Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-90-06

The Window Tool Kit

Andreas Becker

Dezember 1990

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988 by the shareholder companies ADV/Orga, AEG, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Mannesmann-Kienzle, Philips, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

The Window Tool Kit

Andreas Becker

DFKI-D-90-06

The Window Tool Kit

Andreas Becker

Universität Kaiserslautern/

DFKI Kaiserslautern

December 6, 1990

Contents

1 Introduction to the Window Tool Kit

The Window Tool Kit allows you to create and access windows, viewports, and bitmaps, and to access the mouse. It supports both character output and graphic output, and it displays characters in a variety of fonts.

Actually there are two Implementations of the Window Tool Kit:

1. **The Allegro Window Tool Kit**
   It uses the Allegro Common Lisp Graphics of the MacIntosh Computer.

2. **The X Window Tool Kit**
   It is built upon the CLX (Common Lisp X Windows) - Interface of the MIT. It runs on all machines, which support CLX and Common Lisp. So together with KCL (Kyoto Common Lisp) we have a public domain window system (KCL and CLX are public domain software).

The package "WINDOWS" contains all Window Tool Kit functions.

Sometimes it is necessary to differ the Allegro Window Tool Kit from the X Window Tool Kit. For this purpose the features **:atoolkit** and **:xtoolkit** are defined. (See the section "Standard Dispatching Macro Character Syntax" in the Common Lisp Reference Manual for more information on features.)

1.1 Initializing the Window Tool Kit

To use the Window Tool Kit, you must first invoke Lisp. You must then initialize the Window Tool Kit. The function **initialize-windows** creates a system window that contains the root viewport. The top level of Lisp is still accessible from the window from which you invoked the function.

The syntax for **initialize-windows** is as follows:

initialize-windows **[Function]**

&key :height :width :screen-x :screen-y

See the function page at the end of this chapter for a detailed description of the keyword arguments.

1.2 Restoring Windows in a Saved Image

If the Window Tool Kit has been initialized and you want to make an image, please first delete the window environment (with function **leave-window-system**). To restore the state of the windows in the newly saved image, call initialize-windows again and rebuild your window environment.

1.3 Basic Structures

The Window Tool Kit has three basic data structures: **positions**, **extents**, and **regions**.

1.3.1 Positions

A position is a data structure with two fixnum components, **x** and **y**. A position is specified in Cartesian coordinates in which the **x** component is the distance to the right of the origin, and the **y** component is the distance below the origin.

The following operations are defined for positions:

make-position	*position-y*
position-x	*positionp*

1.3.2 Extents

An extent is a data structure that describes the size of a rectangular area. An extent has two nonnegative fixnum components called **width** and **height**.

The following operations are defined for extents:

extent-height	*extentp*
extent-width	*make-extent*

1.3.3 Regions

A region describes a rectangular area.

The origin of a region is its top-left point. The corner of a region is the point just below and to the right of its bottom-right point.

The Window Tool Kit includes functions for accessing the attributes of a region, for finding the corners of a region, for testing containment and equality for regions, and for testing whether a position is inside a region.

The following operations are defined for regions:

make-region	*region-size*
region-contains-point-p	*region-union*
region-contains-position-p	*region-width*
region-corner	*region/=*
region-corner-x	*region<*
region-corner-y	*region<=*
region-height	*region=*
region-intersection	*region>*
region-origin	*region>=*
region-origin-x	*regionp*
region-origin-y	

1.4 Bitmaps

In the Window Tool Kit, all graphic operations are performed either directly or indirectly on **bitmaps**. A bitmap is a rectangular array of bits.

Most bitmaps are created with the function **make-bitmap**. The Window Tool Kit also provides functions for accessing bitmap data structures, for setting the width or height of a bitmap and for copying bitmaps.

The following operations are defined for bitmaps:

bitblt	*clear-bitmap*
bitblt-position	*copy-bitmap*
bitblt-region	*draw-circle*
bitmap-extent	*draw-line*
bitmap-height	*draw-polyline*
bitmap-p	*draw-polypoint*
bitmap-value	*make-bitmap*
bitmap-width	*stringblt*
charblt	*with-fast-drawing-environment*

1.5 Fonts

A **font** is a set of character images and an associated font name. Each image specifies what the corresponding character looks like when displayed.

Each character's image in a font has a width. Some fonts are fixed width, which means that every character has the same width. Other fonts are variable width, which means, for example, that a "W" may be wider than an "i."

Some characters have an ascender, which is the part of the character above the baseline. Some characters, such as "j" and "q," also have a descender, which is the part of the character below the baseline. The height of a character is the combined height of its ascender and descender. The baseline height of a character is the height of just its ascender.

1.5.1 Operations on Fonts

The following operations are defined for fonts:

find-font	*font-name*
font-baseline	*fontp*
font-fixed-width	*string-width*
font-height	

1.6 Viewports

In the Window Tool Kit, a **viewport** is a mapping between a region of a bitmap and a region of the screen. The **bitmap clipping region** is the piece of a bitmap that a viewport views. The **screen clipping region** is the region of the screen onto which the viewport maps. The screen clipping region and the bitmap clipping region must be the same size.

Whether or not a viewport is actually displayed on the display screen depends on whether the viewport is activated and whether it is occluded (covered up) by other viewports.

The mapping between the bitmap and the screen is as follows: bits in the bitmap that are within the bitmap clipping region are mapped into the viewport's coordinate system by subtracting the origin of the clipping region. These bits are then mapped onto the screen by adding the origin of the viewport's screen clipping region. Bitmap positions that are outside of the bitmap clipping region are undefined under this mapping. Similarly, if a bitmap position maps onto a screen position that is occluded by another viewport, the screen position of the bit is undefined.

Whenever the mouse position lies on top of an unoccluded portion of some viewport, the inverse mapping carries it back to some point of that viewport's corresponding bitmap.

You can reshape viewports and move them around on the screen.

1.6.1 Creating a Viewport

The following function creates a viewport:

make-viewport

1.6.2 The Viewport Hierarchy

Viewports are arranged in a hierarchy that controls occlusion. The root of the hierarchy is the **root viewport**, which is created when the Window Tool Kit is initialized. The function **root-viewport** returns the root viewport. The root viewport is a viewport onto a special bitmap that requires less memory but has limited capabilities. You cannot modify the bits of this special bitmap in any way without signaling an error. The root viewport covers the entire screen. All other viewports occlude the root viewport.

Every viewport except the root viewport has a **parent viewport**. A viewport does not need to lie within its parent's region.

All viewports that are children of one viewport are called **sibling viewports**. They may overlap on the screen.

Sibling viewports are arranged in a stack. The function **viewport-children** returns a list of a viewport's children in the order that they appear in the sibling stack, with the sibling at the top of the stack appearing at the beginning of the list. The function **expose-viewport** moves a viewport to the top of its sibling stack. The function **hide-viewport** moves a viewport to the bottom of its sibling stack.

A viewport may be either active or inactive. A viewport is displayed on the screen only if it is active. A viewport that is inactive is still in the viewport hierarchy, but it is not displayed. If a viewport is inactive, none of its descendants are active.

If two active viewports overlap on the screen, the following rules determine which viewport occludes the other:

- A viewport occludes all of its ancestor viewports.
- If two viewports are siblings, then the viewport that is closest to the top of the sibling stack and all of its descendants occlude the viewport that is farther down and all of its descendants.

1.6.3 Accessing Viewport Data Structures

The following operations are defined for viewports:

activate-viewport	*viewport-bitmap*
deactivate-viewport	*viewport-bitmap-offset*
expose-viewport	*viewport-bitmap-region*
hide-viewport	*viewport-bitmap-x-offset*
move-viewport	*viewport-bitmap-y-offset*
reshape-viewport	*viewport-children*
root-viewport	*viewport-parent*
viewport-at-point	*viewport-screen-region*
viewport-at-position	*viewportp*

Any function that takes a bitmap argument can be passed a viewport argument. The function is then performed on the viewport's bitmap.

1.7 Bitmap Output Streams

Because both input and output in Common Lisp are stream oriented, the Window Tool Kit provides a stream-oriented interface to bitmaps, the **bitmap output stream**, which is an output stream that supports all the Common Lisp character output functions.

Each bitmap output stream maintains an output position that specifies the next available position for writing to the bitmap. You can modify this position.

Each bitmap output stream also maintains a current font and a current linefeed distance. The linefeed distance of a stream is initially the character height of the initial font. If the user does not specify a font, the value of the variable ***default-font*** becomes the initial font.

A bitmap output stream has a default operation for combining new bits with bits already in the bitmap. This operation can be any of the 16 boolean constants that can be the first argument to the function boole. **Note:** For the Allegro Window Tool Kit only the following operations are defined: **boole-1, boole-and, boole-andc1, boole-c1, boole-eqv, boole-ior, boole-orc1, boole-xor.**

The default value is the value of the constant **boole-xor**. (See the chapter "Numbers" in the Common Lisp Reference Manual for more information on **boole**.)

The following operations are defined for bitmap output streams:

bitmap-output-stream-p	*stream-linefeed-distance*
make-bitmap-output-stream	*stream-operation*
stream-current-font	*stream-position*
stream-draw-circle	*stream-x-position*
stream-draw-line	*stream-y-position*
stream-draw-polyline	

A bitmap output stream can be used as the stream argument in any of the following functions:

wclear-output	*wprin1*
wfinish-output	*wprint*
wforce-output	*wterpri*
wformat	*wwrite*
wfresh-line	*wwrite-char*
wpprint	*wwrite-string*
wprinc	

The syntax of these functions is similar to the syntax of the related Common Lisp Functions.

(See the chapter "Streams" in the Common Lisp Reference Manual for more information on Common Lisp streams.)

Any function that takes a bitmap argument can be passed a bitmap output stream. The operation is performed on the bitmap output stream's underlying bitmap.

1.7.1 Using the Mouse

The Window Tool Kit provides several ways in which the mouse can be accessed and used in programs. These include polling the mouse, queuing mouse events, and specifying active regions.

1.7.2 The Mouse Cursor

The position of the mouse is indicated on the screen by a mouse cursor. Mouse cursors are manipulated by using **mouse cursor objects**, which are specifications of mouse cursors.

The following functions access mouse cursor objects:

current-mouse-cursor	*mouse-cursor-p*

The function **current-mouse-cursor** returns the mouse cursor object that is currently tracking the mouse on the screen. The **setf** macro can be used with **current-mouse-cursor** to change the cursor.

In addition, the following form allows you to move the cursor:

move-mouse (Does not work for the Allegro Window Tool Kit.)

1.7.3 Polling the Mouse

The most basic way to access the mouse is **polling**-that is, having a program examine the current state of the mouse. The following functions provide information about the position of the mouse:

mouse-x	*mouse-y*

The functions mouse-x and mouse-y return the current x- and y-coordinates of the mouse, which are specified in terms of the root viewport. A mouse with two buttons has only a left button and right button, and the middle button is undefined. If you use the Allegro Window Tool Kit, you get the middle mouse button by pressing the control key and the mouse key together and the right button by pressing the alternate key and the mouse key together.

1.7.4 Handling Mouse Events

Queuing **mouse events** is a more versatile way to access the mouse.

A mouse event occurs when the mouse is moved or when one of its buttons is pressed or released. Mouse events recognized by the Window Tool Kit are the following:

:mouse-left-down	*:mouse-right-up*
:mouse-middle-down	*:mouse-enter-region*
:mouse-right-down	*:mouse-exit-region*
:mouse-left-up	*:mouse-move*
:mouse-middle-up	

The meaning of each mouse event is summarized below:

- :mouse-left-down

  :mouse-middle-down

  :mouse-right-down

  The corresponding button has been pressed.

- :mouse-left-up

  :mouse-middle-up

  :mouse-right-up

  The corresponding button has been released.

- :mouse-enter-region

  :mouse-exit-region

  The mouse has entered or exited an active region. (Active regions are discussed in the next subsection.)

- :mouse-moved

  The mouse has been moved.

If your mouse has only two buttons, the two mouse events :mouse-middle-down and :mouse-middle-up cannot occur.

A mouse event object is a special data structure that is used to encode mouse events. A mouse event object specifies what mouse event has occurred, where the mouse was when the event occurred, and which buttons were pushed at the time.

