

DuinoCube User Manual

ver. 0.1

by Simon Que

April 27, 2014

Table of Contents

[Table of Contents](#)

[Introduction](#)

[Why Arduino?](#)

[System Overview](#)

[Introduction to Arduino](#)

[DuinoCube shields](#)

[GFX Shield architecture](#)

[UI Shield architecture](#)

[Software architecture](#)

[Setup Guide](#)

[Getting the Arduino IDE](#)

[Installing the DuinoCube library](#)

[Hardware setup](#)

[Loading a sketch](#)

[Programming Guide](#)

[Arduino programming model](#)

[Setting up the game](#)

[Loading the graphics data](#)

[Defining strings in Arduino code](#)

[Drawing the tile layers and sprites](#)

[The main game loop](#)

[API Reference](#)

[Top-level module](#)

[printf\(\)](#)

[Core module](#)

[System control functions](#)

[Data loading functions](#)

[Tile layer functions](#)

[Sprite functions](#)

[File module](#)

[Gamepad module](#)

[Mem module](#)

Introduction

This manual is a starter guide for developing games on DuinoCube. It provides an overview of the system architecture. It also introduces the Arduino boards and editor. However, it does not delve into the finer details of some system components, such as the FAT file system and the USB or SPI protocols.

You should have a working knowledge of:

- How computer systems work.
- Downloading, installing, and copying things on your computer.
- Programming in C/C++.
- Electronic circuits (helpful but not required).

Why Arduino?

There are a number of reasons for building a game console on the Arduino platform.

First, Arduino is well-established. There a standardized selection of Arduino boards to suit your needs. You can also find hundreds of different parts and accessories to customize your system.

Arduino also makes microcontroller development simple. There is no need to deal with programming cables, different architectures, pin and port numbers, etc. Everything is packaged neatly behind an integrated environment. You need only a USB cable to program an Arduino board.

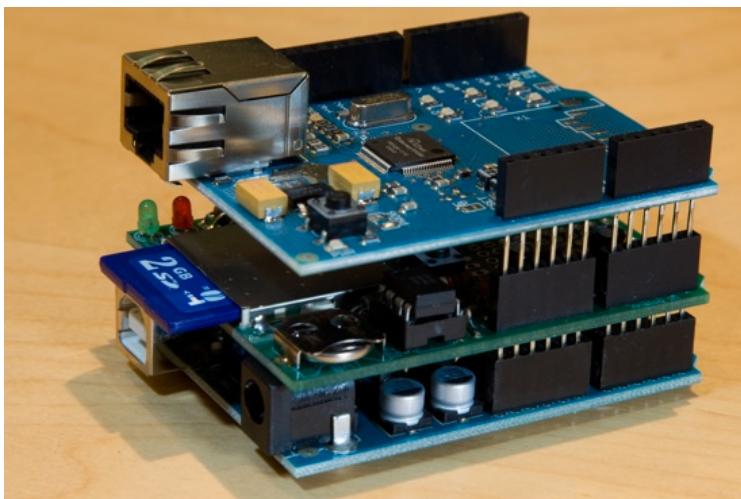
Finally, it is very easy to find help for Arduino. Just go online and browse the official Arduino forum at <http://forum.arduino.cc>.

System Overview

DuinoCube is based on the popular **Arduino** platform. DuinoCube adds various capabilities that make up the components of a game console: graphics, audio, data loading, and user input.

Introduction to Arduino

Arduino systems can be built using a base Arduino board plus one or more shields. A **shield** is an expansion board that attaches to the top of an Arduino. Each shield has a specific function. There are shields that provide SD card slots, motor control, ethernet modem, and more. Most importantly, Arduino shields can be designed to stack on top of one another, as shown in the photo here:



The stacking structure allows all Arduino pins to reach all pins of each shield. As long as the shields' control pins don't overlap, the Arduino board can access all of the connected shields.

