A graphics library to aid in the creation of games and other software
written by Dr.J
for the Commander X16 modern-retro computer

*

drjstudio.com/cx16.html

Table of Contents

1. Introduction

Xist16 is a library for the Commander X16 computer to aid the development of C programs for the CX16 by providing an abstraction layer and convenience functions for many common graphical operations that games need to perform, such as loading and managing graphical data, animating sprites and tiles, and displaying text. It also provides some minor conveniences for handling user input, and a pseudorandom number generator. Xist16 is made for C programs compiled using the cc65 compiler suite.

The goal of this library is to jump-start the development of games (and other graphics-intensive software) by providing plug-and-play functionality for these fundamental, common requirements so that developers don't have to reinvent the wheel and develop their own solutions.

1.1. Restrictions & Limitations

Xist16 is designed with certain assumptions in mind. The two most significant of these are:

- It is designed to run at 320 × 240 resolution. This was a very common resolution for gaming consoles of the era, and could potentially save a lot of memory and processing time since it doesn't take nearly as many graphics to fill up this screen resolution as 640 × 480.
- Since it is designed to support advanced graphical applications, it only supports 256-color depth.
- Xist16 dedicates layer 1 for the use of text display. This means you are restricted to using layer 0 for displaying background/terrain/map tiles, and will have to use sprites if you want to achieve parallax scrolling effects.
- Xist16 is primarily designed to make it easier to work with sprites and tiles; as such, it has no support for bitmap mode.
- Xist16 provides no sound support. For that, I recommend looking at the ZSound or ZSMKit library.

The library does consume a bit of code space. A program compiled with the X16 library that does nothing except run the initialization functions and enter an endless loop is 6 KB. This is not insignificant when you only have 40 KB of space to work with (plus the 8 KB high-RAM bank). However, since most games would need to implement most of this functionality anyway, thus consuming a comparable amount of space with their own implementations, this is not a serious limitation.

There are some pieces of functionality which would be nice to have which have not yet been implement. By far the most significant of these is tile streaming. I would like to add support for this in a future version of the library, but there is no ETA on that as of yet. At the moment, games that require tile streaming will have to implement that piece of functionality themselves.

1.2. Attribution

I am providing this library to the Commander X16 community in the hope that it is useful and aids developers in creating their own games and software on the CX16. All I ask in return is that somewhere in your software's credits you provide attribution that your software utilizes the Xist16 library by Dr.J. A link to drjstudio.com would also be appreciated. Something similar to the following:

```
Utilizes the Xist16 library by Dr.J (drjstudio.com)
```

You're free to modify the library for use in your projects, in which case just specify that you're using a customized version of the library in your attribution.

For legal purposes, I chose the MIT License which seems to be the open-source license that most closely suits my intent for the library, namely that users can use it however they want provided they provide attribution and don't hold the author (me) liable. You can find a copy of the license at the end of this document. Please be sure to include a copy of that license with software you build using Xist16.

1.3. Installation

If you are reading this document, you probably already have or know where to get the library, but if not, go to drjstudio.com/cx16.html and follow the links there to the downloads (as of this writing, this will take you to a Github repository).

Download the library to your local system. In the root folder you will find the C source files and a `makefile`. The header files are in the `inc` folder, and in the `lib` folder you will find a precompiled library file, `xist.lib`. You don't actually need to compile anything if you just want to start using the library in your projects without customizing any of the macros in the header files. However, if you want to customize any of these values or make modifications to the library, just run `make` from the root folder and it will produce a new `xist.lib` in the `lib` folder. You will need the cc65 compiler suite (cc65.github.io) on your system and its bin folder in your path environment variable to compile the library.

To use the library in your own projects, copy the header files in the `inc` folder to your own includes folder and `xist.lib` to your library folder, reference them in your compilation command, and you're good to go. See the makefiles in the example projects for how to do this if you're unsure.

1.4. Examples

The library comes with several fully-working examples to demonstrate how to use Xist16's various features. Each of these examples comes with a `makefile` to compile and run them. The example projects are:

- **`example1-sprites:`** Demonstrates defining, loading, and animating sprites with a ship sprite and 32 scrolling star sprites.
- **`example2-tiles:`** Demonstrates creating a tilemap with animated tiles and the ability to scroll around the map with the d-pad, stopping at the map edge.
- **`example3-text:`** Demonstrates the various text-drawing functions by printing text to the screen in a variety of ways.

2. Quickstart Guide

This section provides a high-level overview for how to get up and running with the Xist16 library quickly. Most of these topics are covered in more detail below.