The following operations are defined for mouse event objects:

mouse-event-event-type	*mouse-event-y*
mouse-event-x	*mouse-event-p*

Special input streams called **mouse input streams** can queue both character input and mouse event objects. Characters typed at the terminal and mouse events are queued on a mouse input stream in the order in which they occur.

The following functions create and access mouse input streams:

make-mouse-input-stream
mouse-input-stream-p
mouse-input-stream-queue-mouse-events-p
mouse-input-stream-viewport

When a mouse event occurs, the value of the function **mouse-input** is examined; it must be a mouse input stream. You cannot read from a mouse input stream unless the stream is the value of **mouse-input**. If the value of the expression **(mouse-input-stream-queue-mouse-events-p (mouse-input))** is true, then a mouse event object encoding the mouse event is created and queued on the stream. Otherwise the mouse event is ignored, and no mouse event object is created. The **setf** macro can be used with the function **mouse-input** to modify the mouse input stream to which mouse input is sent.

Each mouse input stream is associated with a particular viewport. The values of the functions **mouse-event-x** and **mouse-event-y** for mouse event objects queued on a mouse input stream are relative to the origin of that viewport.

The following operations are defined for mouse input streams:

listen-any
read-any-no-hang
peek-any
unread-any
read-any

These five functions are similar to the Common Lisp functions listen, peek-char, read-char, read-char-no-hang, and unread-char respectively. (See the chapter "Input/Output" in the Common Lisp Reference Manual for more information.) They differ from their Common Lisp analogues in that they check the input stream for both mouse event objects and characters.

1.8 Active Regions

Specifying **active regions** is a third way to access the mouse. Active regions facilitate the creation of menus, scroll bars, and other display objects that interact with the mouse.

An active region is a region that can be attached to a bitmap and that causes that region of the bitmap to become **mouse sensitive.** If that region of the bitmap is displayed on the display screen and the mouse enters or leaves that region of the screen, the Window Tool Kit's mouse handler calls a method specified by the active region. Similarly, if a mouse event occurs while the mouse is inside an active region displayed on the screen, a method specified by the active region is called.

The following operations are defined for active regions:

active-region-bitmap	*bitmap-active-regions*
active-region-method	*clear-bitmap-active-regions*
active-region-p	*detach-active-region*
attach-active-recion	*make-active-region*

Any function that can be passed a region can be passed an active region instead.

When a mouse event occurs, the process handling the mouse determines which mouse methods, if any, are invoked. First, the viewport containing the mouse is found. Then, if the viewport's bitmap contains any active regions, they are searched. If the mouse's projected position on the viewport's bitmap is such that it falls inside one or more active regions, then the following rules apply:

- If the mouse has exited an active region, the active region's exit method is invoked.
- If the mouse has entered an active region, the active region's entry method is invoked.
- If the mouse is inside one or more active regions, each active region's method for the event is invoked.

The method is called with the following sequence of arguments:

- The viewport on which the mouse event occurred
- The active region
- The mouse event
- The x-coordinate of the position on which the mouse event occurred
- The y-coordinate of the position on which the mouse event occurred

The x- and y-coordinates are given relative to the origin of the active region's bitmap.

For all mouse events except **:mouse-exit-region**, the x-coordinate and y-coordinate arguments specify a position inside the active region. For :mouse-exit-region, the specified position lies outside the active region; it may also lie outside the bitmap.

You can use the macro **with-mouse-methods-preempted** to force the Window Tool Kit to ignore all active regions or to ignore all active regions except those attached to a specific bitmap.

Normally, active region methods and interrupt character methods are executed in the order that they occur, and no method is executed until the code for the previous method has finished. You can use the macro **with-asynchronous-method-invocation-allowed** inside a method to allow the execution of other methods before that method has finished execution.

1.9 Windows

A **window** is a composite object that combines the functionality of a bitmap, a viewport, a bitmap output stream, and a mouse input stream. Any function that takes one of these as an argument can take a window as an argument.

The predicates **viewportp**. **bitmap-output-stream-p** , and **mouse-input-stream-p** are true for a window.

Windows are included in the viewport hierarchy and, like viewports, are mappings from a bitmap onto the screen. A window can have a border and a title. The border consists of two parts: a black strip around the edge of the window and a white strip inside the black strip.

The window's viewport and bitmap output stream write onto the area inside the border.

1.9.1 Scroll Bars

You can create windows with two scroll bars by using the options provided for the function make-window. Scroll bars do the following:

They indicate what portion of the bitmap is inside the viewport's bitmap clipping region.

They let you move the bitmap clipping region with the mouse.

Scroll bars are generally used when a window's bitmap is larger than the bitmap clipping region of the window's viewport. When this is the case, you see only a portion of the bitmap at a time.

Scroll bars are two gray bars-a vertical bar that appears on the right-hand side of the window and a horizontal bar that appears at the bottom of the window. The top and bottom edges of the vertical scroll bar represent the top and bottom edges of the bitmap respectively. Similarly, the left and right edges of the horizontal scroll bar represent the left and right edges of the bitmap respectively.

Within each of the two scroll bars is a "bubble". This bubble represents the position of the bitmap clipping region within the bitmap. If the bubble is near the top of the vertical scroll bar, then the visible portion of the bitmap is near the top of the bitmap. If the bubble is near the center of the horizontal scroll bar, then the visible portion of the bitmap is about halfway between the right and left edges of the bitmap.

Scroll bars are mouse sensitive. You can move the bitmap clipping region by moving the mouse onto the right or bottom scroll bar. When you move the mouse onto a scroll bar, the mouse cursor changes. The new cursor indicates that you can now use the mouse to move the bitmap clipping region.

Once you move the mouse off either of the scroll bars, the mouse cursor changes back to its former shape.

1.9.2 Operations on Windows

The following operations are defined for windows:

make-window	*window-title*
window-frame	*window-title-font*
window-horizontal-scroll-ratio	*window-vertical-scroll-ratio*
window-inner-border-width	*windowp*
window-outer-border-width	*windows-available-p*

1.9.3 Keyboard Input and Interrupt Characters

When a character is typed at the keyboard, that character is sent to the mouse input stream that is the value of the function **keyboard-input**. The **setf** macro can be used to modify the mouse input stream to which characters typed at the keyboard are sent.

Each mouse input stream can have a set of interrupt characters associated with it. When they are typed to the mouse input stream, these interrupt characters do not get queued on the stream. Instead, the Window Tool Kit immediately calls the function that is associated with that character.

The function **mouse-input-stream-interrupt-char** accesses the function that is called when a character is typed to a mouse input stream. Its syntax is the following:

mouse-input-stream-interrupt-char **[Function]**

mouse-input-stream char

This function returns **nil** if the char argument is not an interrupt character on the stream **mouse-input-stream**.

The **setf** macro can be used with **mouse-input-stream-interrupt-char** to modify a character's interrupt function. If you set the value to **nil**, the character is no longer an interrupt character. If you set the value to a function, the character becomes an interrupt character on that mouse input stream.

When an interrupt character is typed on the mouse input stream, the corresponding function is called with these two arguments:

- The mouse input stream that received the character.
- The character.

Normally, active region methods and interrupt character methods are executed in the order that they occur, and no method is executed until the code for the previous method has finished. You can use the macro **with-asynchronous-method-invocation-allowed** inside a method to allow the execution of other methods before that method has finished execution.

1.10 Pop-Up Menus

A pop-up menu is a viewport that is displayed temporarily on the screen and that offers you a set of options. You can either select one of the options by placing the mouse over that item and clicking the right button or make no choice by moving the mouse off the menu. In either case, the pop-up menu then disappears.

This process is divided into two steps. The function **make-pop-up-menu** creates a new pop-up menu object. When the function **pop-up-menu-choose** is passed a pop-up menu object, that menu appears on the screen near the current location of the mouse. The function returns a value that depends on what you choose from the menu. A pop-up menu object can be passed repeatedly to the function **pop-up-menu-choose.**

The following operations are defined for pop-up menus:

make-pop-up-menu	*pop-up-menu-p*
pop-up-menu-choose	

2 Window Tool Kit Index

activate-viewport, deactivate-viewport

Purpose:

The function **activate-viewport** makes the specified viewport and all of its ancestors active. If the key *activate-children* is non-nil, then all of the viewport's descendants are also made active.

The function **deactivate-viewport** makes the specified viewport and all of its descendants inactive. The viewport *viewport* maintains its position in the display stack of its siblings. However, the viewport and its descendants do not appear on the screen until they are reactivated.

Syntax:

activate-viewport **[Function]**

viewport &key activate-children

deactivate-viewport **[Function]**

viewport

Remarks:

If a viewport is active, all of its ancestors are active. If a viewport is inactive, all of its descendants are inactive.

If **deactivate-viewport** tries to deactivate a viewport that is already inactive, nothing happens.

If **activate-viewport** tries to activate a viewport that is already active, nothing happens.

These functions are extensions to Common Lisp.

active-region-bitmap

Purpose:

The function **active-region-bitmap** returns the bitmap to which the argument *active-region* is attached. If *active-region* is not attached to a bitmap, it returns nil.

Syntax:

active-region-bitmap **[Function]**

active-region

Remarks:

This function is an extension to Common Lisp.

active-region-method

Purpose:

The function **active-region-method** accesses the method that is called when a mouse event occurs inside an active region or when the mouse enters or leaves an active region. The function returns nil if no method is associated with the event.

Syntax:

active-region-method **[Function]**

active-region event-name

Remarks:

The **setf** method for this function updates the appropriate method. If you set the value to nil, no method is called when the corresponding mouse event occurs.

The method is called with the following sequence of arguments:

- The viewport on which the mouse event occurred
- The active region
- The mouse event
- The x-coordinate of the position on which the mouse event occurred
- The y-coordinate of the position on which the mouse event occurred

The x- and y-coordinates are given relative to the origin of the active region's bitmap. For all mouse events except **:mouse-exit-region**, the x- coordinate and y-coordinate arguments specify a position inside the active region. For **:mouse-exit-region**, the specified position lies outside the active region; it may also lie outside the bitmap.

If the mouse's projected position on the viewport's bitmap falls inside one or more active regions, the following methods are invoked in the order given:

- If the mouse has exited an active region, the active region's exit method is invoked.
- If the mouse has entered an active region, the active region's entry method is invoked.
- If the mouse is inside one or more active regions, each active region's method for the event is invoked.

This function is an extension to Common Lisp.

active-region-p

Purpose:

The predicate **active-region-p** tests whether its argument *object* is an active region. It returns true if *object* is an active region.

Syntax:

active-region-p **[Function]**
object

Remarks:

This function is an extension to Common Lisp.

attach-active-region, detach-active-region, bitmap-active-regions, clear-bitmap-active-regions

Purpose:

The function **attach-active-region** attaches an active region to a bitmap.

The function **detach-active-region** detaches an active region from its bitmap.

The function **bitmap-active-regions** returns a list of all the active regions that are attached to a bitmap.

The function **clear-bitmap-active-regions** detaches all active regions that are attached to a bitmap.

Syntax:

attach-active-region **[Function]**
bitmap active-region

detach-active-region **[Function]**
active-region

bitmap-active-regions **[Function]**
bitmap

clear-bitmap-active-regions **[Function]**
bitmap

Remarks:

When you attempt to attach an active region to a bitmap, the active region must be located in the bitmap.

If **detach-active-region** is called with an active region that is not attached to a bitmap, nothing happens.

These functions are extensions to Common Lisp.

bitblt, bitblt-position, bitblt-region

Purpose:

The function **bitblt** copies regions from one bitmap to another.

The function **bitblt-position** is similar to **bitblt**, except that the locations in each bitmap are expressed as positions rather than as x- and y-coordinates.

The function **bitblt-region** is similar to **bitblt**, except that the arguments explicitly specify the source and destination regions.

Syntax:

bitblt **[Function]**

source-bitmap source-x source-y
destination-bitmap destination-x destination-y
width height operation
&key :clipping-region

bitblt-position **[Function]**

source-bitmap source-position
destination-bitmap destination-position
width height operation
&key :clipping-region

bitblt-region **[Function]**

source-bitmap source-region
destination-bitmap destination-region
operation

Remarks:

The arguments *source-bitmap* and *destination-bitmap* specify the bitmap from which the copying is performed and the bitmap to which the copying is done respectively. They may be the same bitmap.

