XCreateGC, XCopyGC, XChangeGC, XGetGCValues, XFreeGC, XGContextFromGC, XGCValues – create or free graphics contexts and graphics context structure

GC XCreateGC(*display***,** *d***,** *valuemask***,** *values***)**
Display ***display;*
Drawable *d**;*
unsigned long *valuemask**;*
XGCValues ***values;*

XCopyGC(*display*, *src*, *valuemask*, *dest*)
Display **display*;
GC *src*, *dest*;
unsigned long *valuemask*;

XChangeGC(*display*, *gc*, *valuemask*, *values*)
Display **display*;
GC *gc*;
unsigned long *valuemask*;
XGCValues **values*;

Status XGetGCValues(*display*, *gc*, *valuemask*, *values_return*)
Display **display*;
GC *gc*;
unsigned long *valuemask*;
XGCValues **values_return*;

XFreeGC(*display*, *gc*)
Display **display*;
GC *gc*;

GContext XGContextFromGC(*gc*)
GC *gc*;

d	Specifies the drawable.
dest	Specifies the destination GC.
display	Specifies the connection to the X server.
gc	Specifies the GC.
src	Specifies the components of the source GC.
valuemask	Specifies which components in the GC are to be set, copied, changed, or returned . This argument is the bitwise inclusive OR of zero or more of the valid GC component mask bits.
values	Specifies any values as specified by the valuemask.
values_return	Returns the GC values in the specified **XGCValues** structure.

The XCreateGC function creates a graphics context and returns a GC. The GC can be used with any destination drawable having the same root and depth as the specified drawable. Use with other drawables results in a **BadMatch** error.

XCreateGC can generate **BadAlloc**, **BadDrawable**, **BadFont**, **BadMatch**, **BadPixmap**, and **BadValue** errors.

The **XCopyGC** function copies the specified components from the source GC to the destination GC. The source and destination GCs must have the same root and depth, or a **BadMatch** error results. The valuemask specifies which component to copy, as for **XCreateGC**.

XCopyGC can generate **BadAlloc**, **BadGC**, and **BadMatch** errors.

The **XChangeGC** function changes the components specified by valuemask for the specified GC. The values argument contains the values to be set. The values and restrictions are the same as for **XCreateGC**. Changing the clip-mask overrides any previous **XSetClipRectangles** request on the context. Changing the dash-offset or dash-list overrides any previous **XSetDashes** request on the context. The order in which components are verified and altered is server dependent. If an error is generated, a subset of the components may have been altered.

XChangeGC can generate **BadAlloc**, **BadFont**, **BadGC**, **BadMatch**, **BadPixmap**, and **BadValue** errors.

The **XGetGCValues** function returns the components specified by valuemask for the specified GC. If the valuemask contains a valid set of GC mask bits (**GCFunction**, **GCPlaneMask**, **GCForeground**, **GCBackground**, **GCLineWidth**, **GCLineStyle**, **GCCapStyle**, **GCJoinStyle**, **GCFillStyle**, **GCFillRule**, **GCTile**, **GCStipple**, **GCTileStipXOrigin**, **GCTileStipYOrigin**, **GCFont**, **GCSubwindowMode**, **GCGraphicsExposures**, **GCClipXOrigin**, **GCCLipYOrigin**, **GCDashOffset**, or **GCArcMode**) and no error occurs, **XGetGCValues** sets the requested components in values_return and returns a nonzero status. Otherwise, it returns a zero status. Note that the clip-mask and dash-list (represented by the **GCClipMask** and **GCDashList** bits, respectively, in the valuemask) cannot be requested. Also note that an invalid resource ID (with one or more of the three most significant bits set to 1) will be returned for **GCFont**, **GCTile**, and **GCStipple** if the component has never been explicitly set by the client.

The **XFreeGC** function destroys the specified GC as well as all the associated storage.

XFreeGC can generate a **BadGC** error.