Setup and Initialization

1. Customize the macro definitions in `xist_tiles.h` and `xist_mem.h` to control the dimensions for tile layer 0 and the memory addresses for your different graphical sections in VRAM, as appropriate for your application. Xist16's defaults are a 64 × 64-tile layer 0 with 16 × 16 tiles. You can also change the maximum number of sprites supported by Xist16 by editing the appropriate macro in `xist_gfx.h`. The default is 64. This can be changed up to 128 (the maximum number of sprites supported by the Commander X16), but the more sprites you animate simultaneously, the greater the strain on the processor and memory.
    - If you are OK with the defaults, you don't need to change anything, unless you need more space for your layer 0 tilebase.
2. Compile the library using the cc65 compiler suite. Copy `xist.lib` and the header files into your project folders and set up your makefile (or other tool you're using for compilation) to reference these files.
3. For your application's graphics, create image files in a format understood by Xist16 (raw sequences of palette indices). As described in the appendix, Xist16 comes with a utility that can help with this by converting PNG files into Xist16-compatible IMG files.
4. Create a `GFXMETA.DAT` file which contains the graphical metadata needed by your application, as explained in section 3.1.
5. In your program's setup and initialization, call the following functions to initialize Xist16:

```
xist_initialize_tiles();
xist_initialize_text_tiles();
xist_load_file_to_highram("gfxmeta.dat", XIST_SPRITE_METADATA_BANK);
```

6. Load graphical data from the SD card (or your computer's hard disk if using the CX16 emulator) using the `xist_load_graphic` function as explained in section 3.1.
7. Prepare sprites for display and animation using the `xist_prepare_sprite` function as explained in section 3.1.
8. Copy whatever tile graphics are necessary to render your current scene into VRAM using the `xist_copy_highram_to_vram_using_metadata` function as described in section 3.1.
9. If your game uses animated tiles, populate the `xist_map_tiles` tile animation lookup table and set the desired value for `xist_tile_animation_time_trigger` as explained in section 3.3.

Main Game Loop

- Enter an endless loop synced to 60Hz by calling the `xist_wait` function once per iteration.
- Animate whichever sprites are currently active using the `xist_process_sprites` function (called once per iteration of your main game loop, or less frequently if you need to reduce processor load, at the cost of reducing the maximum animation speed of your sprites), and update their positions onscreen with the `xist_update_sprite_position` function as explained in section 3.2.
- If you have scrolling background graphics, you can scroll them on-demand (and optionally constrain the view so it doesn't go beyond the edge of the tilemap) with the `xist_scroll_camera` function as explained in section 3.3.
- If you have animated tiles, animate them by calling `xist_animate_tiles` once per iteration of your main game loop (or less frequently if you need to reduce processor load).
- Display text using the text display functions explained in section 3.4.
- Handle user input using the functions explained in section 3.5.

3. Using the Library

Following are instructions in a tutorial-like format about how to use Xist16's functionality. For a more formal reference, please consult the auto-generated Doxygen documentation.

3.1. Defining and Managing Graphical Data

Having every graphic for a game in a separate file is messy and inefficient. It is preferable to store multiple images in a single file. Juggling graphics between different parts of memory can also be error-prone. Xist aims to solve these issues by providing a metadata format so that graphical data can be defined in an external file, read into high-RAM, and used to load graphics on-demand, either storing them in high-RAM or directly into VRAM.

GFXMETA.DAT

This is a convention rather than a strict requirement, but the metadata for handling images in Xist16 is stored in a file called `GFXMETA.DAT`, which should be loaded early in program initialization. Whether you adhere to this convention or call your file something else, you will need to understand how this file is formatted. It consists of rows of 14 bytes each, where each row defines one graphic to be used by your game, where a graphic can consist of a single images or multiple images comprising frames of animation. The format is as follows:

BYTES 0-3 **Filename**	**BYTE 4** **X Offset**	**BYTE 5** **Y Offset**
ASCII codes for the filename of the file containing your graphical data, not counting the file extension. Each filename should be exactly 4 characters long, followed by a `.IMG` extension. These files are required to consist of raw palette indices, with no 2-byte C64 header.	The number of pixels from the left side of the image file where the sub-image starts. This enables creating spritesheets by placing multiple images in one image file.	The number of pixels from the top of the image file where the sub-image starts. This enables creating spritesheets by placing multiple images in one image file.

BYTE 6 **Subimage Width**	**BYTE 7** **Subimage Height**	**BYTE 8** **Total Width**
The width, in pixels, of the subimage.	The height, in pixels, of the subimage.	The width of the entire image.

BYTE 9 **Number of Frames**	**BYTE 10** **Frame Timing**
The total number of frames of animation (1 for static images). If this number is greater than 1, Xist16 assumes that: 1. Every frame has the same dimensions. 2. Every subsequent frame is directly to the right of the previous one until the right edge of the image is reached, and then the next frame starts at the left side of the image, down a number of pixels equal to the subimage height specified in byte 7.	The number of time units to wait before going to the next frame of animation. The value of this time unit depends on how often you call the process_sprites function. If you call it every iteration of a 60-Hz game loop, then the value of this time unit is 1/60 second.

