NAME

X11::Protocol − Perl module for the X Window System Protocol, version 11

SYNOPSIS

```
  use X11::Protocol;
  $x = X11::Protocol−>new();
  $win = $x−>new_rsrc;
  $x−>CreateWindow($win, $x−>root, 'InputOutput',
                   $x−>root_depth, 'CopyFromParent',
                   ($x_coord, $y_coord), $width,
                   $height, $border_w);
  ...
```

DESCRIPTION

X11::Protocol is a client-side interface to the X11 Protocol (see X(1) for information about X11), allowing perl programs to display windows and graphics on X11 servers.

A full description of the protocol is beyond the scope of this documentation; for complete information, see the *X Window System Protocol, X Version 11*, available as Postscript or *roff source from `ftp://ftp.x.org`, or *Volume 0: X Protocol Reference Manual* of O'Reilly & Associates's series of books about X (ISBN 1−56592−083−X, `http://www.oreilly.com`), which contains most of the same information.

DISCLAIMER

"The protocol contains many management mechanisms that are not intended for normal applications. Not all mechanisms are needed to build a particular user interface. It is important to keep in mind that the protocol is intended to provide mechanism, not policy." — Robert W. Scheifler

BASIC METHODS

new

```
  $x = X11::Protocol−>new();
  $x = X11::Protocol−>new($display_name);
  $x = X11::Protocol−>new($connection);
  $x = X11::Protocol−>new($display_name, [$auth_type, $auth_data]);
  $x = X11::Protocol−>new($connection, [$auth_type, $auth_data]);
```

Open a connection to a server. `$display_name` should be an X display name, of the form 'host:display_num.screen_num'; if no arguments are supplied, the contents of the DISPLAY environment variable are used. Alternatively, a pre-opened connection, of one of the X11::Protocol::Connection classes (see X11::Protocol::Connection, X11::Protocol::Connection::FileHandle, X11::Protocol::Connection::Socket, X11::Protocol::Connection::UNIXFH, X11::Protocol::Connection::INETFH, X11::Protocol::Connection::UNIXSocket, X11::Protocol::Connection::INETSocket) can be given. The authorization data is obtained using X11::Auth or the second argument. If the display is specified by `$display_name`, rather than `$connection`, a **choose_screen()** is also performed, defaulting to screen 0 if the '.screen_num' of the display name is not present. Returns the new protocol object.

new_rsrc

```
  $x−>new_rsrc;
```

Returns a new resource identifier. A unique resource ID is required for every object that the server creates on behalf of the client: windows, fonts, cursors, etc. (IDs are chosen by the client instead of the server for efficiency — the client doesn't have to wait for the server to acknowledge the creation before starting to use the object).

Note that the total number of available resource IDs, while large, is finite. Beginning from the establishment of a connection, resource IDs are allocated sequentially from a range whose size is server dependent (commonly 2**21, about 2 million). If this limit is reached and the server does not support the XC_MISC extension, subsequent calls to new_rsrc will croak. If the server does support this extension, the module will attempt to request a new range of free IDs from the server. This should allow the program to continue, but it is an imperfect solution, as over time the set of available IDs may fragment, requiring increasingly frequent round-trip range requests from the server. For long-running programs, the best approach may be to keep track of free IDs as resources are destroyed. In the current version, however, no special support is provided for this.

handle_input

```
$x−>handle_input;
```

Get one chunk of information from the server, and do something with it. If it's an error, handle it using the protocol object's handler ('error_handler' — default is kill the program with an explanatory message). If it's an event, pass it to the chosen event handler, or put it in a queue if the handler is 'queue'. If it's a reply to a request, save using the object's 'replies' hash for further processing.

atom_name

```
$name = $x−>atom_name($atom);
```

Return the string corresponding to the atom `$atom`. This is similar to the GetAtomName request, but caches the result for efficiency.

atom

```
$atom = $x−>atom($name);
```

The inverse operation; Return the (numeric) atom corresponding to `$name`. This is similar to the InternAtom request, but caches the result.