The *source-bitmap* region that is copied is specified by one of the following:

- The *source-x*, *source-y*, *width*, and *height* arguments of **bitblt**. The *source-x* and *source-y* arguments specify the x- and y-coordinates respectively of the region's origin. The *width* and *height* arguments specify the region's width and height respectively.
- The *source-position*, *width*, and *height* arguments of **bitblt-position**. The *source-position* argument specifies the position of the region's origin. The *width* and *height* arguments specify the region's width and height respectively.
- The *source-region* argument of **bitblt-region**.

The *destination-bitmap* region that is to be modified is specified by one of the following:

- The *destination-x*, *destination-y*, *width*, and *height* arguments of **bitblt**. The *destination-x* and *destination-y* arguments specify the x- and y-coordinates respectively of the region's origin. The *width* and *height* arguments specify the region's width and height respectively.
- The *destination-position*, *width*, and *height* arguments of **bitblt-position**. The *destination-position* argument specifies the position of the region's origin. The *width* and *height* arguments specify the region's width and height respectively.
- The *destination-region* argument of **bitblt-region**.

Each position in the source bitmap region is combined with the corresponding position in the destination bitmap region, and the result is stored in the destination bitmap. The new value of the destination bitmap is the value returned when the function **boole** is applied to these three arguments: the *operation* argument, the value of the bit at the source bitmap position, and the value of the bit at the destination bitmap position. **Note:** For the Allegro Window Tool Kit only the following operations are defined: **boole-1, boole-and, boole-andc1, boole-c1, boole-eqv, boole-ior, boole-orc1, boole-xor.**

The keyword argument **:clipping-region** specifies a region of the destination bitmap. If this keyword argument is given, only the region of the destination region that is located inside the clipping region is modified.

If the *source-region* and *destination-region* arguments of **bitblt-region** are different widths, the width of the region that is actually copied is the smaller of the two. Similarly, if the *source-region* and *destination-region* arguments have different heights, the height of the region that is copied is the smaller of the two.

These functions are extensions to Common Lisp.

bitmap-extent, bitmap-height, bitmap-width

Purpose:

These functions access and modify information about a bitmap.

The function **bitmap-extent** creates a copy of a bitmap's extent.

The function **bitmap-height** returns the height of a bitmap.

The function **bitmap-width** returns the width of a bitmap.

Syntax:

bitmap-extent [Function]

bitmap &optional result-extent

bitmap-height [Function]

bitmap

bitmap-width [Function]

bitmap

Remarks:

If a *result-extent* argument is specified for **bitmap-extent**, that extent is modified to the output extent and then returned. Otherwise a new extent is created and returned.

You can use the **setf** macro with these functions. Increasing the width or height of a bitmap causes new area to appear at its boundaries. Decreasing the width or height may cause loss of data.

These functions are extensions to Common Lisp.

Examples:

```
> (setq my-bitmap (make-bitmap :width 100 :height 200))
#<Bitmap 100x200 25F391>
> (bitmap-extent my-bitmap)
#<Extent 100x200 25F7A7>
> (bitmap-height my-bitmap)
200
> (bitmap-width my-bitmap)
100
;; Create a 0x0 extent.
> (setq empty-extent (make-extent))
#<Extent 0x0 25F84C>
;; Copy the extent of my-bitmap into empty-extent.
> (bitmap-extent my-bitmap empty-extent)
#<Extent 100x200 25F84C>
;; Now look at the value of empty-extent.
> empty-extent
#<Extent 100x200 25F84C>
```

bitmap-output-stream-p

Purpose:

The predicate **bitmap-output-stream-p** tests whether its argument *object* is a bitmap output stream. It returns true if *object* is a bitmap output stream.

Syntax:

bitmap-output-stream-p **[Function]**
object

Remarks:

This function is an extension to Common Lisp.

Examples:

>(bitmap-output-ctream-p (make-bitmap-output-stream))
T
> (bitmap-output-stream-p 7)
NIL

bitmap-p

Purpose:

The predicate **bitmap-p** tests whether its argument *object* is a bitmap. It returns true if *object* is a bitmap.

Syntax:

bitmap-p **[Function]**

object

Remarks:

This function is an extension to Common Lisp.

Examples:

```
> (bitmap-p (make-bitmap :height 100 :width 200))
T
> (bitmap-p 7)
NIL
```

bitmap-value

Purpose:

The function **bitmap-value** returns the value of a bitmap's point at a given x-y coordinate.

Syntax:

bitmap-value **[Function]**

bitmap x y

Remarks:

The result is either 0 or 1.

You can use the **setf** macro with this function to set the value of a point in a bitmap.

Note: This function is not yet defined for the X Window Tool Kit!

This function is an extension to Common Lisp.

Examples:

```
;; Create a 100x200 bitmap.
> (setq bmp (make-bitmap :width 100 :height 200))
#<Bitmap 100x200 AEC25B>
;; Look at the value of a point.
> (bitmap-value bmp 23 56)
0
;; Set the point to one.
> (setf (bitmap-value bmp 23 56) 1)
1
;; Look at the value of that point.
> (bitmap-value bmp 23 56)
1
```

charblt, stringblt

Purpose:

The function **charblt** paints a character image from a font onto a bitmap.

The function **stringblt** paints a string of character images from a font onto a bitmap.

Syntax:

charblt **[Function]**

bitmap position font char &key :operation

stringblt **[Function]**

bitmap position font string &key :operation

Remarks:

The **:operation** keyword argument controls how the font is painted onto the bitmap. The new value of the destination bitmap is the value returned by applying the function boole to these three arguments: the **:operation** argument, the value of the font's bit, and the value of the destination bitmap position. If the **:operation** keyword argument is omitted or nil, the default value is the value of **boole-1**. This default value causes the bits of the font's bitmap to overwrite whatever was previously on the bitmap. **Note:** For the Allegro Window Tool Kit only the following operations are defined: **boole-1, boole-and, boole-andc1, boole-c1, boole-eqv, boole-ior, boole-orc1, boole-xor.**

The *position* argument specifies the position at which the character or characters are output. The first character is aligned so that the left-most point of its baseline is at the point given by the *position* argument.

The function **stringblt** cannot handle tabs and other characters that have an ambiguous print representation. It can handle newline and space characters.

These functions are extensions to Common Lisp.

See Also:

bitblt

clear-bitmap

Purpose:

The function **clear-bitmap** clears a bitmap. That is, the value of every point in the bitmap is set to 0.

Syntax:

clear-bitmap **[Function]**

bitmap &optional region

Remarks:

If a *region* argument is specified, only that region of the bitmap is cleared. Otherwise the entire bitmap is cleared.

This function is an extension to Common Lisp.

Examples:

```
;; Create a 10x10 bitmap.
> (setq btmp (make-bitmap :width 10 :height 10))
#<Bitmap 10x10 596D5D>
;; Put ones on the diagonal of the bitmap.
> (dotimes (i 10) (setf (bitmap-value btmp i i) 1))
NIL
;; A point on the diagonal has a value of one.
> (bitmap-value btmp 3 3)
1
;; A point not on the diagonal has a value of zero.
> (bitmap-value btmp 3 2)
0
;; Clear a region of the bitmap.
> (clear-bitmap btmp (make-region :x 2 :y 2 :height 3 :width 3))
#<Bitmap 10x10 596D5D>
;; Look at a diagonal point that was cleared.
> (bitmap-value btmp 3 3)
0
;; Look at a diagonal point that was not cleared.
> (bitmap-value btmp 9 9)
1
```

copy-bitmap

Purpose:

The function **copy-bitmap** copies a bitmap.

Syntax:

copy-bitmap **[Function]**
bitmap

Remarks:

The original bitmap and the copy can be modified without affecting each other.

This function is an extension to Common Lisp.

Examples:

```
;; Create a 100x200 bitmap.
> (make-bitmap :height 100 :width 200)
#<Bitmap 200x100 5D95EE>
;; Make a copy of the bitmap.
> (copy-bitmap *)
#<Bitmap 200x100 5D9937>
```

current-mouse-cursor

Purpose:

The function **current-mouse-cursor** returns the mouse cursor object that is currently tracking the mouse on the display screen.

Syntax:

current-mouse-cursor **[Function]**

Remarks:

You can use the macro **setf** with this function to modify the mouse cursor object that is tracking the mouse.

This function is an extension to Common Lisp.

default-font

Purpose:

The value of the variable ***default-font*** is used as a default value by the functions **make-bitmap-output-stream** and **make-window**.

Syntax: *default-font* **[Variable]**

Remarks:

This variable is an extension to Common Lisp.

Examples:

```
;; Create a bitmap output stream.
;; Do not give an :initial-font keyword argument.
> (make-bitmap-output-stream :width 100 :height 200)
#<Output-Stream to #<Bitmap 100x200 1A45D8> 1A49E0>
;; Check to see that the stream's font is *default-font*.
> (eq (stream-current-font *) *default-font*)
T
```

See Also:

find-font

delete-viewport

Purpose:

The function **delete-viewport** deletes a viewport or window and removes it from the viewport hierarchy. The viewport's resources can then be garbage collected if no user-defined data structures refer to the viewport.

Syntax:

delete-viewport **[Function]**
viewport

Remarks:

This function is an extension to Common Lisp.

draw-circle, draw-line, draw-polyline, draw-polypoint

Purpose:

The function **draw-circle** draws a circle whose center is the position *center* and whose radius is *radius*.

The function **draw-line** draws a line segment from the position *start* to the position *end*.

The function **draw-polyline** takes a sequence of positions *positions* and connects each adjacent pair.

The function **draw-polypoint** takes a sequence of positions *positions* and draws a dot at each one.

Syntax:

draw-circle **[Function]**

bitmap center radius
&key :width :operation

draw-circle draw-line **[Function]**

bitmap start end
&key :width :operation

draw-circle draw-polyline **[Function]**

bitmap positions
&key :width :operation

draw-circle draw-polypoint **[Function]**

bitmap positions
&key :width :operation

Remarks:

Note: The function **draw-line** is **not exported** from the X Window Tool Kit, when KCL (Kyoto Common Lisp) is used! Please refer to it with **windows::draw-line**.

If the **:width** keyword argument is given, it defines the line width that is used for drawing the line segments and circles. For **draw-polypoint**, the **:width** argument specifies the diameter of the dot. For **draw-circle**, the border is drawn so that its outer edge is at the specified radius; the width must be less than or equal to the radius. If the **:width** keyword argument is omitted or nil, the default value 1 is used.

The **:operation** keyword value is used to control how the values that are being written onto the bitmap combine with the values that are already present. If this keyword argument is omitted or nil, the default value is the value of the constant **boole-1** this default value causes the values that are being written onto the bitmap to overwrite whatever was already on the bitmap.

These functions are extensions to Common Lisp.

expose-viewport, hide-viewport

Purpose:

The function **expose-viewport** moves a viewport to the top of its sibling stack. Nothing happens if the viewport is already at the top of the stack.

The function **hide-viewport** moves a viewport to the bottom of its sibling stack. Nothing happens if the viewport is already at the bottom of the stack.

Syntax:

expose-viewport **[Function]**

viewport

hide-viewport **[Function]**

viewport

Remarks:

In complex hierarchies, **expose-viewport** may not place the viewport on the screen unoccluded because it may be occluded by its children, or because its parent may be occluded.

If two active viewports overlap on the screen, the following rules determine which viewport occludes the other:

- A viewport occludes all of its ancestor viewports.
- If two viewports are siblings, then the viewport that is closest to the top of the sibling stack and all of its descendants occlude the viewport that is farther down and all of its descendants.

These functions are extensions to Common Lisp.

See Also:

viewport-children

extent-height, extent-width

Purpose:

The function **extent-height** returns the height of an extent.

The function **extent-width** returns the width of an extent.

Syntax:

extent-height **[Function]**

extent

extent-width **[Function]**

extent

Remarks:

You can use the **setf** macro with these functions to modify the height and width of an extent.

These functions are extensions to Common Lisp.

Examples:

```
> (setq x (make-extent 100 200))
#<Extent 100x200 1A4D04>
> (extent-height x)
200
> (setf (extent-width x) 300)
300
> x
#<Extent 300x200 1A4D04>
```

extentp

Purpose:

The predicate **extentp** tests whether its argument *object* is an extent. It returns true if *object* is an extent.