BYTE 11 Xist16 Sprite Bitmask	BYTE 12 CX16 Sprite Byte 6	BYTE 13 CX16 Sprite Byte 7
Bitmask used by the Xist16 for sprite processing. The only one relevant here is the second from the right, which determines whether the sprite's animation (if this graphic is for an animated sprite) repeats in a loop or only plays once. Set this value to 1 for a repeating animation and 0 for one that only plays once. All the other values should be zeroes. 00000000 = one-time 00000010 = repeating animation	Bitmask corresponding to byte 6 of the sprite metadata used by the Commander X16 in VRAM. The first 4 bits are for the collision mask, the next 2 are the z-depth, and the last 2 are for v-flip and h-flip. See the CX16 documentation for more details.	Bitmask corresponding to byte 7 of the sprite metadata used by the Commander X16 in VRAM. The first 2 bits are the sprite height, the next 2 bits are the sprite width, and the next 4 bits are the palette offset. See the CX16 documentation for more details.

You will need a hex editor to create and edit the Xist16 sprite metadata file. You should set the width of each row to 14 bytes to make it easier to edit. It is also recommended that you keep a record of which row in the metadata file corresponds to each of your graphics.

Loading Graphics into Memory

Once you have created your image files and defined the metadata Xist requires to understand them, you will typically load them into memory using the following flow:

1. Load image file to high-RAM or VRAM.

Call the `xist_load_graphic` function to load the image file from the SD card (or your computer's hard drive, if using the emulator) into either high-RAM or VRAM. This function has the following signature:

```
void xist_load_graphic(unsigned short metadata_index,
                       BOOL to_highram,
                       unsigned char to_highram_bank,
                       BOOL to_upper_vram,
                       unsigned long vram_address)
```

`metadata_index:` The row in your graphics metadata file that contains the information for this graphic.
`to_highram:` Set to `TRUE` (1) to load the graphics into high-RAM, or `FALSE` (0) to load the graphics into VRAM.
`to_highram_bank:` The bank of high-RAM to load the image data into, if loading into high-RAM (ignored if `to_highram` is set to `FALSE`).
`to_upper_vram:` If loading to VRAM, set this to `TRUE` to set the parameter into the Kernal's `LOAD` function to 3, loading into VRAM starting from `0x10000` + the specified VRAM address. Set to `FALSE` to set the parameter into the Kernal's `LOAD` function to 2, loading into VRAM starting from `0x00000` + the specified starting address. Ignored if `to_highram` is set to `TRUE`.
`vram_address:` If loading to VRAM, supply the VRAM address to load into with this parameter. Ignored if `to_highram` is set to `TRUE`.

The typical use case will be to load the file into high-RAM first, then when ready to display it onscreen, use the `xist_prepare_sprite function` (if it's a sprite) or `xist_copy_highram_to_vram_using_metadata` (if a tile).

2A. For sprites, initialize the sprite for display and (if applicable) animation onscreen.

Xist16 maintains its own array of metadata structs for sprites separate from that maintained by the Commander X16 in VRAM, since Xist16 needs a lot of additional information for how to animate the sprite. The struct's definition is:

```
struct XistAnimation {
  unsigned short first_frame_address[XIST_MAX_SPRITES];
  unsigned short address_offset[XIST_MAX_SPRITES];
  unsigned char num_frames[XIST_MAX_SPRITES];
  unsigned char current_frame[XIST_MAX_SPRITES];
  unsigned char frame_delay[XIST_MAX_SPRITES];
  unsigned char frame_timer[XIST_MAX_SPRITES];
  unsigned char bitmask[XIST_MAX_SPRITES];
  signed short x_pos[XIST_MAX_SPRITES];
  signed short y_pos[XIST_MAX_SPRITES];
};
```

This struct is referenced by the following variable:

```
extern struct XistAnimation xist_sprite_bank;
```

The `xist_sprite_bank` variable has a 1:1 correspondence with the CX16's sprite metadata in VRAM, such that index 0 in the `xist_sprite_bank` arrays refers to sprite 0; index 1 refers to sprite 1, etc.

Initializing a sprite in Xist16 involves not only copying all of its frames of animation into VRAM, but also populating its data in the `xist_sprite_bank`. Fortunately, the following function exists to take care of all this:

```
void xist_prepare_sprite(unsigned short metadata_index,
                         BOOL copy_to_vram,
                         unsigned long vram_address,
                         unsigned char highram_bank,
                         unsigned char sprite_index,
                         signed short x_pos,
                         signed short y_pos,
                         unsigned char add_bitmask)
```

`metadata_index:` The row in your graphics metadata file that contains the information for this sprite.
`copy_to_vram:` Set to `TRUE` if you want to copy the image data from high-RAM into VRAM. The use case for setting this to `FALSE` is if you have many sprites that use the same image data, in which case you could set this to `TRUE` when preparing the first sprite and `FALSE` for all subsequent calls to save a lot of unnecessary copying.
`vram_address:` If loading to VRAM, supply the VRAM address to load into with this parameter. Ignored if `to_ vram` is set to `FALSE`.
`highram_bank`: The high-RAM bank that contains the image data to be copied.
`sprite_index:` The index of the sprite in both the `xist_sprite_bank` and the Commander X16's sprite metadata.
`x_pos:` The onscreen X position (in pixels) to display the sprite at.
`y_pos:` The onscreen Y position (in pixels) to display the sprite at.
`add_bitmask:` A bitmask with any flags you want to turn on for this sprite. By far the most common use case would be to provide `XIST_SPRITE_ACTIVE` to make the sprite immediately active for use.