The XGCValues structure contains:

/* GC attribute value mask bits */

lw(.5i) lw(2.5i) lw(.75i). #define T{ **GCFunction** T} T{ (1L<<0) T} #define T{ **GCPlaneMask** T} T{ (1L<<1) T} #define T{ **GCForeground** T} T{ (1L<<2) T} #define T{ **GCBackground** T} T{ (1L<<3) T} #define T{ **GCLineWidth** T} T{ (1L<<4) T} #define T{ **GCLineStyle** T} T{ (1L<<5) T} #define T{ **GCCapStyle** T} T{ (1L<<6) T} #define T{ **GCJoinStyle** T} T{ (1L<<7) T} #define T{ **GCFillStyle** T} T{ (1L<<8) T} #define T{ **GCFillRule** T} T{ (1L<<9) T} #define T{ **GCTile** T} T{ (1L<<10) T} #define T{ **GCStipple** T} T{ (1L<<11) T} #define T{ **GCTileStipXOrigin** T} T{ (1L<<12) T} #define T{ **GCTileStipYOrigin** T} T{ (1L<<13) T} #define T{ **GCFont** T} T{ (1L<<14) T} #define T{ **GCSubwindowMode** T} T{ (1L<<15) T} #define T{ **GCGraphicsExposures** T} T{ (1L<<16) T} #define T{ **GCClipXOrigin** T} T{ (1L<<17) T} #define T{ **GCClipYOrigin** T} T{ (1L<<18) T} #define T{ **GCClipMask** T} T{ (1L<<19) T} #define T{ **GCDashOffset** T} T{ (1L<<20) T} #define T{ **GCDashList** T} T{ (1L<<21) T} #define T{ **GCArcMode** T} T{ (1L<<22) T}

/* Values */

```
typedef struct {
        int function;                   /* logical operation */
        unsigned long plane_mask;       /* plane mask */
        unsigned long foreground;       /* foreground pixel */
        unsigned long background;       /* background pixel */
        int line_width;                 /* line width (in pixels) */
        int line_style;                 /* LineSolid, LineOnOffDash, LineDoubleDash */
        int cap_style;                  /* CapNotLast, CapButt, CapRound, CapProjecting */
        int join_style;                 /* JoinMiter, JoinRound, JoinBevel */
        int fill_style;                 /* FillSolid, FillTiled, FillStippled FillOpaqueStippled*/
        int fill_rule;                  /* EvenOddRule, WindingRule */
        int arc_mode;                   /* ArcChord, ArcPieSlice */
        Pixmap tile;                    /* tile pixmap for tiling operations */
```

```
        Pixmap stipple;                 /* stipple 1 plane pixmap for stippling */
        int ts_x_origin;                /* offset for tile or stipple operations */
        int ts_y_origin;
        Font font;                      /* default text font for text operations */
        int subwindow_mode;             /* ClipByChildren, IncludeInferiors */
        Bool graphics_exposures;        /* boolean, should exposures be generated */
        int clip_x_origin;              /* origin for clipping */
        int clip_y_origin;
        Pixmap clip_mask;               /* bitmap clipping; other calls for rects */
        int dash_offset;                /* patterned/dashed line information */
        char dashes;
} XGCValues;
```

The function attributes of a GC are used when you update a section of a drawable (the destination) with bits from somewhere else (the source). The function in a GC defines how the new destination bits are to be computed from the source bits and the old destination bits. **GXcopy** is typically the most useful because it will work on a color display, but special applications may use other functions, particularly in concert with particular planes of a color display. The 16 GC functions, defined in <**X11/X.h**>, are:

lw(1.5i) cw(.5i) lw(2i). _

Function Name ValueOperation

_

T{ GXclear T} T{ 0x0 T} T{ 0 T} T{ **GXand** T} T{ 0x1 T} T{ src AND dst T} T{ **GXandReverse** T} T{ 0x2 T} T{ src AND NOT dst T} T{ **GXcopy** T} T{ 0x3 T} T{ src T} T{ **GXandInverted** T} T{ 0x4 T} T{ (NOT src) AND dst T} T{ **GXnoop** T} T{ 0x5 T} T{ dst T} T{ **GXxor** T} T{ 0x6 T} T{ src XOR dst T} T{ **GXor** T} T{ 0x7 T} T{ src OR dst T} T{ **GXnor** T} T{ 0x8 T} T{ (NOT src) AND (NOT dst) T} T{ **GXequiv** T} T{ 0x9 T} T{ (NOT src) XOR dst T} T{ **GXinvert** T} T{ 0xa T} T{ NOT dst T} T{ **GXorReverse** T} T{ 0xb T} T{ src OR (NOT dst) T} T{ **GXcopyInverted** T} T{ 0xc T} T{ NOT src T} T{ **GXorInverted** T} T{ 0xd T} T{ (NOT src) OR dst T} T{ **GXnand** T} T{ 0xe T} T{ (NOT src) OR (NOT dst) T} T{ **GXset** T} T{ 0xf T} T{ 1 T}

_

Many graphics operations depend on either pixel values or planes in a GC. The planes attribute is of type long, and it specifies which planes of the destination are to be modified, one bit per plane. A monochrome display has only one plane and will be the least significant bit of the word. As planes are added to the display hardware, they will occupy more significant bits in the plane mask.