Syntax:

extentp **[Function]**

object

Remarks:

This function is an extension to Common Lisp.

Examples:

```
> (extentp (make-extent))
T
> (extentp 7)
NIL
```

find-font

Purpose:

The function **find-font** finds the font whose name is the *name* argument and returns that font. The function returns nil if it cannot find a font with that name.

Syntax:

find-font **[Function]**

name

Remarks:

This function is an extension to Common Lisp.

font-baseline, font-height, font-fixed-width

Purpose:

The function **font-baseline** returns the baseline height of a font. This number is the baseline height of every character in the font.

The function **font-height** returns the height of a font. This number is the height of every character in the font.

The function **font-fixed-width** returns the width of every character in a font if the font is a fixed-width font; otherwise it returns nil.

Syntax:

font-baseline **[Function]**

font

font-height **[Function]**

font

font-fixed-width **[Function]**

font

Remarks:

These functions are extensions to Common Lisp.

See Also:

find-font

font-name

Purpose:

The function **font-name** returns the name of a font.

Syntax:

font-name **[Function]**

font

Remarks:

Note: The function **font-name** is **not exported** from the X Window Tool Kit, when KCL (Kyoto Common Lisp) is used! Please refer to it with **windows::font-name**.

This function is an extension to Common Lisp.

See Also:

find-font

fontp

Purpose:

The predicate **fontp** tests whether its argument *object* is a font. It returns true if *object* is a font.

Syntax:

fontp **[Function]**

object

Remarks:

This function is an extension to Common Lisp.

Examples:

```
> (fontp *default-font*)
T
> (fontp 7)
NIL
```

initialize-windows

Purpose:

The function **initialize-windows** initializes the Window Tool Kit.

Syntax:

initialize-windows **[Function]**

&key :height :width :screen-x :screen-y

Remarks:

You must call the function **initialize-windows** to initialize the window system.

The keyword arguments **:height** and **:width** specify the size of the usable portion of the system window that will be the root viewport. Because the usable portion does not include the window border and legend, the size of the created system window will be larger than the specified size. The **:width** and **:height** keywords both default as the maximum width and the maximum height.

The keyword arguments **:screen-x** and **:screen-y** specify the position of the upper left corner of the usable portion of the system window.

If you try to initialize the Window Tool Kit but it has already been initialized, nothing happens.

If the Window Tool Kit has been initialized and you want to make an image, please first delete the window environment (with function **leave-window-system**). To restore the state of the windows in the newly saved image, call initialize-windows again and rebuild your window environment.

This function is an extension to Common Lisp.

See Also:

leave-window-system

keyboard-input

Purpose:

The function **keyboard-input** determines where keyboard input is sent. Any character typed at the keyboard is sent to the mouse input stream that is the value of this function.

Syntax:

keyboard-input **[Function]**

Remarks:

The **setf** macro can be used with this function to change the stream to which keyboard input is sent. The second argument to **setf** must be a mouse input stream.

This function is an extension to Common Lisp.

leave-window-system

Purpose:

The function **leave-window-system** exits the Window Tool Kit.

Syntax:

leave-window-system **[Function]**

Remarks:

If you exit the window environment by calling **leave-window-system**, you cannot return to it. If you wish to use the Window Tool Kit after calling this function, you must set up new windows by invoking **initialize-windows**.

This function is an extension to Common Lisp.

See Also:

initialize-windows

listen-any

Purpose:

The predicate **listen-any** is true if a character or mouse event object can be read from the given mouse input stream; otherwise it is false.

Syntax:

listen-any **[Function]**

&optional mouse-input-stream

Remarks:

Note: For the X Window Tool Kit the input functions may return keywords like *:escape*, *:meta-a*, or *:f1* instead of characters.

The argument *mouse-input-stream* specifies a mouse input stream. If this argument is omitted or nil, the mouse input stream that is the value of the function **mouse-input** is used. If the *mouse-input-stream* argument is t, the mouse input stream that is the value of the function **keyboard-input** is used.

This function is an extension to Common Lisp.

See Also:

keyboard-input
mouse-input
peek-any
read-any
read-any-no-hang

make-active-region

Purpose:

The function **make-active-region** creates an active region for the region *region*; as an option it can attach that active region to a bitmap.

Syntax:

make-active-region **[Function]**

region &key :bitmap
:mouse-left-down
:mouse-left-up
:mouce-middle-down
:mouse-middle-up
:mouse-right-down
:mouse-right-up
:mouse-moved
:mouse-enter-region
:mouse-exit-region

Remarks:

The **:bitmap** keyword argument is the bitmap to which this active region should be attached. If this keyword argument is omitted or nil, then the active region is not attached to any bitmap. Later it may be attached to a bitmap by using the function **attach-active-region**.

The rest of the keyword arguments specify the methods for each of the nine types of mouse events. The value of each keyword argument must be a function of five arguments. The method is called whenever the corresponding mouse event occurs inside the created active region.

If a mouse event keyword argument is omitted or nil, no method is associated with the mouse event. No function is called when the mouse event occurs inside the created active region.

The method is called with the following sequence of arguments:

- The viewport on which the mouse event occurred
- The active region
- The mouse event
- The x-coordinate of the position on which the mouse event occurred
- The y-coordinate of the position on which the mouse event occurred

The x- and y-coordinates are given relative to the origin of the active region's bitmap.

For all mouse events except **:mouse-exit-region**, the x-coordinate and y-coordinate arguments specify a position inside the active region. For **:mouse-exit-region**, the specified position lies outside the active region; it may also lie outside the bitmap.

This function is an extension to Common Lisp.

See Also:

attach-active-region

make-bitmap

Purpose:

The function **make-bitmap** creates a bitmap.

Syntax:

make-bitmap **[Function]**

&key :extent :width :height

Remarks:

Note: The function **make-bitmap** is **not exported** from the Allegro Window Tool Kit! Please refer to it with **windows::make-bitmap**.

The width and height of the bitmap are specified by using the **:width** and **:height** keyword arguments or by supplying an extent with the **:extent** keyword argument. **Note:** For the Allegro Window Tool Kit the maximum width is **ccl:*screen-width*** and the maximum height is **ccl:*screen-height***.

Unspecified dimensions default to 0.

This function is an extension to Common Lisp.

Examples:

> (make-bitmap : height 100 : width 200)
#<Bitmap 200x100 856EE6>

make-bitmap-output-stream

Purpose:

The function **make-bitmap-output-stream** creates a bitmap output stream. The bitmap output stream can be attached to an already existing bitmap, or it can be attached to a new bitmap created by this function.

Syntax:

make-bitmap-output-stream **[Function]**
&key :bitmap
:extent :width :height
:operation
:initial-font

Remarks:

The value of the **:bitmap** keyword argument must be a bitmap. The bitmap output stream is attached to that bitmap. If this keyword argument is omitted or nil, a new bitmap is created. The new bitmap's size can be specified with either the **:width** and **:height** keyword arguments or with the **:extent** keyword argument (whose value should be an extent). An unspecified width or height defaults to 0. **Note:** For the Allegro Window Tool Kit the maximum width is **ccl:*screen-width*** and the maximum height is **ccl:*screen-height***.

The **:operation** keyword argument is the boolean operation used by the stream to write onto the bitmap. Its default value is the value of the constant boole-xor.

The **:initial-font** keyword argument is the font in which characters are painted onto the bitmap. The value of this keyword argument must be a font, a string, or a symbol. If the argument is a string or a symbol, the function find-font is called to find the font whose name is the string or symbol. The default value is the value of ***default-font***.

The stream position of a newly created bitmap output stream is the position whose x-coordinate is 0 and whose y-coordinate is the baseline height of the initial font. Its linefeed distance is the height of the initial font.

Note: Do not attach an output stream to the bitmap that is associated with the root viewport.

This function is an extension to Common Lisp.

make-extent

Purpose:

The function **make-extent** creates an extent whose width is *width* and whose height is *height.*

Syntax:

make-extent **[Function]**

&optional width height

Remarks:

The arguments to **make-extent** are fixnums. If either argument is omitted, the default value 0 is used.

This function is an extension to Common Lisp.

Examples:

```
> (make-extent)
#<Extent 0x0 855B5B>
> (make-extent 100 200)
#<Extent 100x200 855B80>
```

make-mouse-input-stream

Purpose:

The function **make-mouse-input-stream** creates a mouse input stream. A mouse input stream can queue both characters and mouse event objects.

Syntax:

make-mouse-input-stream **[Function]**

&key :queue-mouse-events-p :viewport

Remarks:

The **:queue-mouse-events-p** keyword argument determines whether this mouse input stream initially queues mouse event objects. The default value for this keyword argument is nil, which means that only characters are queued on the newly created mouse input stream.

The **:viewport** keyword argument is the viewport associated with the mouse input stream that is being created. If this keyword argument is omitted or nil, the mouse input stream is associated with the root viewport.

This function is an extension to Common Lisp.

See Also:

listen-any
mouse-input-stream-queue-mouse-events-p
peek-any read-any
unread-any

make-pop-up-menu

Purpose:

The function **make-pop-up-menu** creates a pop-up menu object.

Syntax:

make-pop-up-menu **[Function]**
choice-list &optional default-value

Remarks:

The argument *choice-list* is a list. Each element of the list is either a symbol or a cons whose car is a string.

If the element is a symbol, then when the function **pop-up-menu-choose** displays the pop-up menu, the print name of the element is displayed as one of the choices. If chosen, the element is returned as the value of **pop-up-menu-choose**.

If the element is a cons, then the car of the element, which must be a string, is displayed as one of the choices. If the element is chosen, the value of **pop-up-menu-choose** is the cdr of the cons.

If the mouse is moved off the choice menu, the *default-value* argument is returned. If this argument is omitted, the default value is nil.

This function is an extension to Common Lisp.

See Also:

pop-up-menu-choose

make-position

Purpose:

The function **make-position** creates a position. The coordinates of this position are the x and y arguments.

Syntax:

make-position **[Function]**

&optional x y

Remarks:

The arguments must be nonnegative fixnums. If either argument is omitted, the default value 0 is used.

This function is an extension to Common Lisp.

Examples:

```
> (make-position)
#<Position (0,0) 855C2D>
> (make-position 100 200)
#<Position (100,200) 855C3E>
```

make-region

Purpose:

The function **make-region** creates a new region.

Syntax:

make-region **[Function]**

&key :origin :x :y
:extent :width :height
:corner :corner-x :corner-y

Remarks:

To create a region, you must specify two of the following three attributes of a region: its origin, its corner, and its size. A region's origin is its top-left position. A region's corner is the point just below and to the right of its bottom-right position. A region's size is its height and width.

You specify the origin of a region by specifying the position of the origin with the **:origin** keyword argument or by specifying the :x- and y-coordinates of the origin separately with the **:x** and **:y** keyword arguments.

You specify the corner of a region by specifying the position of the corner with the **:corner** keyword argument or by specifying the :x- and y-coordinates of the corner separately with the **:corner-x** and **:corner-y** keyword arguments.

You specify the size of a region by specifying the region's extent with the **:extent** keyword argument or by specifying the width and height of the region separately with the **:width** and **:height** keyword arguments.

This function is an extension to Common Lisp.

Examples:

```
;; You can specify a region whose origin is the point (400,500)
;; and whose corner is the point (480,690) in several different ways.
;; mid-screen is the position of the origin.
> (setq mid-screen (make-position 400 500))
#<Position (400,500) 855D50>
;; ext is the size of the region.
> (setq ext (make-extent 80 90))
#<Extent 80x90 855D64>
;; Give the origin and size of the region.
> (setq reg1 (make-region :origin mid-screen :extent ext))
#<Region 80x90 at (400,500) 855D84>
;; Give the origin and size but specify each coordinate separately.
> (setq reg2 (make-region :x 400 :y 500 :width 80 :height 90))
#<Region 80x90 at (400,500) 855DBE>
;; Give the size and the corner.
> (setq reg3 (make-region :extent ext
:corner (make-position 480 590)))
#<Region 80x90 at (400,500) 855DEB>
;; Verify that all three regions specify the same region.
> (region= reg1 reg2 reg3)
T
```

make-viewport

Purpose:

The function **make-viewport** creates a viewport. The viewport is attached to an already existing bitmap or to a newly created bitmap.