2B. For tiles, copy the image data into VRAM.

Tiles are handled differently from sprites (as described in section 3.3), so for tiles, you only need to copy the image data into VRAM using the following function:

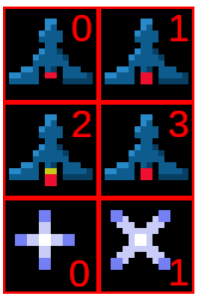
```
void xist_copy_highram_to_vram_using_metadata(unsigned long vram_address,
                                              unsigned char highram_bank,
                                              unsigned short metadata_index)
```

`vram_address:` The address in VRAM to copy your image data to.
`highram_bank:` The high-RAM bank to copy your image data from.
`metadata_index:` The row in your graphics metadata file that contains the information for this graphic.

Recommendations for Image Files

In order to make the best use of space, make your graphical data easier to work with, cut down on unnecessary data copies, and so on, I recommend following these guidelines when creating your images:

- Make each image as close to 8 KB as possible without going over 8 KB. 8 KB is the size of each bank in high-RAM, so the closer each image comes to filling a high-RAM bank without spilling over, the more efficient your overall space utilization will be.
- 8 KB is larger than the image sizes natively supported by the Commander X16, so this means your image files will each contain multiple images. That's not only fine, it's how you *should* create your graphical data and how all games do it, because it reduces the number of file read operations (opening and reading from files is a slow operation with a lot of overhead).
- To the maximum extent you can, try to have each image file contain sub-images with the same dimensions so that they all fit in a uniform grid. This makes them easier to reference and work with, and it's how the Xist16 library expects your animations to be laid out: in a uniform grid where, from the first frame of animation, each subequent frame is to the right of the previous until you hit the edge of the image, then the next one is one row down start from the left side of the image. Here's an example:



32x48 image file with a ship and star sprite

Each frame of animation is 16x16 pixels
Ship sprite animation starts from position (0,0) with 4 frames of animation
Star sprite animation starts from position (0,32) with 2 frames of animation

3.2. Animating Sprites

Once you have loaded sprites into VRAM and populated the `xist_sprite_bank` using the `xist_prepare_sprite` function, it is quite easy to animate your sprites. It only takes a single function call:

`void xist_process_sprites()`

This function is a performance bottleneck, so it has had performance optimizations applied. One of these means it does not take parameters directly (because the 6502 CPU has no hardware support for a call stack, so the cc65 compiler has to emulate one in software, which is very slow). However, it takes what I will call "virtual parameters" in the form of global variables which must be set before calling the function.

`xist_proc_sprite_start_idx:` The index of the first sprite (inclusive) to be animated. This is the sprite's index in the `xist_sprite_bank` and the Commander X16's sprite metadata in VRAM.
`xist_proc_sprite_end_idx:` The index of the last sprite (inclusive) to be animated.

When called, the function iterates through all of the sprites within the range provided and performs all the updates necessary to cycle through each sprite's animation frames. Note that it ignores any sprites whose `XIST_SPRITE_ACTIVE` bit is set to zero, so you can selectively deactivate sprites without having to remove or overwrite them by setting this bit to zero using the `bitmask` field for that sprite in the `xist_sprite_bank`.

The other function of interest is the function for updating a sprite's onscreen X and Y coordinates to make sure they stay synced between the sprite's metadata in VRAM and the `xist_sprite_bank`:

`void xist_update_sprite_position()`

Like `xist_process_sprites`, it is important that this function be performant, so it uses also "virtual parameters" in the form of globals that must be set before being called:

`xist_curr_sprite_idx:` The index of the sprite whose position needs updated. This is the sprite's index in the `xist_sprite_bank` and the Commander X16's sprite metadata in VRAM.
`xist_curr_sprite_x_pos:` The sprite's new onscreen X position (in pixels).
`xist_curr_sprite_y_pos:` The sprite's new onscreen Y position (in pixels).

Xist16 doesn't have built-in convenience functions for updating other sprite values such as V- and H-flip or palette offset, but if you require these, you can easily add them using the implementation for `xist_update_sprite_position` as an example.

3.3. Using Tilemaps

In this context, the term "tilemap" is used to mean a game map/terrain/background graphics rendered using the Commander X16's built-in support for tile layers. Xist16 provides support to make working with tilemaps easier.

Note that Xist16 reserves tile layer 1 for text display, so by default, you can only use layer 0 for displaying map/terrain/background graphics. You are welcome to modify the library to work differently from this assumption, but if so, you are responsible for any changes that need to be made to the library code to accommodate your customizations. This documentation assumes you are using Xist16 unmodified.