choose_screen

```
$x−>choose_screen($screen_num);
```

Indicate that you prefer to use a particular screen of the display. Per-screen information, such as 'root', 'width_in_pixels', and 'white_pixel' will be made available as

```
$x−>{'root'}
```

instead of

```
$x−>{'screens'}[$screen_num]{'root'}
```

SYMBOLIC CONSTANTS

Generally, symbolic constants used by the protocol, like 'CopyFromParent' or 'PieSlice' are passed to methods as strings, and converted into numbers by the module. Their names are the same as those in the protocol specification, including capitalization, but with hyphens ('−') changed to underscores ('_') to look more perl-ish. If you want to do the conversion yourself for some reason, the following methods are available:

num

```
$num = $x−>num($type, $str)
```

Given a string representing a constant and a string specifying what type of constant it is, return the corresponding number. `$type` should be a name like 'VisualClass' or 'GCLineStyle'. If the name is not recognized, it is returned intact.

interp

```
$name = $x−>interp($type, $num)
```

The inverse operation; given a number and string specifying its type, return a string representing the constant.

You can disable **interp()** and the module's internal interpretation of numbers by setting `$x`−>{'do_interp'} to zero. Of course, this isn't very useful, unless you have you own definitions for all the constants.

Here is a list of available constant types:

```
AccessMode, AllowEventsMode, AutoRepeatMode, BackingStore,
BitGravity, Bool, ChangePropertyMode, CirculateDirection,
CirculatePlace, Class, ClipRectangleOrdering, CloseDownMode,
ColormapNotifyState, CoordinateMode, CrossingNotifyDetail,
CrossingNotifyMode, DeviceEvent, DrawDirection, Error, EventMask,
Events, FocusDetail, FocusMode, GCArcMode, GCCapStyle, GCFillRule,
GCFillStyle, GCFunction, GCJoinStyle, GCLineStyle, GCSubwindowMode,
GrabStatus, HostChangeMode, HostFamily, ImageFormat,
InputFocusRevertTo, KeyMask, LedMode, MapState, MappingChangeStatus,
MappingNotifyRequest, PointerEvent, PolyShape, PropertyNotifyState,
Request, ScreenSaver, ScreenSaverAction, Significance, SizeClass,
StackMode, SyncMode, VisibilityState, VisualClass, WinGravity
```

SERVER INFORMATION

At connection time, the server sends a large amount of information about itself to the client. This information is stored in the protocol object for future reference. It can be read directly, like

```
$x->{'release_number'}
```

or, for object oriented True Believers, using a method:

```
$x->release_number
```

The method method also has a one argument form for setting variables, but it isn't really useful for some of the more complex structures.

Here is an example of what the object's information might look like:

```
'connection' => X11::Connection::UNIXSocket(0x814526fd),
'byte_order' => 'l',
'protocol_major_version' => 11,
'protocol_minor_version' => 0,
'authorization_protocol_name' => 'MIT-MAGIC-COOKIE-1',
'release_number' => 3110,
'resource_id_base' => 0x1c000002,
'motion_buffer_size' => 0,
'maximum_request_length' => 65535, # units of 4 bytes
'image_byte_order' => 'LeastSiginificant',
'bitmap_bit_order' => 'LeastSiginificant',
'bitmap_scanline_unit' => 32,
'bitmap_scanline_pad' => 32,
'min_keycode' => 8,
'max_keycode' => 134,
'vendor' => 'The XFree86 Project, Inc',
'pixmap_formats' => {1 => {'bits_per_pixel' => 1,
                           'scanline_pad' => 32},
                     8 => {'bits_per_pixel' => 8,
                           'scanline_pad' => 32}},
'screens' => [{'root' => 43, 'width_in_pixels' => 800,
               'height_in_pixels' => 600,
               'width_in_millimeters' => 271,
               'height_in_millimeters' => 203,
               'root_depth' => 8,
               'root_visual' => 34,
               'default_colormap' => 33,
               'white_pixel' => 0, 'black_pixel' => 1,
               'min_installed_maps' => 1,
               'max_installed_maps' => 1,
               'backing_stores' => 'Always',
               'save_unders' => 1,
               'current_input_masks' => 0x58003d,
               'allowed_depths' =>
                  [{'depth' => 1, 'visuals' => []},
                   {'depth' => 8, 'visuals' => [
                      {'visual_id' => 34, 'blue_mask' => 0,
                       'green_mask' => 0, 'red_mask' => 0,
                       'class' => 'PseudoColor',
                       'bits_per_rgb_value' => 6,
                       'colormap_entries' => 256},
                      {'visual_id' => 35, 'blue_mask' => 0xc0,
                       'green_mask' => 0x38, 'red_mask' => 0x7,
                       'class' => 'DirectColor',
                       'bits_per_rgb_value' => 6,
                       'colormap_entries' => 8}, ...]}]],
'visuals' => {34 => {'depth' => 8, 'class' => 'PseudoColor',
                     'red_mask' => 0, 'green_mask' => 0,
                     'blue_mask'=> 0, 'bits_per_rgb_value' => 6,
                     'colormap_entries' => 256},
              35 => {'depth' => 8, 'class' => 'DirectColor',
                     'red_mask' => 0x7, 'green_mask' => 0x38,
                     'blue_mask'=> 0xc0, 'bits_per_rgb_value' => 6,
                     'colormap_entries' => 8}, ...}
'error_handler' => &\X11::Protocol::default_error_handler,
'event_handler' => sub {},
'do_interp' => 1
```