The function returns two values: the newly created viewport and the bitmap to which the viewport is attached.

Syntax:

make-viewport **[Function]**

&key :bitmap :width :height
:bitmap-region
:parent :fixed
:screen-position
:screen-x :screen-y
:activate

Remarks:

The keyword options to this function are described as follows:

- **:bitmap**
  This keyword argument specifies the bitmap to which the viewport is attached. Its value must be a bitmap made with the function **make-bitmap**.

  If this keyword argument is omitted or nil, the viewport is attached to a new bitmap whose dimensions are specified by **:width** and **:height**.

- **:width, :height**
  These keyword arguments specify the width and height of the bitmap to which the viewport is attached. The value of each must be a nonnegative fixnum. If either is omitted or nil, its default value is 0. **Note:** For the Allegro Window Tool Kit the maximum width is **ccl:*screen-width*** and the maximum height is **ccl:*screen-height***.

You only need to use **:width** and **:height** if **:bitmap** is omitted or nil.

- **:bitmap-region**
  This keyword argument specifies the viewport's bitmap clipping region. Its value must be a region made with the function **make-region**.

  If this keyword argument is omitted or nil, the bitmap clipping region is the entire bitmap; thus, the viewport and bitmap have the same size.

- **:parent**
  This keyword argument specifies the parent viewport of the new viewport. Its value must be an existing viewport. If it is omitted or nil, the root viewport becomes the parent viewport. The new viewport is put at the top of its sibling stack.

- **:screen-position**
  This keyword argument specifies the position of the viewport's top-left corner. Its value must be a position made with the function **make-position**.

  The **:screen-x** and **:screen-y** keyword arguments can be used as an alternative to **:screen-position**. The default value for the :x- and y-coordinates is 0.

- **:screen-x, :screen-y**
  These keyword arguments specify the coordinates of the viewport's top-left corner relative to the root viewport. The value of each must be a nonnegative fixnum. If either is omitted, its default value is 0.

- **:activate**
  This keyword argument specifies whether the viewport is active or inactive. If it is omitted or non-nil, the viewport is active. If it is specified and nil, the viewport is inactive.

Note: A viewport's screen clipping region is the region whose top-left corner is the point specified by either **:screen-position** or **:screen-x** and **:screen-y**, and whose extent is the same as that of the viewport's bitmap clipping region.

This function is an extension to Common Lisp.

Examples:

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.
;; Create a 100x200 bitmap and a viewport onto that bitmap.
> (make-viewport :width 100 :height 200)
#<Viewport 100x200 at (0,0) onto #<Bitmap 100x200 85516A>
855572>
#<Bitmap 100x200 85516A>
;; Create another 100x200 bitmap.
> (setq btmp (make-bitmap :width 100 :height 200))
#<Bitmap 100x200 85560C>
;; Create a viewport onto that bitmap.
;; Note that the bitmap is returned
;; as the second value.
> (make-viewport :bitmap btmp)
#<Viewport 100x200 at (0,0) onto #<Bitmap 100x200 85560C>
855A2A>
#<Bitmap 100x200 85560C>
```

make-window

Purpose:

The function **make-window** creates and returns a window.

A window combines the functionality of a viewport, a bitmap, a bitmap output stream, and a mouse input stream. On the display screen, a window appears as a viewport. It may be surrounded by a border and may have a title. A window may also have a scroll bar.

Syntax:

make-window **[Function]**

&key :position :x :y
:extent :width :height
:inner-border-width :outer-border-width
:viewport-width :viewport-height
:initial-font :operation
:title :title-font
:parent :scroll :activate
:calculate-vertical-scroll-ratio
:calculate-horizontal-scroll-ratio
:vertical-scroll :horizontal-scroll
:calculate-horizontal-bubble-width
:calculate-vertical-bubble-height

Remarks:

The keyword options to this function are described as follows:

- **:position**
  This keyword argument specifies the position of the window's top-left corner. Its value must be a position made with the function **make-position**.

The :x and :y keyword arguments can be used as an alternative to **:position**. The default value for the x- and y-coordinates is 0.

- **:x, :y**
  These keyword arguments specify the coordinates of the window's top-left corner relative to the root viewport. The value of each must be a nonnegative fixnum. If either is omitted, its default value is 0.

- **:extent**
  This keyword argument specifies the size of the window's bitmap. Its value must be an extent made with the function **make-extent**.

  The **:width** and **:height** keyword arguments can be used as an alternative to **:extent**.

- **:width, :height**
  These keyword arguments specify the size of the window's bitmap. The value of each must be nonnegative fixnum. If either is omitted or nil, its default value is 0. **Note:** For the Allegro Window Tool Kit the maximum width is **ccl:*screen-width*** and the maximum height is **ccl:*screen-height***.

- **:inner-border-width, :outer-border-width**
  These keyword arguments specify the width of the window's inner and outer borders. The inner border is strip of white space that surrounds the viewport, and the outer border is a black box that surrounds the inner border. If **:inner-border-width** is omitted or nil, its default value is 1. If **:outer-border-width** is omitted or nil, its default value is 2.

- **:viewport-width, :viewport-height**
  These keyword arguments specify the width and height of the window's viewport. These dimensions can be different from those of the window's bitmap (specified with either **:width** and **:height** or **:extent**). However, if **:viewport-width** and **:viewport-height** are omitted or nil, their default values are the width and height of the bitmap respectively.

The total width of the window is the width of the viewport plus twice the thickness of the inner border plus twice the thickness of the outer border. The total height of the window is the height of the viewport plus twice the thickness of the inner border plus twice the thickness of the outer border plus the height of the title.

- **:initial-font**
  This keyword argument specifies the initial font used by the window's bitmap output stream. If it is omitted or nil, its default value is the value of the variable ***default-font***.
- **:operation**
  This keyword argument specifies the boolean operation that the bitmap output stream uses to write onto the bitmap. Its value must be an acceptable first argument to the boole function. If it is omitted or nil, its default value is the value of the constant **boole-xor**. **Note:** For the Allegro Window Tool Kit only the following operations are defined: **boole-1, boole-and, boole-andc1, boole-c1, boole-eqv, boole-ior, boole-orc1, boole-xor.**
- **:title**
  This keyword argument specifies the title of the window. Its value must be a string. If the window has a title, it appears in a title bar at the top of the window. If this keyword argument is omitted or nil, the window has no title.
- **:title-font**
  This keyword argument specifies the font in which the title is displayed. If it is omitted or nil, its default value is the value of the **:initial-font** keyword argument.
- **:parent**
  This keyword argument specifies the parent viewport of the new viewport. Its value must be an existing viewport. If it is omitted or nil, the root viewport becomes the parent viewport. The new viewport is put at the top of its sibling stack.
- **:activate**
  This keyword argument specifies whether the viewport is active or inactive. If it is omitted or non-nil, the viewport is active. If it is nil, the viewport is inactive.

- **:scroll**
  This keyword argument specifies whether the window has scroll bars. If it is t, the window is created with scroll bars on the right and bottom. If it is omitted or nil, the window is created without scroll bars. Do not give this keyword any value other than t or nil.

- **:calculate-horizontal-scroll-ratio, :calculate-vertical-scroll-ratio**
  These keyword arguments calculate the vertical and horizontal scroll ratio respectively. The scroll ratio is a Common Lisp ratio between 0 and 1. Generally, the scroll ratio is the ratio of the current location of the window to the size of the window's underlying bitmap. However, window system developers may redefine the methods for scrolling and for calculating these ratios so that scrolling may be performed over an abstract bitmap or extent. If either keyword argument is specified, it must be a function that takes the window as an argument. The functions cannot be used with the macro setf to specify the respective ratios; they can only return a ratio or nil.

- **:vertical-scroll, :horizontal-scroll**
  These keyword arguments replace the default scrolling methods for the window. If either is given, it must be a function that takes two arguments: the window to be scrolled and a vertical or horizontal scroll ratio that describes the location of scrolling.

- **:calculate-horizontal-bubble-width, :calculate-vertical-bubble-height**
  These keyword arguments replace the default scrolling methods for the window. If either is given, it must be a function that takes two arguments: the window to be scrolled and the maximum width of the vertical scroll bar bubble or the maximum height of the horizontal scroll bar bubble. By default the bubble size corresponds to the ratio of the size of the viewport to the size of the window's underlying bitmap.

This function is an extension to Common Lisp.

Examples:

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.
> (setq w (make-window :width 100 :height 200 :title "hello"))
#<WINDOW 4AC0AB>
> (windowp w)
T
```

See Also:

window-vertical-scroll-ratio
window-horizontal-scroll-ratio

mouse-x, mouse-y

Purpose:

The functions **mouse-x** and **mouse-y** return the current x- and y-coordinates of the mouse. These positions are relative to the root viewport.

Syntax:

mouse-x	**[Function]**
mouse-y	**[Function]**

Remarks:

These functions are extensions to Common Lisp.

See Also:

mouse-event-x
mouse-event-y
move-mouse

mouse-cursor-p

Purpose:

The predicate **mouse-cursor-p** tests whether its argument *object* is a mouse cursor object. It returns true if *object* is a mouse cursor object.

Syntax:

mouse-cursor-p **[Function]**

object

Remarks:

This function is an extension to Common Lisp.

mouse-event-p

Purpose:

The predicate **mouse-event-p** tests whether its argument *object* is a mouse event object. It returns true if *object* is a mouse event object.

Syntax:

mouse-event-p **[Function]**
object

Remarks:

This function is an extension to Common Lisp.

mouse-event-x, mouse-event-y, mouse-event-event-type

Purpose:

These functions access the fields of a mouse event object.

The functions **mouse-event-x** and **mouse-event-y** give the x- and y-coordinates of the mouse when the mouse event occurred that created the mouse event object. These coordinates are relative to the viewport that owns the mouse input stream on which the mouse event object was read.

The function **mouse-event-event-type** returns a keyword that indicates what mouse event created a particular mouse event object.

Syntax:

mouse-event-x **[Function]**

mouse-event-object

mouse-event-y **[Function]**

mouse-event-object

mouse-event-event-type **[Function]**

mouse-event-object

Remarks:

These functions are extensions to Common Lisp.

mouse-input

Purpose:

The function **mouse-input** determines where mouse input is sent.

Syntax:

mouse-input **[Function]**

Remarks:

The value of the function **mouse-input** is examined when a mouse event occurs. If the expression **(mouse-input-stream-queue-mouse-events-p (mouse-input))** is non-nil, an object encoding the mouse event is appended to the mouse input stream that is the value of the expression **(mouse-input)** .

The **setf** macro can be used with this function to change the stream to which mouse input is sent. The second argument to **setf** must be a mouse input stream.

This function is an extension to Common Lisp.

mouse-input-stream-interrupt-char

Purpose:

The function **mouse-input-stream-interrupt-char** returns the function that is called when the given character is typed to a mouse input stream.

Syntax:

mouse-input-stream-interrupt-char **[Function]**

mouse-input-stream char

Remarks:

The function returned by **mouse-input-stream-interrupt-char** takes two arguments: *mouse-input-stream* and *char*. The function is called as soon as *char* is typed to *mouse-input-stream*.

If the *char* argument is not an interrupt character, this function returns nil.

You can use the **setf** macro to modify a character's interrupt handler. If you set the function value to nil, *char* is no longer an interrupt character on *mouse-input-stream*. If you set the function value to a function of two arguments, *char* becomes an interrupt character, and the function is called when *char* is typed.

This function is an extension to Common Lisp.

mouse-input-stream-p

Purpose:

The predicate **mouse-input-stream-p** tests whether its argument *object* is a mouse input stream. It returns true if *object* is a mouse input stream.

Syntax:

mouse-input-stream-p **[Function]**

object

Remarks:

This function is an extension to Common Lisp.

mouse-input-stream-queue-mouse-events-p

Purpose:

When a mouse event occurs, the predicate **mouse-input-stream-queue-mouse-events-p** is called on the mouse input stream that is the value of the function **mouse-input**. If the value returned is non-nil, a mouse event object encoding the mouse event is queued on the mouse input stream that is the value of the function **mouse-input**.

Syntax:

mouse-input-stream-queue-mouse-events-p **[Function]**

mouse-input-stream

Remarks:

You can use the **setf** macro with this function to cause a mouse input stream to start or stop queueing mouse event objects.