Arduino boards come in various shapes and sizes. DuinoCube is designed to be compatible with four of them:

Board name	Microcontroller	Program size	RAM size	Notes
Arduino Uno	ATmega328P (AVR)	32 kB	2 kB	Most popular
Arduino Esplora	ATmega32U4 (AVR)	32 kB	2.5 kB	Built-in gamepad
Arduino Mega	ATmega2560 (AVR)	256 kB	8 kB	Best value
Arduino Due	AT91SAM3X8E (ARM)	512 kB	96 kB	Most powerful

References:

<http://arduino.cc/en/Main/Products>

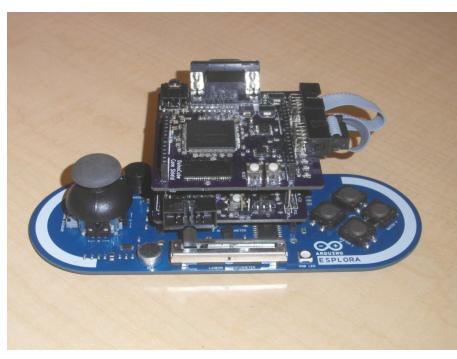
DuinoCube shields

To build a DuinoCube on an Arduino board, you must attach two types of shields on top:

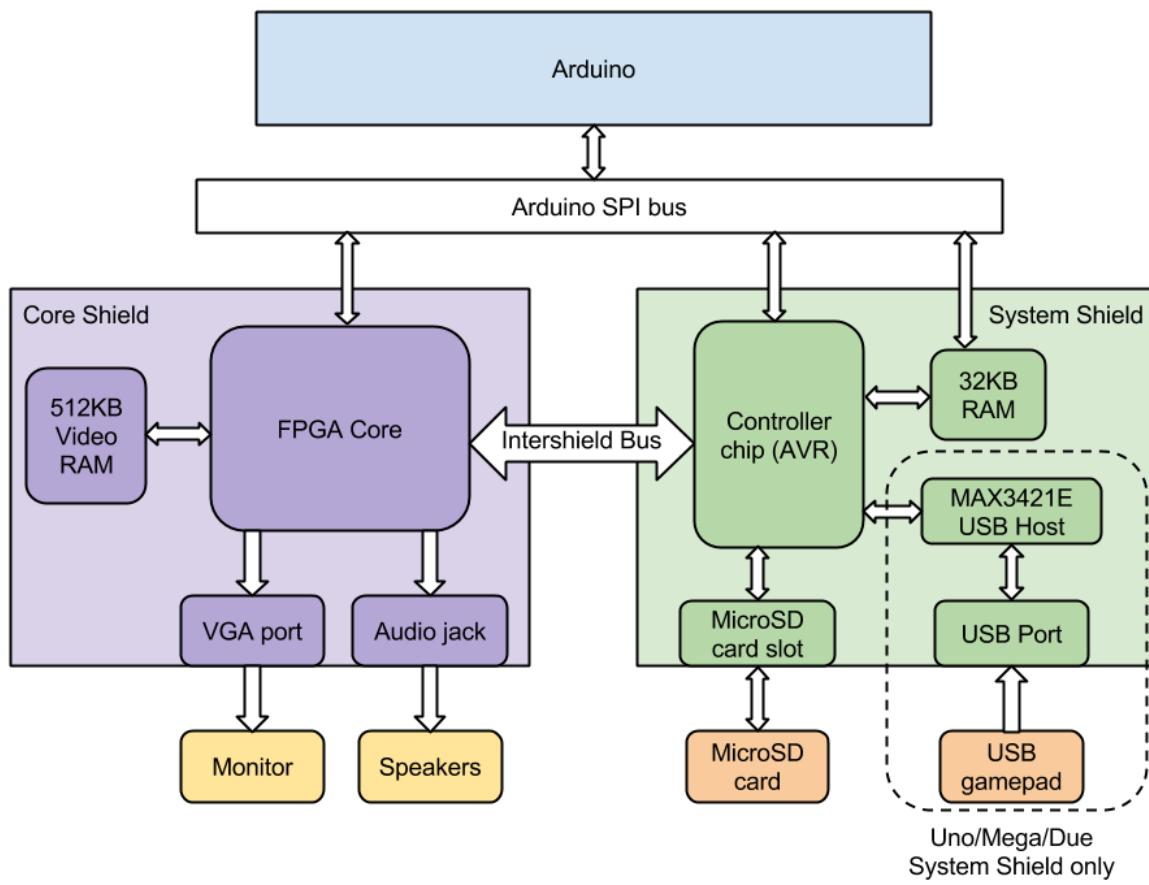
- The DuinoCube **GFX Shield**, which provides video and audio.
- One of two DuinoCube **UI Shields**, depending on your Arduino board. They contain these features.
 - **Basic UI Shield** for Uno/Mega/Due: has SD card, USB host, and 32 kB extra RAM.
 - **Esplora UI Shield**: has SD card and 32 kB of extra RAM.