REQUESTS

request

```
$x->request('CreateWindow', ...);
$x->req('CreateWindow', ...);
$x->CreateWindow(...);
```

Send a protocol request to the server, and get the reply, if any. For names of and information about individual requests, see below and/or the protocol reference manual.

robust_req

```
$x->robust_req('CreateWindow', ...);
```

Like **request()**, but if the server returns an error, return the error information rather than calling the error handler (which by default just croaks). If the request succeeds, returns an array reference containing whatever **request()** would have. Otherwise, returns the error type, the major and minor opcodes of the failed request, and the extra error information, if any. Note that even if the request normally wouldn't have a reply, this method still has to wait for a round-trip time to see whether an error occurred. If you're concerned about performance, you should design your error handling to be asynchronous.

add_reply

```
$x->add_reply($sequence_num, \$var);
```

Add a stub for an expected reply to the object's 'replies' hash. When a reply numbered `$sequence_num` comes, it will be stored in `$var`.

delete_reply

```
$x->delete_reply($sequence_num);
```

Delete the entry in 'replies' for the specified reply. (This should be done after the reply is received).

send

```
$x->send('CreateWindow', ...);
```

Send a request, but do not wait for a reply. You must handle the reply, if any, yourself, using **add_reply()**, **handle_input()**, **delete_reply()**, and **unpack_reply()**, generally in that order.

unpack_reply

```
$x->unpack_reply('GetWindowAttributes', $data);
```

Interpret the raw reply data `$data`, according to the reply format for the named request. Returns data in the same format as `request($request_name, ...)`.

This section includes only a short calling summary for each request; for full descriptions, see the protocol standard. Argument order is usually the same as listed in the spec, but you generally don't have to pass lengths of strings or arrays, since perl keeps track. Symbolic constants are generally passed as strings. Most replies are returned as lists, but when there are many values, a hash is used. Lists usually come last; when there is more than one, each is passed by reference. In lists of multi-part structures, each element is a list ref. Parenthesis are inserted in arg lists for clarity, but are optional. Requests are listed in order by major opcode, so related requests are usually close together. Replies follow the '=>'.

```
$x->CreateWindow($wid, $parent, $class, $depth, $visual, ($x, $y),
                 $width, $height, $border_width,
                 'attribute' => $value, ...)

$x->ChangeWindowAttributes($window, 'attribute' => $value, ...)

$x->GetWindowAttributes($window)
=>
('backing_store' => $backing_store, ...)
```

This is an example of a return value that is meant to be assigned to a hash.

```
$x->DestroyWindow($win)

$x->DestroySubwindows($win)

$x->ChangeSaveSet($window, $mode)

$x->ReparentWindow($win, $parent, ($x, $y))

$x->MapWindow($win)

$x->MapSubwindows($win)

$x->UnmapWindow($win)

$x->UnmapSubwindows($win)

$x->ConfigureWindow($win, 'attribute' => $value, ...)

$x->CirculateWindow($win, $direction)
```