The initial value for this function can be set in the function **make-mouse-input-stream** with the **:queue-mouse-events-p** keyword argument.

This function is an extension to Common Lisp.

See Also:

make-mouse-input-stream read-any

mouse-input-stream-viewport

Purpose:

The function **mouse-input-stream-viewport** returns the viewport that is associated with a mouse input stream.

Syntax:

mouse-input-stream-viewport **[Function]**

mouse-input-stream

Remarks:

This function is an extension to Common Lisp.

move-mouse

Purpose:

The function **move-mouse** moves the mouse cursor from its current position to the position specified by the x and y arguments.

Syntax:

move-mouse **[Function]**

x y

Remarks:

This function does not work for the Allegro Window Tool Kit.

This function is an extension to Common Lisp.

move-viewport

Purpose:

The function **move-viewport** moves a viewport's origin so that its top-left corner is at the point whose screen coordinates are specified by the *x* and *y* arguments.

Syntax:

move-viewport **[Function]**

viewport x y

Remarks:

The root viewport cannot be moved.

This function is an extension to Common Lisp.

peek-any

Purpose:

The function **peek-any** peeks at and returns the next character or mouse event object in a mouse input stream without reading it. The character or mouse event object is read at a later time.

You can also use **peek-any** for skipping over characters and mouse event objects in the input stream until a particular character is encountered.

Syntax:

peek-any **[Function]**

&optional peek-type mouse-input-stream

Remarks:

Note: For the X Window Tool Kit the input functions may return keywords like *:escape*, *:meta-a*, or *:f1* instead of characters.

The argument *mouse-input-stream* specifies a mouse input stream. If this argument is omitted or nil, the mouse input stream that is the value of the function **mouse-input** is used. If the *mouse-input-stream* argument is t, the mouse input stream that is the value of the function **keyboard-input** is used.

The *peek-type* argument specifies the type of object searched for on the mouse input stream. If *peek-type* is specified, it must be either nil, t, or a character. If this argument is omitted, the *peek-type* argument defaults to nil.

If the *peek-type* argument is nil, **peek-any** looks at and returns the next character or mouse event object in the mouse input stream without reading it from the stream.

If the *peek-type* argument is a character, then **peek-any** discards characters and mouse event objects from the front of the input stream until it encounters a character that is equal to (**char=**) the *peek-type* argument. That character is returned without being read from the stream.

This function is an extension to Common Lisp.

See Also:

keyboard-input
listen-any
mouse-event-p
mouse-input
read-any
read-any-no-hang

pop-up-menu-choose

Purpose:

The function **pop-up-menu-choose** displays a pop-up menu specified by the *pop-up-menu-object* argument. The menu appears on the display screen near the current position of the mouse. You can choose one of the objects on the menu by clicking a mouse button on top of the selected item, or you can move the mouse off the menu.

Syntax:

pop-up-menu-choose **[Function]**

pop-up-menu-object

Remarks:

Once you have made a choice or moved the mouse off the menu, the menu disappears and two values are returned. The first value is the item that you selected, and the second is a keyword that indicates which button you used to select the item. If you did not make a selection and the menu has a default value, the default value and nil are returned; if the menu does not have a default value, both of the values returned are nil.

This function is an extension to Common Lisp.

See Also:

make-pop-up-menu

pop-up-menu-p

Purpose:

The predicate **pop-up-menu-p** tests whether its argument *object* is a pop-up menu. It returns true if *object* is a pop-up menu.

Syntax:

pop-up-menu-p **[Function]**
object

Remarks:

This function is an extension to Common Lisp.

position-x, position-y

Purpose:

The functions **position-x** and **position-y** return the :x- and y-coordinates respectively of a position.

Syntax:

position-x **[Function]**

position

position-y **[Function]**

position

Remarks:

You can use the **setf** macro with the functions **position-x** and **position-y** to set the x- and y-coordinates of a position.

These functions are extensions to Common Lisp.

Examples:

```
> (setq pos (make-position 100 200))
#<position (100,200) 595ba3>
> (position-x pos)
100
> (setf (position-y pos) 300)
300
> pos
#<position (100,300) 595ba3>
```

positionp

Purpose:

The predicate **positionp** tests whether its argument *object* is a position. It returns true if *object* is a position.

Syntax:

positionp **[Function]**

object

Remarks:

This function is an extension to Common Lisp.

Examples:

```
> (positionp (make-position 100 200))
T
> (positionp 7)
NIL
```

read-any, read-any-no-hang

Purpose:

The functions **read-any** and **read-any-no-hang** read either a single character or a single mouse event object from a mouse input stream.

Syntax:

read-any **[Function]**

&optional mouse-input-stream

read-any-no-hang **[Function]**

&optional mouse-input-stream

Remarks:

Note: For the X Window Tool Kit the input functions may return keywords like *:escape*, *:meta-a*, or *:f1* instead of characters.

The argument *mouse-input-stream* specifies a mouse input stream. If this argument is omitted or nil, the mouse input stream that is the value of the function **mouse-input** is used. If the *mouse-input-stream* argument is t, the mouse input stream that is the value of the function **keyboard-input** is used.

If there is no character or mouse event object ready to be input, the function **read-any** waits until a character is typed to the stream or a mouse event occurs on the stream *mouse-input-stream*. In this same situation, the function **read-any-no-hang** returns the value nil without waiting.

These functions are extensions to Common Lisp.

See Also:

keyboard-input
listen-any
make-mouse-input-stream
mouse-event-p
mouse-input
mouse-input-stream-queue-mouse-events-p
peek-any
unread-any

refresh-windows

Purpose:

The function **refresh-windows** runs all pending active region methods and interrupt character methods. Call **refresh-windows** inside the body of a **with-asynchronous-method-allowed** macro to guarantee that interrupts can occurr asynchronously.

Syntax:

refresh-windows **[Function]**

Remarks:

This function is an extension to Common Lisp.

See Also:

with-asynchronous-method-invocation-allowed

region-contains-point-p, region-contains-position-p

Purpose:

The predicates **region-contains-point-p** and **region-contains-position-p** test whether a given position is in a given region.

The predicate **region-contains-point-p** is true if the position whose coordinates are x and y is in the given region.

The predicate **region-contains-position-p** is true when the position *position* is in the given region.

Syntax:

region-contains-point-p **[Function]**

region x y

region-contains-position-p **[Function]**

region position

Remarks:

These functions are extensions to Common Lisp.

Examples:

```
;; Create a region whose origin is (100,100)
;; and whose corner is (400,300).
> (setq reg (make-region :x 100 :y 100 :width 300 :height 200))
#<region 300x200 at (100,100) 596507>
> (region-contains-point-p reg 150 299)
T
> (region-contains-position-p reg (make-position 150 300))
NIL
```

region-corner, region-corner-x, region-corner-y, region-height, region-width, region-origin, region-origin-x, region-origin-y, region-size

Purpose:

Each of these functions returns a component of a region.

Syntax:

region-corner region **[Function]**
&optional result-position

region-corner-x **[Function]**
region

region-corner-y **[Function]**
region

region-height **[Function]**
region

region-width **[Function]**
region

region-origin **[Function]**
region &optional result-position

region-origin-x **[Function]**
region

region-origin-y **[Function]**
region

region-size **[Function]**
region &optional result-extent

Remarks:

You can use the macro **setf** with all these functions.

If a *result-position* argument is given for **region-corner** and **region-origin**, that position is modified to the region's corner position or origin position and returned. Otherwise a new position is created and returned.

If a *result-extent* argument is given for **region-extent**, that extent is modified to the region's extent and then returned. Otherwise a new extent is created and returned.

These functions are extensions to Common Lisp.

Examples:

```
;; Create a region whose origin is (400,500)
;; and whose corner is (480,590).
> (setq r (make-region :x 400 :y 500 :width 80 :height 90))
#<region 80x90 at (400..500) ada243>
> (region-corner r)
#<position (480.590) ada2e3>
> (region-corner-x r)
480
> (region-corner-y r)
590
> (region-height r)
90
> (region-width r)
80
> (region-origin r)
#<position (400,500) adb1db>
> (region-origin-x r)
400
> (region-origin-y r)
500
> (region-size r)
#<extent 80x90 adb253>
```

region-intersection, region-union

Purpose:

The function **region-intersection** returns the region covered in common by all of the given regions. If there is no intersection, it returns nil.

The function **region-union** returns the smallest region that contains all of the supplied regions.

Syntax:

region-intersection **[Function]**

region region &rest regions

region-union **[Function]**

region region &rest regions

Remarks:

These functions are extensions to Common Lisp.

Examples:

```
> (setq r1 (make-region :x 0 :y 0 :width 100 :height 200))
#<region 100x200 at (0,0) 5ac098>
> (setq r2 (make-region :x 50 :y 150 :width 100 :height 100))
#<region 100x100 at (50,150) 5ac0bd>
> (region-union r1 r2)
#<region 150x250 at (0,0) 5ac0ce>
> (region-intersection r1 r2)
#<region 50x50 at (50,150) 5ac0e3>
```

region<, region<=, region=, region/=, region>, region>=

Purpose:

These functions test containment and equality for regions.

The predicate **region<** is true if each argument except the last is contained in the following argument.

The predicate **region<=** is true if each argument except the last is contained in or equals the following argument.

The predicate **region=** is true if every argument is the same region.

The predicate **region/=** is true if no two arguments are the same region.

The predicate **region>** is true if each argument except the last contains the argument that follows it.

The predicate **region>=** is true if each argument except the last contains or is equal to the argument that follows it.

Syntax:

region< **[Function]**

region region &rest regions

region<= **[Function]**

region region &rest regions

region= **[Function]**

region region &rest regions

region/= **[Function]**

region region &rest regions

region> **[Function]**

region region &rest regions

region>= **[Function]**

region region &rest regions

Remarks:

These functions are extensions to Common Lisp.

Examples:

```
> (setq
region1 (make-region :x 0 :y 0 :corner-x 100 :corner-y 100)
region2 (make-region :x 0 :y 0 :corner-x 100 :corner-y 101)
region3 (make-region :x 50 :y 50 :corner-x 200 :corner-y 300)
 region4 (make-region :x 200 :y 300 :corner-x 500 :corner-y 700)
region5 (make-region :x 150 :y 299 :corner-x 500 :corner-y 701)
region6 (make-region :x 150 :y 299 :corner-x 500 :corner-y 700)
)
 #<Region 350x401 at (150,299) 4850CB>
> (region< region1 region2)
 T
> (region< region1 region1)
NIL
> (region<= region1 region2)
T
> (region<= region1 region1)
T
> (region> region2 region1)
 T
> (region> region2 region2)
NIL
> (region>= region2 region1)
T
> (region>= region2 region2)
 T
```

```
;; For region/= to be true, the regions must be all different.
> (region/= region1 region2 region3 region4 region5 region6)
T
> (region/= regionl region2 region3 region4 region5 region6 region1)
NIL
> (region= region1 region1 region1 region1 region1)
T
> (region= region1 region2)
NIL
```

regionp

Purpose:

The predicate **regionp** tests whether its argument *object* is a region. It returns true if *object* is a region.

Syntax:

regionp **[Function]**
object

Remarks:

This function is an extension to Common Lisp.

Examples:

```
> (regionp (make-region :x 0 :y 0 :width 100 :height 200))
T
> (regionp 8)
NIL
```

reshape-viewport

Purpose:

The function **reshape-viewport** moves and reshapes a viewport so that its screen region is the region specified by the keyword arguments.

Syntax:

reshape-viewport **[Function]**

viewport &key :region :x :y
:width :height
:corner-x :corner-y

Remarks:

The keyword arguments are used to specify the new region. All coordinates are given in terms of the root viewport. You must specify enough keyword arguments to identify the region uniquely.

The **:x** and **:y** keyword arguments specify the x- and y-coordinates respectively of the top-left corner of the region.

The **:corner-x** and **:corner-y** keyword arguments specify the x- and y-coordinates respectively of the point just below and to the right of the region.

The **:width** and **:height** keyword arguments specify the width and height respectively of the region.

Moving a viewport also moves all of its descendants.

This function is an extension to Common Lisp.

root-viewport

Purpose:

The function **root-viewport** returns the root viewport.