In graphics operations, given a source and destination pixel, the result is computed bitwise on corresponding bits of the pixels. That is, a Boolean operation is performed in each bit plane. The plane_mask restricts the operation to a subset of planes. A macro constant **AllPlanes** can be used to refer to all planes of the screen simultaneously. The result is computed by the following:

((src FUNC dst) AND plane-mask) OR (dst AND (NOT plane-mask))

Range checking is not performed on the values for foreground, background, or plane_mask. They are simply truncated to the appropriate number of bits. The line-width is measured in pixels and either can be greater than or equal to one (wide line) or can be the special value zero (thin line).

Wide lines are drawn centered on the path described by the graphics request. Unless otherwise specified by the join-style or cap-style, the bounding box of a wide line with endpoints [x1, y1], [x2, y2] and width w is a rectangle with vertices at the following real coordinates:

[x1-(w*sn/2), y1+(w*cs/2)], [x1+(w*sn/2), y1-(w*cs/2)],
[x2-(w*sn/2), y2+(w*cs/2)], [x2+(w*sn/2), y2-(w*cs/2)]

Here sn is the sine of the angle of the line, and cs is the cosine of the angle of the line. A pixel is part of the line and so is drawn if the center of the pixel is fully inside the bounding box (which is viewed as having infinitely thin edges). If the center of the pixel is exactly on the bounding box, it is part of the line if and only if the interior is immediately to its right (x increasing direction). Pixels with centers on a horizontal edge are a special case and are part of the line if and only if the interior or the boundary is immediately below (y increasing direction) and the interior or the boundary is immediately to the right (x increasing direction).

Thin lines (zero line-width) are one-pixel-wide lines drawn using an unspecified, device-dependent algorithm. There are only two constraints on this algorithm.

1. If a line is drawn unclipped from [x1,y1] to [x2,y2] and if another line is drawn unclipped from [x1+dx,y1+dy] to [x2+dx,y2+dy], a point [x,y] is touched by drawing the first line if and only if the point [x+dx,y+dy] is touched by drawing the second line.
2. The effective set of points comprising a line cannot be affected by clipping. That is, a point is touched in a clipped line if and only if the point lies inside the clipping region and the point would be touched by the line when drawn unclipped.

A wide line drawn from [x1,y1] to [x2,y2] always draws the same pixels as a wide line drawn from [x2,y2] to [x1,y1], not counting cap-style and join-style. It is recommended that this property be true for thin lines, but this is not required. A line-width of zero may differ from a line-width of one in which pixels are drawn. This permits the use of many manufacturers' line drawing hardware, which may run many times faster than the more precisely specified wide lines.

In general, drawing a thin line will be faster than drawing a wide line of width one. However, because of their different drawing algorithms, thin lines may not mix well aesthetically with wide lines. If it is desirable to obtain precise and uniform results across all displays, a client should always use a line-width of one rather than a line-width of zero.

The line-style defines which sections of a line are drawn:

lw(1.3i) lw(4.5i). T{ **LineSolid** T} T{ The full path of the line is drawn. T}

T{ **LineDoubleDash** T} T{ The full path of the line is drawn, but the even dashes are filled differently from the odd dashes (see fill-style) with **CapButt** style used where even and odd dashes meet. T}

T{ **LineOnOffDash** T} T{ Only the even dashes are drawn, and cap-style applies to all internal ends of the individual dashes, except **CapNotLast** is treated as **CapButt**. T}

The cap-style defines how the endpoints of a path are drawn:

lw(1.3i) lw(4.5i). T{ **CapNotLast** T} T{ This is equivalent to **CapButt** except that for a line-width of zero the final endpoint is not drawn. T}