This table summarizes the shields required to build DuinoCube for each Arduino board type:

Board name	Shield #1	Shield #2	Photo of complete system
Arduino Uno	GFX Shield	Basic UI Shield	
Arduino Mega	GFX Shield	Basic UI Shield	

Arduino Due	GFX Shield	Basic UI Shield	
Arduino Esplora	GFX Shield	Esplora UI Shield	

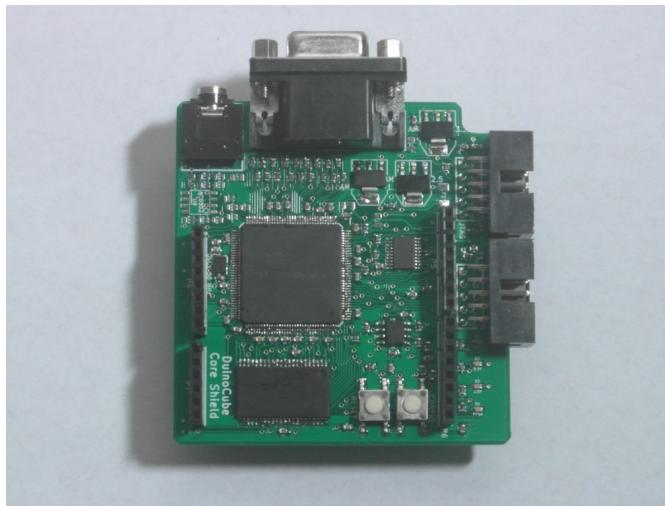
The Arduino controls the two shields using the **Serial Peripheral Interface (SPI)** bus. There is also an interface between the two shields that allows data to be directly copied between them. The following diagram shows the components of the two shields and how they fit into the system:



References:

http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

GFX Shield architecture



The GFX Shield contains a **field programmable gate array** (FPGA). An FPGA is a programmable logic chip. But instead of a software program, it lets you program a circuit onto it. The GFX Shield's FPGA contains a custom video and audio system called the **Core**. As the name suggests, the Core is central to the operation of DuinoCube.

Here is a list of some of the Core's features:

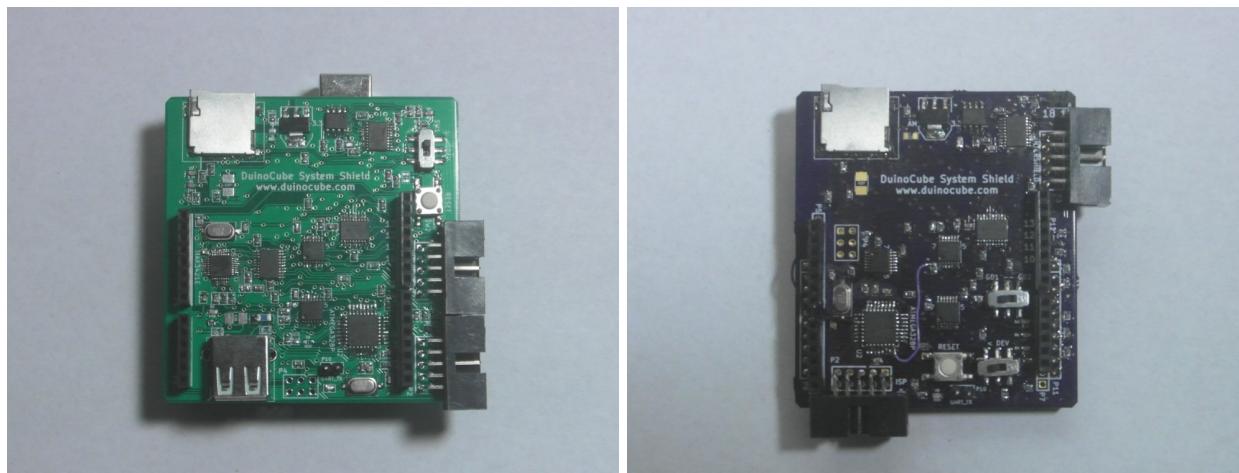
- **VGA graphics output.** Lets you display the graphics output on a computer monitor and some newer TV's. The image resolution is 320x240. More video modes TBD.
- **Four color palettes.** Each palette contains 256 colors. The output can thus contain up to 1024 colors. The full range of possible colors is 252,144 (18-bit).
- **Four tile layers.** Each tile layer is composed of a 32x32 grid of 16x16 tiles. Alternatively, 8x8 tiles can be used. The tile layers can be moved independently of one another.
- **256 sprites.** A sprite is a 2D graphics object that stands on its own. Each sprite can be moved independently of the others. There is also collision detection for the sprites.
- **Stereo audio output.** The left and right channels can be independently programmed to generate tones with different frequencies. The audio system is not fully developed yet so a lot of details are TBD.