Xist provides built-in support to make working with tilemaps easier in the following ways:

Automatic Configuration by Editing Header Files

Setting up the Commander X16's tile layers involves setting several different memory addresses and VERA values and making sure they line up with the math and logic in your code. Even if you have a good understanding of all these elements, it's easy to make a small mistake. To reduce the possibility of human error, Xist16 is set up so you just have to edit a few values in a couple of header files, and then call the tile initialization function to have it automatically configure all the other values for you.

In `xist_tiles.h`, edit the following values to reflect the requirements of your program:

`LAYER_0_MAPBASE_WIDTH:` The number of tiles wide your layer-0 mapbase is.
`LAYER_0_MAPBASE_HEIGHT:` The number of tiles high your layer-0 mapbase is.
`LAYER_0_MAPBASE_TOTAL:` This value must be the product of the two values above. The only reason it isn't auto-calculated is because it's used to initialize the array size of the xistmaptiles struct.
`LAYER_0_TILE_DIMENSIONS:` The width and height of your layer 0 tiles, in pixels. The only valid values are 8 or 16. Xist16 doesn't have built-in support for tiles that have different width and height dimensions, but you could add it yourself by editing the library code if your program requires it.

`xist_tilemap_width_pix:` This is not a universal define, but a value that you can edit on a per-map basis. It defines the viewable area of the current tilemap. If your current tilemap is smaller than the maximum size, set this value to constrain the amount of the tilemap that can be viewed with the `xist_scroll_camera` function. This is the total width of the current tilemap in *pixels*, so it should equal to the total desired width in tiles multiplied by the dimensions of your tiles.
`xist_tilemap_width_pix:` Same as the above, but for height rather than width.

In `xist_mem.h`, define the addresses of your mapbases, tilebases, and sprite graphical data with the following macros:

`XIST_LAYER_1_TILE_BASE:` The address of the layer-1 tilebase. If you are using Xist16's text drawing functionality, do not change this value.
`XIST_LAYER_1_MAPBASE:` The address of the layer-1 mapbase. If you are using Xist16's text drawing functionality, do not change this value.
`XIST_LAYER_0_MAPBASE:` The address of the layer-1 mapbase. If you are using Xist16's text drawing functionality, you don't need to change this value either, because it will start right at the point where the space allocated to Xist16's text layer ends.
`XIST_LAYER_0_TILE_BASE:` The address of the layer-1 tilebase. Assuming you are keeping the order of these addresses (layer-1 tilebase first, then layer-1 mapbase, then layer-0 mapbase, then layer-0 tilebase), the starting address for this section should be equal to the address of your layer-0 mapbase, plus the total tiles in your mapbase multiplied by 2 (because each tile has 2 bytes).
`XIST_VRAM_SPRITE_IMAGE_DATA:` The starting address of the graphical data for your sprites. This should be equal to the address of your layer-0 tilebase plus however much space is necessary to hold the maximum amount of layer-0 tile graphics you'll hold in VRAM at a time.

IMPORTANT! If you change the macros in these header files from their defaults, you will need to recompile the library and use the new library file in your project!

Once you've defined your tile macros and addresses in VRAM, simply call the `xist_initialize_tiles` function early in your program's run and it will automatically take care of setting up the other necessary values and VERA configuration.

Scrolling the “Camera”

By default, when you scroll a tile layer, the Commander X16 will repeat that layer when you hit the edge, so that it can scroll endlessly. This makes sense if a layer is representing purely background graphics that the player cannot interact with, and a minority of interactive game maps may also want this behavior. However, in games built with the Xist16, tile layer 0 will probably represent interactable terrain in the majority of cases, and the majority of interactable maps do not want to scroll endlessly because they represent a finite area.

Therefore, having a function that automatically constrains the “camera” (where camera here is defined as the area of the tilemap that is currently visible onscreen) to the edges of the map would be useful. For this purpose, the following function exists, which will scroll the camera (i.e., set the H- and V-scroll values of layer 0) by the amount indicated:

```
void xist_scroll_camera()
```

This function uses “virtual parameters” in the form of globals that must be set before being called:

`xist_camera_change_x:` The number of pixels by which to scroll the camera horizontally (a negative value moves the camera left; a positive value moves it to the right).

`xist_camera_change_y:` The number of pixels by which to scroll the camera vertically (a negative value moves the camera up; a positive value moves it down).

`xist_tilemap_width_pix:` This value doesn’t need to be changed every time this function is called, it just needs to be set each time you initialize a new tilemap. It should be equal to the viewable area of the current tilemap, in pixels. Its minimum value should be the horizontal screen resolution and its maximum value should be equal to the width of your layer-0 mapbase (in tiles) multiplied by the dimension of your tiles.

`xist_tilemap_height_pix:` Same as the above parameter, but for height rather than width.

`xist_allow_tile_scroll_wrap:` Set to `FALSE` to constrain the camera to the viewable area defined by `xist_tilemap_width_pix` and `xist_tilemap_height_pix`. Set to `TRUE` if you want to allow the default Commander X16 behavior (endless, repeating scrolling).