Syntax:

root-viewport **[Function]**

Remarks:

The root viewport is a viewport onto a special bitmap that requires less memory but has limited capabilities. You cannot modify the bits of this special bitmap in any way without signaling an error.

This function is an extension to Common Lisp.

stream-current-font

Purpose:

The function **stream-current-font** returns the current font of a bitmap output stream.

Syntax:

stream-current-font **[Function]**

bitmap-output-stream

Remarks:

You can use the macro **setf** to modify the stream's current font. The second argument to **setf** must be a font. Changing a stream's current font does not modify the stream's linefeed distance.

This function is an extension to Common Lisp.

Examples:

```
;; Create a 100x200 bitmap.
> (setq btmp (make-bitmap :height 100 :width 200))
#<Bitmap 200x100 5ACEC3>
;; Create a bitmap output stream to that bitmap.
> (setq b-o-s (make-bitmap-output-stream :bitmap btmp))
#<Output-Stream to #<Bitmap 200x100 5ACEC3> 5AD21D>
;; The bitmap output sream's current font is *default-font*.
> (eq (stream-current-font *) *default-font*)
T
```

See Also:

stream-linefeed-distance

stream-draw-circle, stream-draw-line, stream-draw-polyline

Purpose:

The function **stream-draw-circle** draws a circle of radius *radius* around a bitmap output stream's current position.

The function **stream-draw-line** draws a line segment from a bitmap output stream's current position to the position *end*. The bitmap output stream's new current position becomes the position *end*.

The function **stream-draw-polyline** draws a series of connected line segments, starting at a bitmap output stream's current position and then going through each position in a sequence of positions. The current position of the bitmap output stream is left at the final position.

Syntax:

stream-draw-circle **[Function]**

bitmap-output-stream radius
&key :width :operation

stream-draw-line **[Function]**

bitmap-output-stream end
&key :width :operation

stream-draw-polyline **[Function]**

bitmap-output-stream positions
&key :width :operation

Remarks:

If the **:width** keyword argument is specified, it defines the line width that is used for drawing the line segments and circles. For the function stream-draw-circle, the border is drawn so that its outer edge is at the specified radius; the width must be less than or equal to the radius. If this keyword argument is omitted or nil, the default value 1 is used.

The value of the keyword argument **:operation** controls how the bits that are written onto the bitmap are combined with the bits that are already there. If this keyword argument is omitted or nil, the bitmap output stream's operation is used. You can find that operation by using the function **stream-operation**.

These functions are extensions to Common Lisp.

See Also:

draw-circle
draw-line
draw-polyline
stream-operation

stream-linefeed-distance

Purpose:

The function **stream-linefeed-distance** accesses the linefeed distance of a bitmap output stream.

Syntax:

stream-linefeed-distance **[Function]**

bitmap-output-stream

Remarks:

You can use the **setf** macro with this function to modify a bitmap output stream's linefeed distance.

When a bitmap output stream is created, its linefeed distance is the height of the initial font.

A stream's linefeed distance is used when one of the Common Lisp output functions sends a newline character to the bitmap output stream. The bitmap output stream's y-coordinate is incremented by the linefeed distance, and the x-coordinate is set to 0.

This function is an extension to Common Lisp.

stream-operation

Purpose:

The function **stream-operation** returns a bitmap output stream's default **bitblt** operation, which is used in writing characters or figures to the bitmap output stream's bitmap.

Syntax:

stream-operation **[Function]**

bitmap-output-stream

Remarks: You can use the macro **setf** with this function to set a new value. The new value must be an acceptable first argument to the **boole** function. **Note:** For the Allegro Window Tool Kit only the following operations are defined: **boole-1, boole-and, boole-andc1, boole-c1, boole-eqv, boole-ior, boole-orc1, boole-xor.**

This function is an extension to Common Lisp.

Examples:

;; Create a 100x200 bitmap.
> (setq x (make-bitmap :height 100 :width 200))
#<Bitmap 200x100 AD476B>
;; Create a bitmap output stream to that bitmap.
;; Specify an operation.
> (setq b-o-s (make-bitmap-output-stream :bitmap x :operation boole-1))
#<Output-Stream to #<Bitmap 200x100 AD476B> AD54B3>
;; The stream operation of b-o-s is the specified operation.
> (eql (stream-operation b-o-s) boole-1)
T

stream-position, stream-x-position, stream-y-position

Purpose:

These functions return the output position of a bitmap output stream. The output position specifies the next position for writing to the bitmap output stream's bitmap.

The function **stream-position** returns a position.

The function **stream-x-position** returns the x-coordinate of the position.

The function **stream-y-position** returns the y-coordinate of the position.

Syntax:

stream-position **[Function]**

bitmap-output-stream
&optional result-position

stream-x-position **[Function]**

bitmap-output-stream

stream-y-position **[Function]**

bitmap-output-stream

Remarks:

If a *result-position* argument is given for the function **stream-position**, that position is modified to the output position and then returned. Otherwise a new position is created and returned.

You can use the **setf** macro with these functions to modify a bitmap output stream's position.

When a bitmap output stream is created, its stream position is the position whose x-coordinate is 0 and whose y-coordinate is the baseline height of the initial font.

These functions are extensions to Common Lisp.

Examples:

```
;; Create a bitmap and a bitmap output stream.
> (setq b-o-s (make-bitmap-output-stream :width 100 :height 200))
#<Output-Stream to #<Bitmap 100x200 AD75BB> AD8463>
;; Check the initial value.
> (and (= 0 (stream-x-position b-o-s))
(= (font-baseline *default-font*) (stream-y-position b-o-s)))
T
;; Set the x position to a new value.
> (setf (stream-x-position b-o-s) 50)
50
```

string-width

Purpose:

The function **string-width** determines how many bits wide the string *string* is when printed in the font *font*.

Syntax:

string-width **[Function]**
string font

Remarks:

Note: The function **string-width** is **not exported** from the Allegro Window Tool Kit! Please refer to it with **windows::string-width**.

The function may give false results if the string contains any characters that cannot be printed, such as the newline character. The space character is a printable character.

This function is an extension to Common Lisp.

unread-any

Purpose:

The function **unread-any** returns a character or mouse event object to the front of a mouse input stream's queue. The character or mouse event object must be the same object that was last read from the queue.

Syntax:

unread-any **[Function]**

char-or-mouse-event &optional mouse-input-stream

Remarks:

The argument *mouse-input-stream* specifies a mouse input stream. If this argument is omitted or nil, the mouse input stream that is the value of the function **mouse-input** is used. If the *mouse-input-stream* argument is t, the mouse input stream that is the value of the function **keyboard-input** is used.

This function is an extension to Common Lisp.

See Also:

keyboard-input
mouse-input
read-any

viewport-active-p

Purpose:

The function **viewport-active-p** returns true if the viewport *viewport* is active. If the viewport is not active the function returns nil.

Syntax:

viewport-active-p **[Function]**
viewport

Remarks:

This function is an extension to Common Lisp.

See Also:

activate-viewport
deactivate-viewport

viewport-at-point, viewport-at-position

Purpose:

The function **viewport-at-point** interprets *x* and *y* as the coordinates of a point on the screen. It returns as its value the viewport that is displayed at that point on the screen.

The function **viewport-at-position** is identical to **viewport-at-point** except that it is passed a single position argument rather than x- and y-coordinates.

Syntax:

viewport-at-point **[Function]**

x y

viewport-at-position **[Function]**

position

Remarks:

These functions are extensions to Common Lisp.

viewport-bitmap

Purpose:

The function **viewport-bitmap** returns a viewport's underlying bitmap.

Syntax:

viewport-bitmap **[Function]**

viewport

Remarks:

This function is an extension to Common Lisp.

viewport-bitmap-offset, viewport-bitmap-x-offset, viewport-bitmap-y-offset

Purpose:

The function **viewport-bitmap-offset** returns the position that represents the offset (from the bitmap's origin) of a viewport's origin. This offset indicates what part of the bitmap is being displayed in the viewport.

The functions **viewport-bitmap-x-offset** and **viewport-bitmap-y-offset** return the x- and y-coordinates of the offset respectively.

Syntax:

viewport-bitmap-offset **[Function]**
viewport &optional result-position

viewport-bitmap-x-offset **[Function]**
viewport

viewport-bitmap-y-offset **[Function]**
viewport

Remarks:

If a *result-position* argument is given for the function **viewport-bitmap-offset**, that position is modified to the viewport's offset and returned. Otherwise a new position containing the viewport's offset is created and returned.

You can use the **setf** macro with these functions to change the viewport's offset. In particular, modifying **viewport-bitmap-y-offset** causes vertical scrolling.

These functions are extensions to Common Lisp.

viewport-bitmap-region, viewport-screen-region

Purpose:

The function **viewport-bitmap-region** returns a copy of a viewport's bitmap clipping region.

The function **viewport-screen-region** returns a copy of a viewport's screen clipping region.

Syntax:

viewport-bitmap-region **[Function]**
viewport &optional result-region

viewport-screen-region **[Function]**
viewport &optional result-region

Remarks:

If a region is passed as the second argument to these functions, the result is copied into that region object and returned; otherwise a new region is created.

These functions are extensions to Common Lisp.

viewport-children, viewport-parent

Purpose:

The function **viewport-children** returns a list of a viewport's children. The list is in the same order as the children's sibling stack.

The function **viewport-parent** returns a viewport's parent.

Syntax:

viewport-children **[Function]**

viewport

viewport-parent **[Function]**

viewport

Remarks:

The **setf** macro for **viewport-parent** changes a viewport's parent. The viewport is put at the top of the sibling stack of its new parent's children.

You cannot use the **setf** macro with the function **viewport-children**.

These functions are extensions to Common Lisp.

viewportp

Purpose:

The predicate **viewportp** tests whether its argument *object* is a viewport. It returns true if *object* is a viewport.

Syntax:

viewportp **[Function]**
object

Remarks:

This function is an extension to Common Lisp.

Examples:

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.
;; Create a 10x10 bitmap and a viewport onto that bitmap.
;; Note that make-viewport returns two values,
;; the viewport and the bitmap.
> (multiple-value-setq (vwpt btmp) (make-viewport :width 10 :height
10))
#Viewport 10x10 at (0,0) onto #<Bitmap 10x10 40D4E5> 40D51D>
> (viewportp vwpt)
T
> (viewportp btmp)
NIL
```

window-frame

Purpose:

The function **window-frame** returns the window frame that is associated with a given window.

Syntax:

window-frame **[Function]**
window

Remarks:

The window frame can be thought of as a special bitmap; the image that this bitmap displays on the screen is the window's frame. You cannot modify the bits of this special bitmap in any way without signaling an error. You may attach active regions to window frames in the same way that you normally attach active regions to bitmaps.

You cannot use the **setf** macro with this function.

This function is an extension to Common Lisp.

window-inner-border-width, window-outer-border-width

Purpose:

The function **window-inner-border-width** returns the width of the inner border of the window.

The function **window-outer-border-width** returns the width of the outer border of the window.

The window border consists of two strips: the black strip around the edge of the window is the outer border, and the white strip inside the black strip is the inner border.

Syntax:

window-inner-border-width **[Function]**
window

window-outer-border-width **[Function]**
window

Remarks:

The **setf** macro can be used with these functions to modify the widths of the inner and outer borders of a window.

These functions are extensions to Common Lisp.

window-title, window-title-font

Purpose:

The function **window-title** returns the title of a window as a string.

The function **window-title-font** returns the font in which a window's title is displayed.

Syntax:

window-title **[Function]**

window

window-title-font **[Function]**

window

Remarks:

Note: The function **window-title** is **not exported** from the Allegro Window Tool Kit! Please refer to it with **windows::window-title**.

You can use the **setf** macro with the function **window-title** to modify the title of a window. The second argument to **setf** must be a string.

You can use the **setf** macro with the function **window-title-font**. Doing so redraws the title of the window in the new font.

These functions are extensions to Common Lisp.

Examples:

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.
;; Create a window with the title "hello".
> (setq w (make-window :width 100 :height 200 :title "hello"))
#<WINDOW 4AC0AB>
> (windov-title w)
"hello"
;; Note that the title font is *default-font*.
> (eq (window-title-font w) *default-font*)
T
;; Modify the title.
> (setf (window-title w) "new-name")
"new-name"
> (window-title w)
"new-name"
```

window-vertical-scroll-ratio, window-horizontal-scroll-ratio

Purpose:

The functions **window-vertical-scroll-ratio** and **window-horizontal-scroll-ratio** return the vertical and horizontal scroll ratio respectively of a given window.