Note that this request actually circulates the subwindows of `$win`, not the window itself.

```
$x->GetGeometry($drawable)
=>
('root' => $root, ...)

$x->QueryTree($win)
=>
($root, $parent, @kids)
```

```
$x->InternAtom($name, $only_if_exists)
=>
$atom

$x->GetAtomName($atom)
=>
$name

$x->ChangeProperty($window, $property, $type, $format, $mode, $data)

$x->DeleteProperty($win, $atom)

$x->GetProperty($window, $property, $type, $offset, $length, $delete)
=>
($value, $type, $format, $bytes_after)
```

Notice that the value comes first, so you can easily ignore the rest.

```
$x->ListProperties($window)
=>
(@atoms)

$x->SetSelectionOwner($selection, $owner, $time)

$x->GetSelectionOwner($selection)
=>
$owner

$x->ConvertSelection($selection, $target, $property, $requestor, $time)

$x->SendEvent($destination, $propagate, $event_mask, $event)
```

The `$event` argument should be the result of a **pack_event()** (see "EVENTS")

```
$x->GrabPointer($grab_window, $owner_events, $event_mask,
                $pointer_mode, $keyboard_mode, $confine_to,
                $cursor, $time)
=>
$status

$x->UngrabPointer($time)

$x->GrabButton($modifiers, $button, $grab_window, $owner_events,
               $event_mask, $pointer_mode, $keyboard_mode,
               $confine_to, $cursor)

$x->UngrabButton($modifiers, $button, $grab_window)

$x->ChangeActivePointerGrab($event_mask, $cursor, $time)

$x->GrabKeyboard($grab_window, $owner_events, $pointer_mode,
                 $keyboard_mode, $time)
=>
$status

$x->UngrabKeyboard($time)
```

```
$x->GrabKey($key, $modifiers, $grab_window, $owner_events,
            $pointer_mode, $keyboard_mode)

$x->UngrabKey($key, $modifiers, $grab_window)

$x->AllowEvents($mode, $time)

$x->GrabServer

$x->UngrabServer

$x->QueryPointer($window)
=>
('root' => $root, ...)

$x->GetMotionEvents($start, $stop, $window)
=>
([$time, ($x, $y)], [$time, ($x, $y)], ...)

$x->TranslateCoordinates($src_window, $dst_window, $src_x, $src_y)
=>
($same_screen, $child, $dst_x, $dst_y)

$x->WarpPointer($src_window, $dst_window, $src_x, $src_y, $src_width,
                $src_height, $dst_x, $dst_y)

$x->SetInputFocus($focus, $revert_to, $time)

$x->GetInputFocus
=>
($focus, $revert_to)

$x->QueryKeymap
=>
$keys
```

`$keys` is a bit vector, so you should use **vec()** to read it.

```
$x->OpenFont($fid, $name)

$x->CloseFont($font)

$x->QueryFont($font)
=>
('min_char_or_byte2' => $min_char_or_byte2,
 ...,
 'min_bounds' =>
 [$left_side_bearing, $right_side_bearing, $character_width, $ascent,
  $descent, $attributes],
 ...,
 'char_infos' =>
 [[$left_side_bearing, $right_side_bearing, $character_width, $ascent,
   $descent, $attributes],
  ...],
 'properties' => {$prop => $value, ...}
 )
```

```
$x->QueryTextExtents($font, $string)
=>
('draw_direction' => $draw_direction, ...)

$x->ListFonts($pattern, $max_names)
=>
@names

$x->ListFontsWithInfo($pattern, $max_names)
=>
({'name' => $name, ...}, {'name' => $name, ...}, ...)
```

