCRIPTION

CMSR_set_cursor_visibility causes the mouse cursor to become visible or invisible.

If *visiblep* is zero, the cursor becomes invisible, but the state of the mouse is not changed. This makes it possible to make the cursor invisible while keeping the mouse grabbed.

Setting *visiblep* to 1 causes the mouse cursor to become visible again after it was hidden. The cursor state is restored to whatever it was before the mouse was hidden, unless one of the **CMSR_set_cursor** routines is called during the hidden time.

NOTE: You do *not* need to explicitly manage cursor visibility with **CMSR_set_cursor_visibility** during normal use of the Generic Display mouse routines. The cursor automatically disappears from the workstation and appears on the display when the mouse is grabbed by **CMSR_grab_mouse** or one of the high-level mouse operations—**CMSR_get_mouse_point**, **CMSR_get_mouse_line**, or **CMSR_get_mouse_rectangle**. The cursor is also automatically returned to the workstation when the mouse is released.

CMSR_set_mouse_motion_threshold CMSR_mouse_motion_threshold

Sets (returns) the current motion threshold set for the generic display mouse.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
void
  CMSR_set_mouse_motion_threshold (threshold)
int threshold;

int
  CMSR_mouse_motion_threshold ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_SET_MOUSE_MOTION_THRESHOLD (threshold)
INTEGER threshold

INTEGER FUNCTION CMSR_MOUSE_MOTION_THRESHOLD ()
```

Lisp Syntax

```
CMSR:set-mouse-motion-threshold (threshold)
CMSR:mouse-motion-threshold ()
```

ARGUMENTS

threshold	The number of display screen pixels beyond which the cursor movement is to be accelerated.

DESCRIPTION

CMSR_set_mouse_motion_threshold sets the motion threshold, in pixels, above which the cursor's motion is accelerated.

When the mouse is moved further than *threshold* pixels, the cursor movements on the display are multiplied by the *acceleration* value set by **CSR_set_mouse_motion_multiple.**

This mouse motion allows you to move the cursor precisely over small distances, but to move large distances across the screen quickly.

CMSR_mouse_motion_threshold returns the current threshold.

CMSR_set_mouse_motion_multiple
CMSR_mouse_motion_multiple

Sets (returns) the acceleration factor for movement of the Generic Display cursor.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
void
  CMSR_set_mouse_motion_multiple (acceleration)

double acceleration;

double
  CMSR_mouse_motion_multiple ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
void
  CMSR_SET_MOUSE_MOTION_MULTIPLE (acceleration)
DOUBLE PRECISION acceleration

DOUBLE PRECISION FUNCTION MOUSE_MOTION_MULTIPLE ()
```

Lisp Syntax

```
CMSR:set-mouse-motion-multiple (acceleration)

CMSR:mouse-motion-multiple ()
```

ARGUMENTS

acceleration — The amount to accelerate the cursor's motion. The motion acceleration multiplies the x and y components of the physical mouse movement before applying them to the cursor motion on the display screen.

DESCRIPTION

CMSR_set_mouse_motion_multiple sets the acceleration factor to be applied to movements of the Generic Display workstation mouse.

When the mouse is moved further than the *threshold* value set by **CMSR_set_mouse_motion_threshold**, the cursor movements on the display are multiplied by *acceleration*.

This mouse motion allows you to move the cursor precisely over small distances, but to move large distances across the screen quickly.

CMSR_mouse_motion_multiple returns the acceleration multiple currently set.

CMSR_set_cursor_bitmap

Sets the appearance of the generic display cursor to the specified bitmap.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

void
  CMSR_set_cursor_bitmap
                    (bitmap, mask, width, height, hot_x, hot_y)
unsigned char  *bitmap, *mask;
unsigned int   width, height;
int            hot_x, hot_y;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

SUBROUTINE CMSR_SET_CURSOR_BITMAP
&                                (bitmap, mask, width, height, hot_x, hot_y)
CHARACTER*(*)  bitmap, mask;
INTEGER        width, height;
INTEGER        hot_x, hot_y;
```

Lisp Syntax

```
CMSR:set-cursor-bitmap(bitmap, mask,
                       &key width, height, (hot-x 0), (hot-y 0))
```

ARGUMENTS

bitmap — An array of unsigned characters. Each bit corresponds to a pixel in the cursor shape going from left to right and top to bottom (row-major order). The most significant bit of each character is the left-most pixel. Pixels corresponding to bits that are 1 in the bitmap are white, and pixels corresponding to 0 bits are black.

mask An array of unsigned characters. Each bit corresponds to a pixel in the cursor bitmap going from left to right and top to bottom (row-major order). The most significant bit of each character is the left-most pixel.

The mask is used to determine which parts of the cursor bitmap will be displayed. Pixels in the bitmap corresponding to bits that are 1 in the mask are displayed, and bitmap pixels corresponding to 0 bits in the mask are transparent, that is, they have no effect on the display.

width, height The width and height, in pixels, of the cursor.

hot_x, hot_y The coordinates of the cursor's "hotspot." The coordinates are specified relative to the upper left corner of the bitmap.

The cursor's hotspot is the active pixel in the cursor, which is reported as the location of the cursor.

DESCRIPTION

CMSR_set_cursor_bitmap sets the display cursor shape to the specified bitmap.

The routine signals an error if the cursor shape defined by *width* and *height* is not supported by the display hardware. You may determine whether a cursor size is supported by calling **CMSR_closest_cursor_size**. Cursors that are 16 x 16 pixels are always supported.

In Lisp, unspecified width and height default to the total number of pixels across and down in the bitmap array.

NOTE: The cursor is a property of the display, so if the display changes, the cursor will be changed to the default cursor for that display.

ERRORS

An error is signaled if the cursor shape is not supported by the display hardware.

SEE ALSO

CMSR_set_cursor_named (*name***)**

CMSR_set_cursor_named

Sets generic display cursor to a predefined shape.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
void
  CMSR_set_cursor_named (name)
char *name;
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
SUBROUTINE CMSR_SET_CURSOR_NAMED (name)
CHARACTER*(*) name
```

Lisp Syntax

```
CMSR:set-cursor-named (name)
```

ARGUMENTS

name	The name of the predefined shape to which the cursor is to be set. *name* is not sensitive to case.