Additionally, there is 512 KB video memory (VRAM) that stores image data.

References:

http://en.wikipedia.org/wiki/Field-programmable_gate_array
[http://en.wikipedia.org/wiki\(Sprite_%28computer_graphics%29](http://en.wikipedia.org/wiki(Sprite_%28computer_graphics%29)

UI Shield architecture



The UI Shield comes in two flavors, the **Basic version** (left) and the **Esplora version** (right). Here are the features that are common to the two versions:

- **Controller chip.** Another AVR microcontroller (ATmega328P) that handles the various operations in the system. It acts as a middleman between the main Arduino board and the other components of the system.
- **32 kB of RAM.** This is called shared memory as it can be accessed by both the controller chip and the Arduino. It is used to pass data between the two processors. It can also be used as extra memory on the smaller Arduinos.
- **MicroSD card slot.** Load data and games from a microSD card.
- **Core Interface.** Lets the controller chip copy data from the microSD card directly to the Core. This makes it easy to load large amounts of image, palette, and tilemap data from files.
- **Boot menu.** Lets you load a new game onto the Arduino or reprogram the FPGA with an updated Core image.

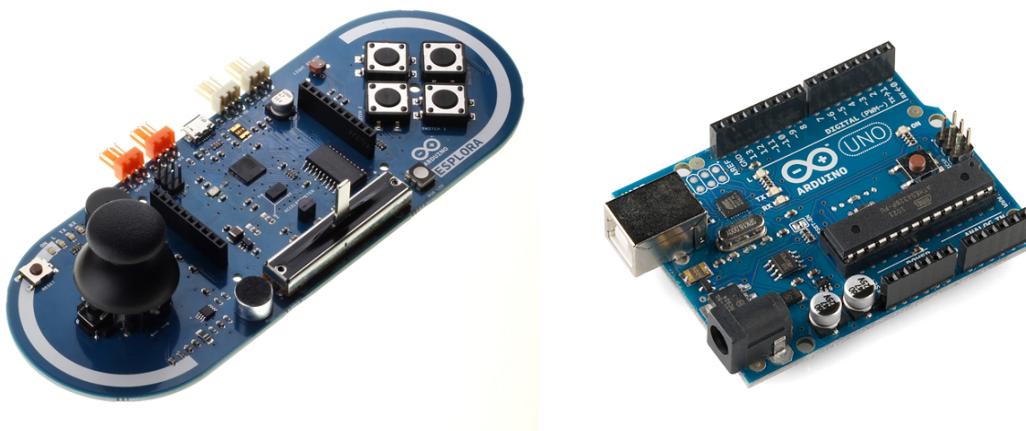
Features found on only the Basic UI Shield:

- **USB Host Controller and USB port.** Plug in a USB gamepad. The controller chip handles reading the gamepad over USB.

Features found only on the Esplora UI Shield:

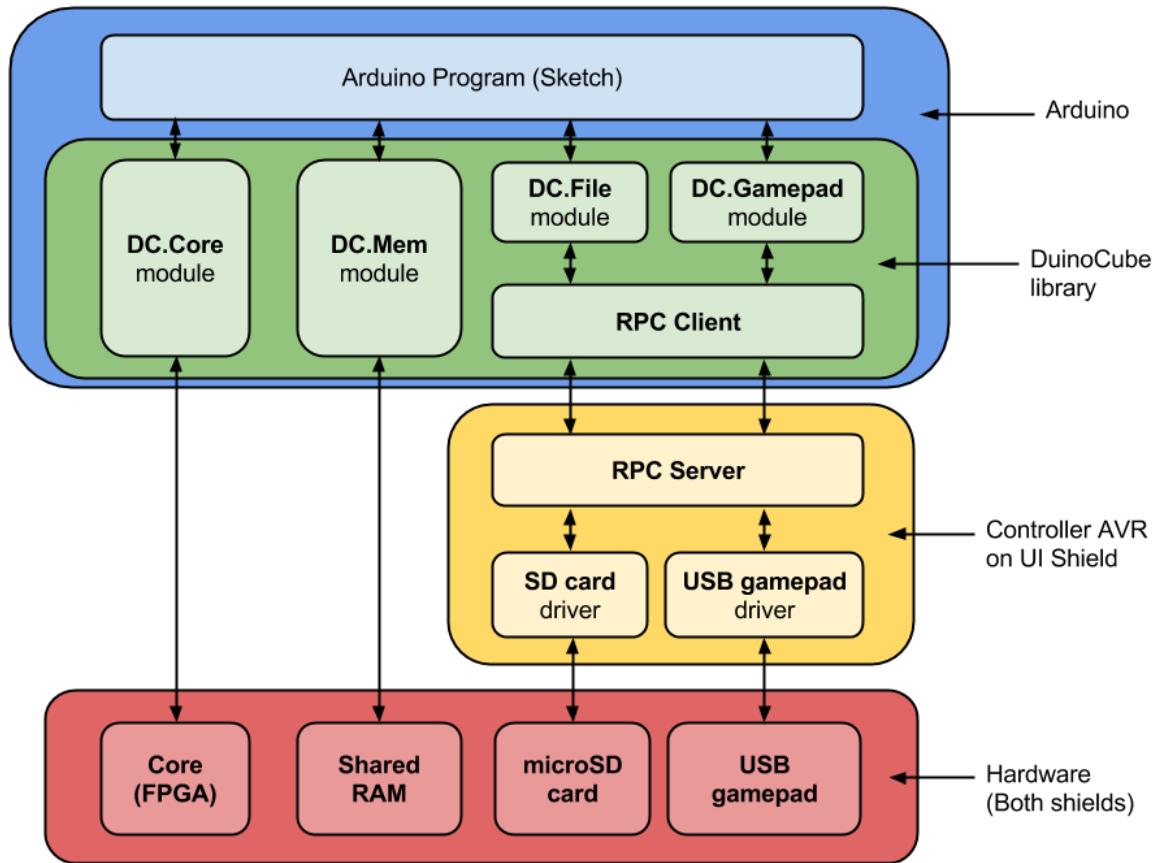
- **Standard Arduino shield adapter.** The Arduino Esplora has a pin header interface that is not physically compatible with standard Arduino shield pin headers like that of the GFX shield. The Esplora UI shield solves this problem by mapping the Esplora pins to a standard Arduino pin interface.

These images show the contrast between an Esplora header interface and a standard header interface:



Software architecture

As you have seen by now, the hardware architecture is rather complex. Fortunately there is a DuinoCube software library that hides the complexity behind a simple API. See the software block diagram below.



The software library forms a bridge between an Arduino program and the DuinoCube hardware. For Core and shared memory access, the library can directly access those hardware components. But accessing SD card and USB gamepad are more complex and require more program space.