The information in each hash is the same as the the information returned by QueryFont, but without per-character size information. This request is special in that it is the only request that can have more than one reply. This means you should probably only use**r equest()** with it, not **send()**, as the reply counting is complicated. Luckily, you never need this request anyway, as its function is completely duplicated by other requests.

```
$x->SetFontPath(@strings)

$x->GetFontPath
=>
@strings

$x->CreatePixmap($pixmap, $drawable, $depth, $width, $height)

$x->FreePixmap($pixmap)

$x->CreateGC($cid, $drawable, 'attribute' => $value, ...)

$x->ChangeGC($gc, 'attribute' => $value, ...)

$x->CopyGC($src, $dest, 'attribute', 'attribute', ...)

$x->SetDashes($gc, $dash_offset, (@dashes))

$x->SetClipRectangles($gc, ($clip_x_origin, $clip_y_origin),
                      $ordering, [$x, $y, $width, $height], ...)

$x->ClearArea($window, ($x, $y), $width, $height, $exposures)

$x->CopyArea($src_drawable, $dst_drawable, $gc, ($src_x, $src_y),
             $width, $height, ($dst_x, $dst_y))

$x->CopyPlane($src_drawable, $dst_drawable, $gc, ($src_x, $src_y),
              $width, $height, ($dst_x, $dst_y), $bit_plane)

$x->PolyPoint($drawable, $gc, $coordinate_mode,
              ($x, $y), ($x, $y), ...)

$x->PolyLine($drawable, $gc, $coordinate_mode,
             ($x, $y), ($x, $y), ...)

$x->PolySegment($drawable, $gc, ($x, $y) => ($x, $y),
                ($x, $y) => ($x, $y), ...)
```

```
$x->PolyRectangle($drawable, $gc,
                  [($x, $y), $width, $height], ...)

$x->PolyArc($drawable, $gc,
            [($x, $y), $width, $height, $angle1, $angle2], ...)

$x->FillPoly($drawable, $gc, $shape, $coordinate_mode,
             ($x, $y), ...)

$x->PolyFillRectangle($drawable, $gc,
                      [($x, $y), $width, $height], ...)

$x->PolyFillArc($drawable, $gc,
                [($x, $y), $width, $height, $angle1, $angle2], ...)

$x->PutImage($drawable, $gc, $depth, $width, $height,
             ($dst_x, $dst_y), $left_pad, $format, $data)
```

Currently, the module has no code to handle the various bitmap formats that the server might specify. Therefore, this request will not work portably without a lot of work.

```
$x->GetImage($drawable, ($x, $y), $width, $height, $plane_mask,
             $format)

$x->PolyText8($drawable, $gc, ($x, $y),
              ($font OR [$delta, $string]), ...)

$x->PolyText16($drawable, $gc, ($x, $y),
               ($font OR [$delta, $string]), ...)

$x->ImageText8($drawable, $gc, ($x, $y), $string)

$x->ImageText16($drawable, $gc, ($x, $y), $string)

$x->CreateColormap($mid, $visual, $window, $alloc)

$x->FreeColormap($cmap)

$x->CopyColormapAndFree($mid, $src_cmap)

$x->InstallColormap($cmap)

$x->UninstallColormap($cmap)

$x->ListInstalledColormaps($window)
=>
@cmaps

$x->AllocColor($cmap, ($red, $green, $blue))
=>
($pixel, ($red, $green, $blue))

$x->AllocNamedColor($cmap, $name)
=>
($pixel, ($exact_red, $exact_green, $exact_blue),
 ($visual_red, $visual_green, $visual_blue))

$x->AllocColorCells($cmap, $colors, $planes, $contiguous)
=>
([@pixels], [@masks])

$x->AllocColorPlanes($cmap, $colors, ($reds, $greens, $blues),
                     $contiguous)
=>
(($red_mask, $green_mask, $blue_mask), @pixels)

$x->FreeColors($cmap, $plane_mask, @pixels)

$x->StoreColors($cmap, [$pixel, $red, $green, $blue, $do_mask], ...)
```

The 1, 2, and 4 bits in `$do_mask` are do-red, do-green, and do-blue. `$do_mask` can be omitted, defaulting to 7, the usual case — change the whole color.

```
  $x->StoreNamedColor($cmap, $pixel, $name, $do_mask)
```