T{ **CapButt** T} T{ The line is square at the endpoint (perpendicular to the slope of the line) with no projection beyond. T}

T{ **CapRound** T} T{ The line has a circular arc with the diameter equal to the line-width, centered on the endpoint. (This is equivalent to **CapButt** for line-width of zero). T}

T{ **CapProjecting** T} T{ The line is square at the end, but the path continues beyond the endpoint for a distance equal to half the line-width. (This is equivalent to **CapButt** for line-width of zero). T}

The join-style defines how corners are drawn for wide lines:

lw(1.3i) lw(4.5i). T{ **JoinMiter** T} T{ The outer edges of two lines extend to meet at an angle. However, if the angle is less than 11 degrees, then a **JoinBevel** join-style is used instead. T}

T{ **JoinRound** T} T{ The corner is a circular arc with the diameter equal to the line-width, centered on the joinpoint. T}

T{ **JoinBevel** T} T{ The corner has **CapButt** endpoint styles with the triangular notch filled. T}

For a line with coincident endpoints (x1=x2, y1=y2), when the cap-style is applied to both endpoints, the semantics depends on the line-width and the cap-style:

lw(1.3i) lw(.5i) lw(4i). T{ **CapNotLast** T} T{ thin T} T{ The results are device dependent, but the desired effect is that nothing is drawn. T}

T{ **CapButt** T} T{ thin T} T{ The results are device dependent, but the desired effect is that a single pixel is drawn. T}

T{ **CapRound** T} T{ thin T} T{ The results are the same as for **CapButt**/thin. T}

T{ **CapProjecting** T} T{ thin T} T{ The results are the same as for **CapButt**/thin. T}

T{ **CapButt** T} T{ wide T} T{ Nothing is drawn. T}

T{ **CapRound** T} T{ wide T} T{ The closed path is a circle, centered at the endpoint, and with the diameter equal to the line-width. T}

T{ **CapProjecting** T} T{ wide T} T{ The closed path is a square, aligned with the coordinate axes, centered at the endpoint, and with the sides equal to the line-width. T}

For a line with coincident endpoints (x1=x2, y1=y2), when the join-style is applied at one or both endpoints, the effect is as if the line was removed from the overall path. However, if the total path consists of or is reduced to a single point joined with itself, the effect is the same as when the cap-style is applied at both endpoints.

The tile/stipple represents an infinite two-dimensional plane, with the tile/stipple replicated in all dimensions. When that plane is superimposed on the drawable for use in a graphics operation, the upper-left corner of some instance of the tile/stipple is at the coordinates within the drawable specified by the tile/stipple origin. The tile/stipple and clip origins are interpreted relative to the origin of whatever destination drawable is specified in a graphics request. The tile pixmap must have the same root and depth as the GC, or a **BadMatch** error results. The stipple pixmap must have depth one and must have the same root as the GC, or a **BadMatch** error results. For stipple operations where the fill-style is **FillStippled** but not **FillOpaqueStippled**, the stipple pattern is tiled in a single plane and acts as an additional clip mask to be ANDed with the clip-mask. Although some sizes may be faster to use than others, any size pixmap can be used for tiling or stippling.

The fill-style defines the contents of the source for line, text, and fill requests. For all text and fill requests (for example, **XDrawText**, **XDrawText16**, **XFillRectangle**, **XFillPolygon**, and **XFillArc**); for line requests with line-style **LineSolid** (for example, **XDrawLine**, **XDrawSegments**, **XDrawRectangle**, **XDrawArc**); and for the even dashes for line requests with line-style **LineOnOffDash** or **LineDoubleDash**, the following apply:

lw(1.8i) lw(4i). T{ **FillSolid** T} T{ Foreground T}

T{ **FillTiled** T} T{ Tile T}

T{ **FillOpaqueStippled** T} T{ A tile with the same width and height as stipple, but with background everywhere stipple has a zero and with foreground everywhere stipple has a one T}

T{ **FillStippled** T} T{ Foreground masked by stipple T}

When drawing lines with line-style **LineDoubleDash**, the odd dashes are controlled by the fill-style in the following manner:

lw(1.8i) lw(4i). T{ **FillSolid** T} T{ Background T}

T{ **FillTiled** T} T{ Same as for even dashes T}

T{ **FillOpaqueStippled** T} T{ Same as for even dashes T}

T{ **FillStippled** T} T{ Background masked by stipple T}

Storing a pixmap in a GC might or might not result in a copy being made. If the pixmap is later used as the destination for a graphics request, the change might or might not be reflected in the GC. If the pixmap is used simultaneously in a graphics request both as a destination and as a tile or stipple, the results are undefined.