Note that Xist16 doesn’t take care of updating sprite positions when you scroll the tilemap, so you will need to make sure to update the positions of any sprites that should scroll together with the tiles.

Animated Tiles

Finally, Xist16 provides support for animated map tiles, so that you can have things like animated water, foliage blowing in the wind, glowing lights, etc., in your map graphics. Because of performance constraints, the way this is implemented is different and simpler from sprites:

There is a “lookup table” for tile animations which is an array where every index is the index of that tile in VRAM, and the corresponding element is the index of the *next tile* that comes after that one in the animation sequence.

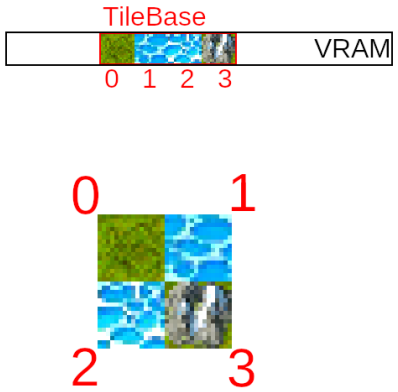
```
struct XistMapTile {
  unsigned char tile_index[LAYER_0_MAPBASE_TOTAL];
};
extern struct XistMapTile xist_map_tiles;
```

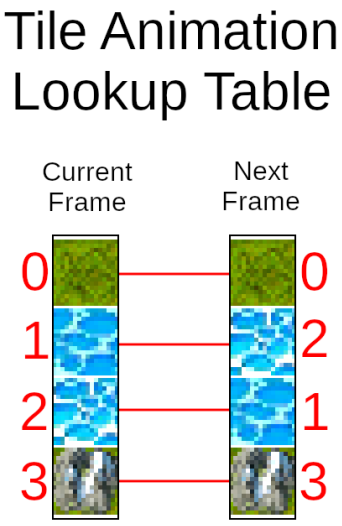
Simply fill in this lookup table with the correct values for your animation sequences, then set the variable `xist_tile_animation_time_trigger` equal to the number of game ticks you want your tiles to animate at (because of resource constraints, all tiles animate on the same timer).

Then, call `xist_animate_map_tiles()` each iteration of your main game loop and it will automatically animate your tiles using the timer value you defined and the animation sequences specified in the lookup table. This function takes no parameters.

> **IMPORTANT!** The `xist_animate_map_tiles` function assumes that wrapping tile layer 0 is disallowed (e.g., by only scrolling tile layer 0 via the `xist_scroll_camera` function with `xist_allow_tile_scroll_wrap` set to `FALSE`). If you call this function with tile layer 0 allowed to wrap, its behavior is **undefined.**

Here's an example of setting up your tile animations. This example has 4 tile graphics: grass (index 0), water with a 2-frame animation (indices 1 and 2), and mountains (index 3). In `xist_map_tiles`, indices 0 and 3 map to themselves (because those tiles aren't animated), while indices 1 and 2 map to each other to create a repeating 2-frame animation:





3.4. Displaying Text

In any game with an advanced UI, `printf` simply isn't going to cut it, so Xist16 provides support for text display.

Important Note 1: The Xist16 text functions all assume standard ASCII encoding. When targeting the Commander X16, the cc65 compiler does not encode all characters in hard-coded strings using standard ASCII characters. This means that if you have any hard-coded strings for your game, you must not define them as hard-coded strings, but as arrays of raw ASCII values. It is of course recommended that if you make a very text-heavy game you keep most of your game's text in external data files to avoid bloating the size of your executable and consuming too much of that previous 40 KB of low-RAM.

Important Note 2: The font used by the Xist16 consists of 8 × 8 monospace characters, and when drawing text, it does so by setting tiles on tile layer 1 to point to the location of that glyph's graphical data in memory. This means that all X and Y coordinates when specifying where to draw strings are given in *tile* coordinates, not pixel coordinates. At 320 × 240 resolution with 8 × 8 tiles, the screen is 40 tiles wide by 30 tiles high. Keep this in mind when setting the coordinates for where to draw your strings.

Xist16 provides the following text functionality:

Initialization and Resetting State

Before using any of Xist16's text functionality, you must call the `xist_initialize_text_tiles` function during your program's setup. This function takes no arguments.

To clear all text off the screen, call the `xist_clear_text` function. This function also takes no arguments.

Before drawing typewriter text (explained below), you should always call `xist_reset_typewriter_counters`. This function also takes no arguments.

Drawing a String Onscreen

Call the following function draw a single line of text:

```
void xist_draw_text()
```

This function uses "virtual parameters" in the form of globals that must be set before being called:

`xist_text:` The "string" (array of raw ASCII codes) to display onscreen. It must be null-terminated (i.e., the last value in the string must be 0) or the behavior of this function is **undefined.**
`xist_text_tile_x:` The The X position (in *tile* coordinates) to draw the string. The string will be *left-aligned* on this position.
`xist_text_tile_y:` The The Y position (in *tile* coordinates) to draw the string.
`xist_palette_offset:` Palette offset value, which can be used to draw text in different colors. Leave this at its default of 0 to leave the text its default color (white). The `xist_text.h` header file has predefined macros for all 16 colors that can be achieved with this parameter.