`$do_mask` has the same interpretation as above, but is mandatory.

```
  $x->QueryColors($cmap, @pixels)
  =>
  ([$red, $green, $blue], ...)

  $x->LookupColor($cmap, $name)
  =>
  (($exact_red, $exact_green, $exact_blue),
   ($visual_red, $visual_green, $visual_blue))

  $x->CreateCursor($cid, $source, $mask,
                   ($fore_red, $fore_green, $fore_blue),
                   ($back_red, $back_green, $back_blue),
                   ($x, $y))

  $x->CreateGlyphCursor($cid, $source_font, $mask_font,
                        $source_char, $mask_char,
                        ($fore_red, $fore_green, $fore_blue),
                        ($back_red, $back_green, $back_blue))

  $x->FreeCursor($cursor)

  $x->RecolorCursor($cursor, ($fore_red, $fore_green, $fore_blue),
                    ($back_red, $back_green, $back_blue))

  $x->QueryBestSize($class, $drawable, $width, $height)
  =>
  ($width, $height)

  $x->QueryExtension($name)
  =>
  ($major_opcode, $first_event, $first_error)
```

If the extension is not present, an empty list is returned.

```
$x->ListExtensions
=>
(@names)

$x->ChangeKeyboardMapping($first_keycode, $keysysms_per_keycode,
                          @keysyms)

$x->GetKeyboardMapping($first_keycode, $count)
=>
($keysysms_per_keycode, [$keysym, ...], [$keysym, ...], ...)

$x->ChangeKeyboardControl('attribute' => $value, ...)

$x->GetKeyboardControl
=>
('global_auto_repeat' => $global_auto_repeat, ...)

$x->Bell($percent)

$x->ChangePointerControl($do_acceleration, $do_threshold,
                         $acceleration_numerator,
                         $acceleration_denominator, $threshold)

$x->GetPointerControl
=>
($acceleration_numerator, $acceleration_denominator, $threshold)

$x->SetScreenSaver($timeout, $interval, $prefer_blanking,
                   $allow_exposures)

$x->GetScreenSaver
=>
($timeout, $interval, $prefer_blanking, $allow_exposures)

$x->ChangeHosts($mode, $host_family, $host_address)

$x->ListHosts
=>
($mode, [$family, $host], ...)

$x->SetAccessControl($mode)

$x->SetCloseDownMode($mode)

$x->KillClient($resource)

$x->RotateProperties($win, $delta, @props)

$x->ForceScreenSaver($mode)

$x->SetPointerMapping(@map)
=>
$status
```

```
$x->GetPointerMapping
=>
@map

$x->SetModifierMapping(@keycodes)
=>
$status

$x->GetModiferMapping
=>
@keycodes

$x->NoOperation($length)
```

`$length` specifies the length of the entire useless request, in four byte units, and is optional.

EVENTS

To receive events, first set the 'event_mask' attribute on a window to indicate what types of events you desire (see "pack_event_mask"). Then, set the protocol object's 'event_handler' to a subroutine reference that will handle the events. Alternatively, set 'event_handler' to 'queue', and retrieve events using **dequeue_event()** or **next_event()**. In both cases, events are returned as a hash. For instance, a typical MotionNotify event might look like this:

```
%event = ('name' => 'MotionNotify', 'sequence_number' => 12,
          'state' => 0, 'event' => 58720256, 'root' => 43,
          'child' => None, 'same_screen' => 1, 'time' => 966080746,
          'detail' => 'Normal', 'event_x' => 10, 'event_y' => 3,
          'code' => 6, 'root_x' => 319, 'root_y' => 235)
```

pack_event_mask

```
$mask = $x->pack_event_mask('ButtonPress', 'KeyPress', 'Exposure');
```

Make an event mask (suitable as the 'event_mask' of a window) from a list of strings specifying event types.

unpack_event_mask

```
@event_types = $x->unpack_event_mask($mask);
```

The inverse operation; convert an event mask obtained from the server into a list of names of event categories.

dequeue_event

```
%event = $x->dequeue_event;
```

If there is an event waiting in the queue, return it.

next_event

```
%event = $x->next_event;
```

Like Xlib's **XNextEvent()**, this function is equivalent to

```
$x->handle_input until %event = dequeue_event;
```

pack_event

```
$data = $x->pack_event(%event);
```

Given an event in hash form, pack it into a string. This is only useful as an argument to **SendEvent()**.

unpack_event

```
%event = $x->unpack_event($data);
```

The inverse operation; given the raw data for an event (32 bytes), unpack it into hash form. Normally, this is done automatically.