DESCRIPTION

CMSR_set_cursor_named sets the display cursor to the desired predefined cursor shape.

The available shapes are:

- **none** an invisible cursor
- **arrow** a pointer cursor

- **cross** a cross
- **small-cross** a smaller cross shape
- **bullet** a small round dot
- **x** an x-shaped cursor

All cursor shapes are black with a white border, device color map permitting.

ERRORS

The routine signals an error if *name* is not the name of a defined cursor.

SEE ALSO

CMSR_set_cursor_bitmap

CMSR_closest_cursor_size

Returns the cursor size supported by the workstation hardware closest to the specified size.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
void
  CMSR_closest_cursor_size (size)
int size[2];
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
void
  CMSR_CLOSEST_CURSOR_SIZE (size)
INTEGER size(2)
```

Lisp Syntax

```
CMSR:set-cursor-named (size)
```

ARGUMENTS

size	An integer array containing the width and height, in pixels, of the desired cursor shape.

DESCRIPTION

CMSR_closest_cursor_size returns, in *size*, the hardware-supported cursor size closest to the values passed in in the array *size*.

A size of 16x16 is always supported.

4.7 Cursor Information

The following routines return information about the Generic Display cursor:

CMSR_cursor_width
CMSR_cursor_height

Returns the width (height), in pixels, of the current Generic Display cursor.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>
int
   CMSR_cursor_width ();
int
   CMSR_cursor_height ();
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'
INTEGER FUNCTION CMSR_CURSOR_WIDTH ()
INTEGER FUNCTION CMSR_CURSOR_HEIGHT ()
```

Lisp Syntax

```
CMSR:cursor-width ()
CMSR:cursor-height ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_cursor_width returns the width of the current cursor, in pixels.

CMSR_cursor_height returns the height of the current cursor, in pixels.

SEE ALSO

CMSR_closest_cursor_size

CMSR_cursor_hot_x
CMSR_cursor_hot_y

Returns the *x* (*y*) coordinate of the hotspot (selection point) of the current cursor.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

int
  CMSR_cursor_hot_x ()

int
  CMSR_cursor_hot_y ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CURSOR_HOT_X ()

INTEGER FUNCTION CMSR_CURSOR_HOT_Y ()
```

Lisp Syntax

```
CMSR:cursor-hot-x ()

CMSR:cursor-hot-y
```

ARGUMENTS

None.

DESCRIPTION

CMSR_cursor_hot_x returns the *x* coordinate of the hotspot (selection point) of the current cursor.

CMSR_cursor_hot_y returns the *y* coordinate of the hotspot (selection point) of the current cursor.

The cursor's hotspot is the active pixel in the cursor, which is reported as the location of the cursor. The coordinates are in pixels relative to the upper left corner of the cursor bitmap.

CMSR_cursor_x
CMSR_cursor_y

Returns the *x* (*y*) coordinate of the currently recorded cursor position.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

int
  CMSR_cursor_x ()

int
  CMSR_cursor_y ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

INTEGER FUNCTION CMSR_CURSOR_X ()

INTEGER FUNCTION CMSR_CURSOR_Y ()
```

Lisp Syntax

```
CMSR:cursor-x ()

CMSR:cursor-y ()
```

ARGUMENTS

None.

DESCRIPTION

`CMSR_cursor_x` returns the *x* coordinate of the current cursor position recorded.

`CMSR_cursor_y` returns the *y* coordinate of the current cursor position recorded.

NOTE: `CMSR_cursor_x` and `CMSR_cursor_y` do not return the current location of the cursor on the display screen, but the current cursor position recorded in the Generic Display software. That is, these routines do not track the mouse. To get an updated cursor location, call `CMSR_get_mouse_point`.

CMSR_cursor_visible_p

Indicates whether the generic display's cursor is currently visible or not.

SYNTAX

C Syntax

```
#include <cm/cmsr.h>

int
  CMSR_cursor_visible_p ()
```

Fortran Syntax

```
INCLUDE '/usr/include/cm/display-fort.h'

LOGICAL FUNCTION FSR_CURSOR_VISIBLE_P ()
```

Lisp Syntax

```
CMSR:cursor-visible-p ()
```

ARGUMENTS

None.

DESCRIPTION

CMSR_cursor_visible_p returns the visibility status of the current cursor.

If the cursor is visible, **CMSR_cursor_visible_p** returns true (.TRUE.in Fortran, non-NULL in C, non-nil in Lisp). If the cursor is not visible, **CMSR_cursor_visible_p** returns false (.FALSE. in Fortran, NULL in C, nil in Lisp).

If the cursor is not grabbed, i.e., there is no current cursor, **CMSR_cursor_visible_p** always returns false.

Alphabetical Index of Routines

This index lists all the Generic Display Interface routines alphabetically.

CMSR_a

CMSR_b

CMSR_c

CMSR_d

CMSR_f

CMSR_g

CMSR_l

CMSR_m

CMSR_r

CMSR_s

CMSR_t

CMSR_w

CMSR_x

Keyword Index of Routines

This index lists the Generic Display Interface routines sorted by the key words that appear in their names.

---

x

x_display

y

zoom