Remember that once you call this function, those tiles will be set to the glyphs of the string you provided until you clear those tiles, either by calling `xist_clear_text` or by overwriting the string with another one.

Typewriter Text

"Typewriter text" is the name for the effect where text is rendered one character at a time instead of all at once, to give the impression of spoken dialogue or narration. Xist16 provides support for typewriter text. Typewriter text is defined using a two-dimensional array:

```
xist_typewriter_text[XIST_TYPEWRITER_TEXT_ROWS][XIST_TYPEWRITER_TEXT_COLUMNS]
```

The default defines provide space for up to 36 columns and 10 rows of typewriter text that can be displayed onscreen at once. You can alter these values in the header file if you want space for more (or less) text, but as always, you'll need to recompile the library for these changes to take effect.

To display typewriter text, you must fill the `xist_typewriter_text` array with the strings you wish to display, then call `xist_reset_typewriter_counters`, then set the `xist_tw_text_tile_x` and `xist_tw_text_tile_y` "virtual parameters" (global variables) to the *upper-left* tile of where you want the typewriter text to begin displaying. As always, all strings must be null-terminated to avoid undefined behavior.

Finally, call `xist_draw_typewriter_text` for each game tick that you want to display the next character in the typewriter text strings. You can control the speed at which the text displays by controlling how frequently you call this function. Assuming your game loop runs at Hz, you could call it once per iteration to display 60 characters per second, or every other iteration to display 30 characters per second, etc.

As with `xist_draw_text`, you can set `xist_palette_offset` to draw your typewriter text with various colors.

Replacing Placeholder Substrings

Many games would benefit from being able to replace placeholder substrings in their text with other strings. For example, you may want the player to be enter a name for his character, and then use that character's name in dialogue. Since you can't know what the player will choose as his name ahead of time, you'll need to save it in a variable and then be able to insert it into your game's text, which is done by using a placeholder and replacing occurrences of that placeholder within your string.

To avoid having to bring in the code bloat of including a standard library for string replacement (and standard library functions might not work anyway since the Xist16 text functions use arrays of ASCII values instead of hard-coded strings), Xist16 defines its own function for substring replacement. To keep the implementation simple (and thus small and fast), it has 2 important limitations to note:

1. It will only replace the *first* occurrence of the placeholder text within a string, not all occurrences.
2. It assumes that the placeholder is equal to or longer than the text that will replace it.

To replace a substring within a string, use the following function (which uses normal rather than virtual parameters since it wouldn't be called often enough to be a performance bottleneck). As always, all strings must be null-terminated to avoid undefined behavior.

```
void xist_replace_substring(char *str, const char *placeholder, const char *replacement)
```

`str:` The string which contains the substring to be replaced.
`placeholder:` The substring to be replaced (only the first occurrence will be replaced).
`replacement:` The string to replace the placeholder with. Its length must be equal to or less than the length of the placeholder.

For example, if you have the following variables and call to `xist_replace_substring` (in actuality, you would define your text as arrays of ASCII codes, but they are shown here as hard-coded strings for clarity):

```
char *str = "Be wary, <<name>>!";
char *placeholder = "<<name>>";
char *replacement = "Umbold";
```

```
xist_replace_substring(str, placeholder, replacement);

// str now contains the text “Be wary, Umbold!”
```

Converting Numerical Data to Text

There are many kinds of numerical data that games need to display as text, such as score, timers, hit points, damage values, and so on. Xist16 provides the following function for converting numerical data into arrays of ASCII codes usable with its text display functions:

```
void xist_convert_long_to_ascii_array(signed long num, char* result)
```

`num:` The number to be converted into a string.
`result:` The string to hold the textual representation of the number.
Returns the length of the resulting string.

3.5. Handling User Input

The Commander X16 has pretty good built-in support for getting user input, but Xist16 provides a few conveniences to make processing user input a bit easier.

Detecting Pressed Joypad Buttons

Xist16 provides a function to return the current state of all joypad buttons as a single 16-bit variable, plus predefined macros in `xist_input.h` which can be ANDed with this variable to determine which buttons are pressed.

```
unsigned short xist_get_joypad()
```

This function takes no arguments. It returns the current state of all joypad buttons as a 16-bit integer. This state has been inverted from that returned by the Commander X16 so that 1 indicates a button is pressed and 0 indicates it is released (which I find more intuitive and easier to work with).

Example usage:

```
unsigned short joypad_state = xist_get_joypad();
if (joypad_state & JOYPAD_B) {
  // respond to the B button being pressed
```

Falling Edges

In many video games, you as the developer don’t only care about when a button is currently being pressed. Sometimes, you care about when a button *has just been released after being pressed.* This state is known as a “falling edge,” and there are many cases when this is the state you want to respond to, not every single frame that a button is pressed. The most obvious example would be when navigating menus.