EXTENSIONS

Protocol extensions add new requests, event types, and error types to the protocol. Support for them is compartmentalized in modules in the X11::Protocol::Ext:: hierarchy. For an example, see X11::Protocol::Ext::SHAPE. You can tell if the module has loaded an extension by looking at

```
$x->{'ext'}{$extension_name}
```

If the extension has been initialized, this value will be an array reference, [$major_request_number, `$first_event_number`, `$first_error_number`, `$obj`], where `$obj` is an object containing information private to the extension.

init_extension

```
$x->init_extension($name);
```

Initialize an extension: query the server to find the extension's request number, then load the corresponding module. Returns 0 if the server does not support the named extension, or if no module to interface with it exists.

init_extensions

```
$x->init_extensions;
```

Initialize protocol extensions. This does a ListExtensions request, then calls **init_extension()** for each extension that the server supports.

WRITING EXTENSIONS

Internally, the X11::Protocol module is table driven. All an extension has to do is to add new add entries to the protocol object's tables. An extension module should `use X11::Protocol`, and should define an **new()** method

```
X11::Protocol::Ext::NAME
  ->new($x, $request_num, $event_num, $error_num)
```

where `$x` is the protocol object and `$request_num`, `$event_num` and `$error_num` are the values returned by **QueryExtension()**.

The **new()** method should add new types of constant like

```
$x->{'ext_const'}{'ConstantType'} = ['Constant', 'Constant', ...]
```

and set up the corresponding name to number translation hashes like

```
$x->{'ext_const_num'}{'ConstantType'} =
  {make_num_hash($x->{'ext_const'}{'ConstantType'})}
```

Event names go in

```
$x->{'ext_const'}{'Events'}[$event_number]
```

while specifications for event contents go in

```
$x->{'ext_event'}[$event_number]
```

each element of which is either `[\&unpack_sub, \&pack_sub]` or `[$pack_format, $field, $field, ...]`, where each `$field` is `'name'`, `['name', 'const_type']`, or `['name', ['special_name_for_zero', 'special_name_for_one']]`, where `'special_name_for_one'` is optional.

Finally,

```
$x->{'ext_request'}{$major_request_number}
```

should be an array of arrays, with each array either `[$name, \&packit]` or `[$name, \&packit, \&unpackit]`, and

```
$x->{'ext_request_num'}{$request_name}
```

should be initialized with `[$minor_num, $major_num]` for each request the extension defines. For examples of code that does all of this, look at X11::Protocol::Ext::SHAPE.

X11::Protocol exports several functions that might be useful in extensions (note that these are *not* methods).

padding

```
$p = padding $x;
```

Given an integer, compute the number need to round it up to a multiple of 4. For instance, `padding(5)` is 3.

pad

```
$p = pad $str;
```

Given a string, return the number of extra bytes needed to make a multiple of 4. Equivalent to `padding(length($str))`.

padded

```
$data = pack(padded($str), $str);
```

Return a format string, suitable for **pack()**, for a string padded to a multiple of 4 bytes. For instance, `pack(padded('Hello'), 'Hello')`gi ves `"Hello\0\0\0"`.

hexi

```
$str = hexi $n;
```

Format a number in hexidecimal, and add a ‘‘0x’’ to the front.

make_num_hash

```
%hash = make_num_hash(['A', 'B', 'C']);
```

Given a reference to a list of strings, return a hash mapping the strings onto numbers representing their position in the list, as used by `$x->{'ext_const_num'}`.

BUGS

This module is too big (˜2500 lines), too slow (10 sec to load on a slow machine), too inefficient (request args are copied several times), and takes up too much memory (3000K for basicwin).

If you have more than 65535 replies outstanding at once, sequence numbers can collide.

The protocol is too complex.

AUTHOR

Stephen McCamant <SMCCAM@cpan.org>.

SEE ALSO

perl(1), X(1), X11::Keysyms, X11::Protocol::Ext::SHAPE, X11::Protocol::Ext::BIG_REQUESTS, X11::Protocol::Ext::XC_MISC, X11::Protocol::Ext::DPMS, X11::Protocol::Ext::XFree86_Misc, X11::Auth, *X Window System Protocol (X Version 11), Inter-Client Communications Conventions Manual, X Logical Font Description Conventions*.