Syntax:

window-vertical-scroll-ratio **[Function]**

window

window-horizontal-scroll-ratio **[Function]**

window

Remarks:

The scroll ratio is a Common Lisp ratio between 0 and 1. Generally the scroll ratio is the ratio of the current location of the window to the size of the window's underlying bitmap. However, window system developers may redefine the methods for scrolling and for calculating these ratios so that scrolling may be performed over an abstract bitmap or extent.

These functions may be used with the macro **setf** to specify a vertical or horizontal ratio for the given window.

These functions are extensions to Common Lisp.

See Also:

make-window

windowp

Purpose:

The predicate **windowp** tests whether its argument *object* is a window. It returns true if *object* is a window.

Syntax:

windowp **[Function]**

object

Remarks:

This function is an extension to Common Lisp.

Examples:

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.
> (setq w (make-window :width 100 :height 200 :title "hello"))
#< WINDOW 4AC0AB>
> (windowp w)
T
> (windowp 7)
NIL
```

windows-available-p

Purpose:

The function **windows-available-p** checks the Lisp environment for a window system that is capable of supporting the Window Tool Kit.

Syntax:

windows-available-p **[Function]**

Remarks:

The function **windows-available-p** returns as values the display width in pixels and the display height in pixels if a window system can be run from the current process; otherwise the function returns nil. The returned display parameters can be used in subsequent calls to **initialize-windows**.

This function is an extension to Common Lisp.

with-asynchronous-method-invocation-allowed

Purpose:

The macro **with-asynchronous-method-invocation-allowed** allows active region methods and interrupt character methods to occur asynchronously rather than sequentially.

It is only used inside the body of an active region method or an interrupt character method. The *form* arguments are evaluated. Any pending active region methods and any interrupt character methods that would normally be queued until the current method terminated are instead run immediately.

Syntax:

with-asynchronous-method-invocation-allowed [Macro]
{form}*

Remarks:

Normally, all active region and interrupt character methods are executed sequentially. However, sometimes an active region or interrupt character method needs to wait for the action taken by another active region method or interrupt character method to occur. The macro **with-asynchronous-method-invocation-allowed** provides for this.

The *form* arguments are evaluated in an an environment where pending active region methods and interrupt character methods are allowed to run. These methods are executed sequentially with respect to each other unless one of them contains a **with-asynchronous-method-invocation-allowed** form, that contains the expression **(refresh-windows)**. In this case all pending methods are executed before that method terminates.

This macro is an extension to Common Lisp.

See Also:

refresh-windows

with-fast-drawing-environment

Purpose:

The macro **with-fast-drawing-environment** groups display operations. Overhead operations that are required to produce output are executed only once rather than for every display operation in the group.

Syntax:

with-fast-drawing-environment [Macro]
{form}*

Remarks:

The output for the group may not appear on the screen until the macro is exited. Therefore, you should not use this macro to group operations that require user input or that will run for a long time.

This macro is an extension to Common Lisp.

with-mouse-methods-preempted

Purpose:

The **with-mouse-methods-preempted** macro evaluates each of its *form* arguments. While these forms are being evaluated, any active region that is not attached to the *bitmap* argument is disabled. Its methods are not called even if a mouse event occurs inside it.

The *bitmap* argument can also be nil. In this case, all active regions are disabled.

Syntax:

with-mouse-methods-preempted **[Macro]**

bitmap {form}*

Remarks:

The results of evaluating the last form are returned as the results of **with-mouse-methods-preempted**.

This macro is an extension to Common Lisp.

A Allegro Window Tool Kit versus X Window Tool Kit

A.1 Fonts

Be careful, when specifying a font: Allegro and X have different fonts. Use ***default-font*** to make your program independent from the machine type.

A.2 Bitmap-Size

For the Allegro Window Tool Kit the maximum width is **ccl:*screen-width*** and the maximum height is **ccl:*screen-height***.

A.3 Exported Functions

The following functions are not exported from the Allegro Window Tool Kit: **make-bitmap, string-width, window-title.**

The following functions are not exported from the X Window Tool Kit: **draw-line font-name.**

A.4 Input-Functions

For the X Window Tool Kit the input functions may return keywords like *:escape*, *:meta-a*, or *:escape* instead of characters.

A.5 Boole-Constants

For the Allegro Window Tool Kit only the following operations are defined: **boole-1, boole-and, boole-andc1, boole-c1, boole-eqv, boole-ior, boole-orc1, boole-xor.**

A.6 Bitmap-Value

The **setf** macro is not defined for use with function **bitmap-value** for the X Window Tool Kit.

A.7 Move-Mouse

The function **move-mouse** does not work for the Allegro Window Tool Kit.

A.8 Mouse-Cursors

Be careful, when specifying a mouse-cursor: Allegro and X have different mouse-cursors. Use **(current-mouse-cursor)** to make your program independent from the machine type.

B Installation of the Window Tool Kit

B.1 Installation of the Allegro Window Tool Kit

Double-click the icon of the file **"atoolkit.image.anlegen"**. This creates the Allegro Window Tool Kit Image **"atoolkit.image"**. (**Note:** When the **"atoolkit.image"** is started, it first tries to load the file **"init.fasl"** from the home directory. If this fails it tries to load the file **"init.lisp"**) from the home directory.)

When the window system is initialized (see **initialize-windows**) it tries to load the mouse-cursor resource-file **"mouse-cursor.lisp"** from the directory, where **"atoolkit.image"** was created. If you want the window system to load the mouse-cursor resource-file from somewhere else you can either copy that file to **"ccl;A-window-toolkit:mouse-cursor.lisp"** or you can change the value of the variable **user::*windows.dir*** to the directory-path of the resource-file.

B.2 Installation of the X Window Tool Kit

1. Start Common Lisp.
2. Make sure that CLX is loaded; otherwise load it.
3. Load file **"load-xtoolkit.lisp"**.
4. Create an image of the X Window Tool Kit.
   (For KCL this means: **(save "xwtk")**)

RR-90-16
Franz Baader, Werner Nutt: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

RR-90-17
Stephan Busemann
Generalisierte Phasenstrukturgrammatiken und ihre Verwendung zur maschinellen Sprachverarbeitung
114 Seiten

RR-91-01
Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, and Gert Smolka :
On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

RR-91-02
Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt:
The Complexity of Existential Quantification in Concept Languages
22 pages

RR-91-03
B.Hollunder, Franz Baader: Qualifying Number Restrictions in Concept Languages
34 pages

RR-91-04
Harald Trost
X2MORF: A Morphological Component Based on Augmented Two-Level Morphology
19 pages

RR-91-05
Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

RR-91-06
Elisabeth André, Thomas Rist: Synthesizing Illustrated Documents
A Plan-Based Approach
11 pages

RR-91-07
Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist
WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta
RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke
A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel
Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-12
J.Mark Gawron, John Nerbonne, and Stanley Peters
The Absorption Principle and E-Type Anaphora
33 pages

RR-91-13
Gert Smolka
Residuation and Guarded Rules for Constraint Logic Programming
17 pages

RR-91-15
Bernhard Nebel, Gert Smolka
Attributive Description Formalisms ... and the Rest of the World
20 pages

RR-91-16
Stephan Busemann
Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

RR-91-17
Andreas Dengel & Nelson M. Mattos
The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

RR-91-20
Christoph Klauck, Ansgar Bernardi, Ralf Legleitner
FEAT-Rep: Representing Features in CAD/CAM
48 pages

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list c currently available publications can be ordered fro the above address.
The reports are distributed free of charge except otherwise indicated.

DFKI Research Reports

RR-90-01
Franz Baader: Terminological Cycles in KL-ONE-based Knowledge Representation Languages
33 pages

RR-90-02
Hans-Jürgen Bürckert: A Resolution Principle for Clauses with Constraints
25 pages

RR-90-03
Andreas Dengel, Nelson M. Mattos: Integration of Document Representation, Processing and Management
18 pages

RR-90-04
Bernhard Hollunder, Werner Nutt: Subsumption Algorithms for Concept Languages
34 pages

RR-90-05
Franz Baader: A Formal Definition for the Expressive Power of Knowledge Representation Languages
22 pages

RR-90-06
Bernhard Hollunder: Hybrid Inferences in KL-ONE-based Knowledge Representation Systems
21 pages

RR-90-07
Elisabeth André, Thomas Rist: Wissensbasierte Informationspräsentation:
Zwei Beiträge zum Fachgespräch Graphik und KI:
1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 pages

RR-90-08
Andreas Dengel: A Step Towards Understanding Paper Documents
25 pages

RR-90-09
Susanne Biundo: Plan Generation Using a Method of Deductive Program Synthesis
17 pages

RR-90-10
Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann: Concept Logics
26 pages

RR-90-11
Elisabeth André, Thomas Rist: Towards a Plan-Based Synthesis of Illustrated Documents
14 pages

RR-90-12
Harold Boley: Declarative Operations on Nets
43 pages

RR-90-13
Franz Baader: Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles
40 pages

RR-90-14
Franz Schmalhofer, Otto Kühn, Gabriele Schmidt: Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories
20 pages

RR-90-15
Harald Trost: The Application of Two-leve Morphology to Non-concatenative Germar Morphology
13 pages

D-90-06
Andreas Becker: The Window Tool Kit
66 Seiten

D-91-01
Werner Stein , Michael Sintek
Relfun/X - An Experimental Prolog Implementation of Relfun
48 pages

D-91-03
Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause
RFM Manual: Compiling RELFUN into the Relational/Functional Machine
43 pages

D-91-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

D-91-06
Gerd Kamp
Entwurf, vergleichende Beschreibung und Integration eines Arbeitsplanerstellungssystems für Drehteile
130 Seiten

D-91-07
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

D-91-08
Thomas Krause
Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 pages

D-91-09
David Powers and Lary Reeker (Eds)
Proceedings MLNLO'91 - Machine Learning of Natural Language and Ontology
211 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-10
Donald R. Steiner, Jürgen Müller (Eds.)
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-11
Thilo C. Horstmann
Distributed Truth Maintenance
61 pages

D-91-12
Bernd Bachmann
$H^{iera}C_{on}$ - a Knowledge Representation System with Typed Hierarchies and Constraints
75 pages

D-91-13
International Workshop on Terminological Logics
Organizers: Bernhard Nebel, Christof Peltason, Kai von Luck
131 pages

RR-91-23
Prof. Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

RR-91-25
Karin Harbusch, Wolfgang Finkler, Anne Schauder
Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

RR-91-26
M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger
Integrated Plan Generation and Recognition
- A Logic-Based Approach -
17 pages

DFKI Technical Memos

TM-89-01
Susan Holbach-Weber: Connectionist Models and Figurative Speech
27 pages

TM-90-01
Som Bandyopadhyay: Towards an Understanding of Coherence in Multimodal Discourse
18 pages

TM-90-02
Jay C. Weber: The Myth of Domain-Independent Persistence
18 pages

TM-90-03
Franz Baader, Bernhard Hollunder: KRIS: Knowledge Representation and Inference System -System Description-
15 pages

TM-90-04
Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profitlich: Terminological Knowledge Representation: A Proposal for a Terminological Logic
7 pages

TM-91-01
Jana Köhler
Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann
Bidirectional Reasoning of Horn Clause Programs Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt
Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model
20 pages

TM-91-04
Harold Boley
A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel and Rainer Bleisinger
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

DFKI Documents

D-89-01
Michael H. Malburg, Rainer Bleisinger: HYPERBIS: ein betriebliches Hypermedia-Informationssystem
43 Seiten

D-90-01
DFKI Wissenschaftlich-Technischer Jahresbericht 1989
45 pages

D-90-02
Georg Seul: Logisches Programmieren mit Feature -Typen
107 Seiten

D-90-03
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Abschlußbericht des Arbeitspaketes PROD
36 Seiten

D-90-04
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: STEP: Überblick über eine zukünftige Schnittstelle zum Produktdatenaustausch
69 Seiten

D-90-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Formalismus zur Repräsentation von Geo-metrie- und Technologieinformationen als Teil eines Wissensbasierten Produktmodells
66 Seiten

The Window Tool Kit

Andreas Becker

D-90-06
Document