To help with getting buttons that are in a falling edge state, Xist16 provides the following function:

```
unsigned short xist_joypad_get_falling_edges(unsigned short last_state)
```

`last_state:` The state of the joypad buttons (as returned by `xist_get_joypad`) since the last time this function was called.

Returns a 16-bit integer where all buttons that are currently in a falling edge state have a value of 1 and those that are not have a value of 0.

In order to use this function, you need to keep track of what the previous button state was. Here is now this might look in your main game loop:

```
for(EVER) {
  xist_wait();

  current_joypad_state = xist_get_joypad();
  current_joypad_falling_edges =
    xist_joypad_get_falling_edges(previous_joypad_state);

  // process input

  previous_joypad_state = current_joypad_state;
}
```

Mouse Input

Xist16 has the same functions for mouse input as for joypad input, and they work the same way, and also have macros in `xist_input.h` to easily check which mouse button is pressed or in a falling edge state:

```
unsigned char xist_get_mouse_buttons()
unsigned char xist_get_mouse_buttons_get_falling_edges(unsigned char last_state)
```

Xist16 also provides the following function to retrieve the position of the mouse cursor:

```
void xist_update_mouse_position()
```

This function takes no arguments and has no return, but it updates the global `mouse_x` and `mouse_y` pointers with the current pixel coordinates of the mouse cursor onscreen.

Note that, before you can use the mouse input functions, you must initialize the mouse with:

```
asm("jsr $FF5F");
asm("lda #1");
asm("jsr $FF68");
```

Warning: I have found that if you initialize the mouse too early in your program's run, it can cause strange behavior up to and including system crashes. I have found that if I wait until the end of my program's setup code to initialize the mouse, the behavior is generally more stable. At present, I don't yet have an understanding for the reason behind this behavior

3.6. Pseudorandom Number Generation

Xist16 includes a pseudorandom number generator designed for speed and for generating random 8-bit numbers that give varied and unpredictable values even when generating very small numbers (a trait which many PRNG algorithms struggle with) on an acceptably long period (in this case, 65,536 generations).

Before calling the PRNG, initialize `xist_seed_a` to a non-zero random value, such as by getting the current frame counter the first time the user presses a button.

After initializing one or both random seed values, you can obtain random numbers with a minimum lower range of 0 and a maximum upper range of 255 by calling the following function. If you need larger random numbers, you can call the PRNG twice and concatenate the two returned bytes into a 16-bit number with bit shifting.

```
unsigned char xist_rand()
```

This function uses “virtual parameters” in the form of globals that must be set before being called:

`xist_rand_min:` The minimum value (inclusive) that you wish returned.
`xist_rand_max:` The maximum value (inclusive) that you wish returned.

Appendix

A.1 Utilities

I have provided a couple of useful utilities to aid in the creation of content compatible with the Xist16 library. They are available at drjstudio.com/cx16.html.

Xist Image Converter

The image converter converts PNG files into Xist16-compatible IMG files. It is created using Processing (processing.org). It is available in 2 versions, Windows and Linux. The Windows version is standalone and can be downloaded and used without any prerequisites. The Linux version requires Java on your system to use it. The source code comes with both downloads, so you can download the Processing framework (it's free) and modify or reimplement the utility if you wish. To use the image converter:

- In the `data` folder you will find a file called `cx16palette.png`. This is an image with exactly 256 pixels where each pixel corresponds to a color from the default Commander X16 palette, in the same order as on the Commander X16. If your application is using a non-standard palette, you can replace this file with a file that is exactly 256 pixels, where each pixel corresponds to a color from your palette, in the same order that your palette data is stored in your CX16 application.
- Place any PNG files that you wish to convert into the `data` folder.
- Run the program.
- Every PNG file in the `data` folder (except `cx16palette.png`) will have an IMG file generated for it. Copy these IMG files into the same folder as your CX16 application and Xist16 will be able to load and parse them using its data loading and copying functions.

String to ASCII Converter

This is a simple Python script that accepts a command-line argument of a string to be converted into a comma-separated series of ASCII codes. This is useful for quickly generating hard-coded ASCII arrays for use with the Xist16 drawing functions without having to generate them by hand. Don't forget to put a null terminator (0) at the end of all your strings. Obviously, you will need Python installed on your system to run this script.

A.2 Useful References

Following are links to a variety of tools and references that I have found very useful while learning Commander X16 development. Of course, I cannot guarantee that all of these links will remain valid indefinitely into the future.

Xist16 Homepage	drjstudio.com/cx16.html
Commander X16 Official Website	commanderx16.com
cc65 Compiler Suite	cc65.github.io
Optimizing cc65 Code	github.com/ilmenit/CC65-Advanced-Optimizations
GNU Make for Windows	gnuwin32.sourceforge.net/packages/make.htm
ZSound Library	github.com/ZeroByteOrg/zsound
ZSMKit	github.com/mooinglemur/zsmkit
Commander X16 C Programming Tutorial	github.com/mwiedmann/cx16CodingInC/

A.3 License