For optimum performance, you should draw as much as possible with the same GC (without changing its components). The costs of changing GC components relative to using different GCs depend on the display hardware and the server implementation. It is quite likely that some amount of GC information will be cached in display hardware and that such hardware can only cache a small number of GCs.

The dashes value is actually a simplified form of the more general patterns that can be set with **XSetDashes**. Specifying a value of N is equivalent to specifying the two-element list [N, N] in **XSetDashes**. The value must be nonzero, or a **BadValue** error results.

The clip-mask restricts writes to the destination drawable. If the clip-mask is set to a pixmap, it must have depth one and have the same root as the GC, or a **BadMatch** error results. If clip-mask is set to **None**, the pixels are always drawn regardless of the clip origin. The clip-mask also can be set by calling the **XSetClipRectangles** or **XSetRegion** functions. Only pixels where the clip-mask has a bit set to 1 are drawn. Pixels are not drawn outside the area covered by the clip-mask or where the clip-mask has a bit set to 0. The clip-mask affects all graphics requests. The clip-mask does not clip sources. The clip-mask origin is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

You can set the subwindow-mode to **ClipByChildren** or **IncludeInferiors**. For **ClipByChildren**, both source and destination windows are additionally clipped by all viewable **InputOutput** children. For **IncludeInferiors**, neither source nor destination window is clipped by inferiors. This will result in including subwindow contents in the source and drawing through subwindow boundaries of the destination. The use of **IncludeInferiors** on a window of one depth with mapped inferiors of differing depth is not illegal, but the semantics are undefined by the core protocol.

The fill-rule defines what pixels are inside (drawn) for paths given in **XFillPolygon** requests and can be set to **EvenOddRule** or **WindingRule**. For **EvenOddRule**, a point is inside if an infinite ray with the point as origin crosses the path an odd number of times. For **WindingRule**, a point is inside if an infinite ray with the point as origin crosses an unequal number of clockwise and counterclockwise directed path segments. A clockwise directed path segment is one that crosses the ray from left to right as observed from the point. A counterclockwise segment is one that crosses the ray from right to left as observed from the point. The case where a directed line segment is coincident with the ray is uninteresting because you can simply choose a different ray that is not coincident with a segment.

For both **EvenOddRule** and **WindingRule**, a point is infinitely small, and the path is an infinitely thin line. A pixel is inside if the center point of the pixel is inside and the center point is not on the boundary. If the center point is on the boundary, the pixel is inside if and only if the polygon interior is immediately to its right (x increasing direction). Pixels with centers on a horizontal edge are a special case and are inside if and only if the polygon interior is immediately below (y increasing direction).

The arc-mode controls filling in the **XFillArcs** function and can be set to **ArcPieSlice** or **ArcChord**. For **ArcPieSlice**, the arcs are pie-slice filled. For **ArcChord**, the arcs are chord filled.

The graphics-exposure flag controls **GraphicsExpose** event generation for **XCopyArea** and **XCopyPlane** requests (and any similar requests defined by extensions).

BadAlloc The server failed to allocate the requested resource or server memory. **BadDrawable** A value for a Drawable argument does not name a defined Window or Pixmap. **BadFont** A value for a Font or GContext argument does not name a defined Font. **BadGC** A value for a GContext argument does not name a defined GContext. **BadMatch** An **InputOnly** window is used as a Drawable. **BadMatch** Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request. **BadPixmap** A value for a Pixmap argument does not name a defined Pixmap. **BadValue** Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

AllPlanes(3X11), XCopyArea(3X11), XCreateRegion(3X11), XDrawArc(3X11), XDrawLine(3X11), XDrawRectangle(3X11), XDrawText(3X11), XFillRectangle(3X11), XQueryBestSize(3X11), XSetArcMode(3X11), XSetClipOrigin(3X11), XSetFillStyle(3X11), XSetFont(3X11), XSetLineAttributes(3X11), XSetState(3X11), XSetTile(3X11)
Xlib – C Language X Interface