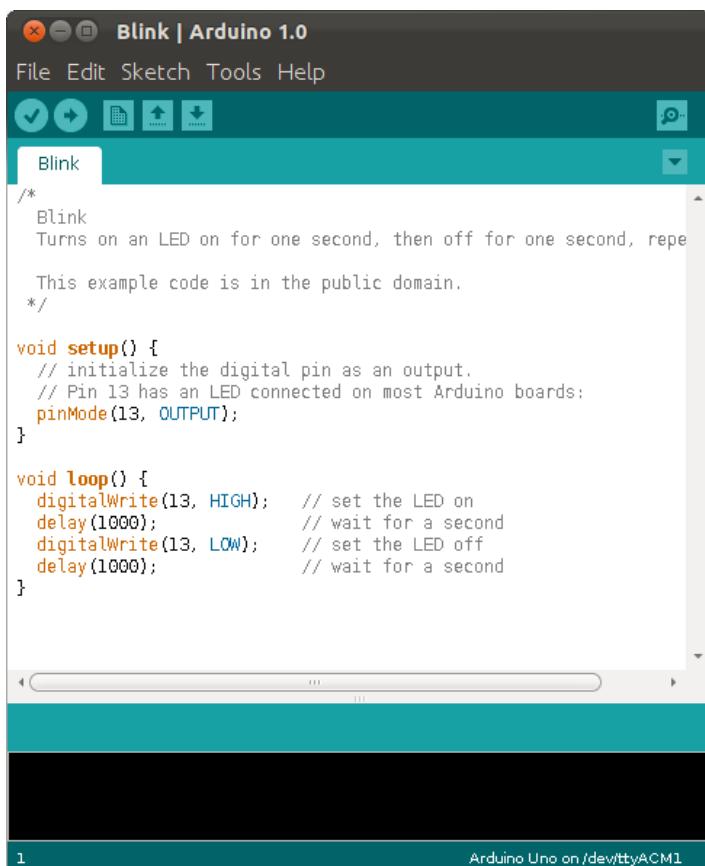
This is where the controller chip of the UI shield comes in. It acts as a **remote procedure call (RPC)** server and the DuinoCube library is the client. An RPC protocol allows a program running on one processor (client) to call a function that runs on a different processor (server). Thus there is no actual driver code for SD card or USB gamepad in the DuinoCube library itself. This frees up much of the program space on the Arduino.

The diagram above does not show all the interfaces between the components. For example, the **DC.Core** module has functions that call into the **DC.File** module; and the RPC client and server use shared RAM to pass data back and forth. The block diagram thus shows only the relationships that are important to the DuinoCube developer.

Setup Guide

Getting the Arduino IDE

Download and install the **Arduino development software (IDE)**. You can find it at:
<http://arduino.cc/en/main/software>



The screenshot shows the Arduino IDE interface with the title bar "Blink | Arduino 1.0". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with icons for Open, Save, Print, and others. The main editor window displays the "Blink" sketch. The code is as follows:

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeating.
  This example code is in the public domain.
*/

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);      // set the LED on
  delay(1000);                // wait for a second
  digitalWrite(13, LOW);       // set the LED off
  delay(1000);                // wait for a second
}
```

The status bar at the bottom indicates "1" and "Arduino Uno on /dev/ttyACM1".

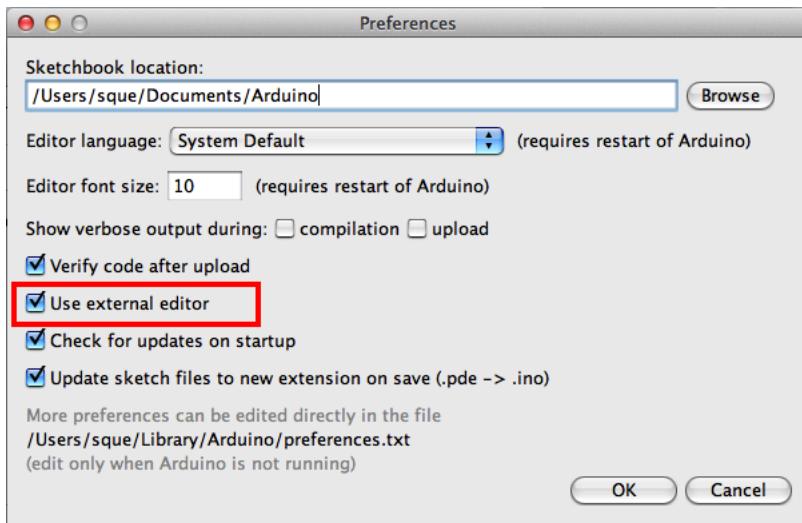
The Arduino IDE has a source code editor that supports C++. The editor is great for beginners but once you get into more advanced programming, you will need a more powerful and flexible tool. For this reason, I recommend using a separate source editor from the get-go. You will still have to use the Arduino IDE to compile and upload your code.

Here are some popular source editors:

- **Notepad++**. My top pick. A lightweight yet powerful GUI-based editor. Officially only for Windows, but can be run on Linux using the Wine emulator. <http://notepad-plus-plus.org/>
- **Eclipse**. One of the most popular GUI-based editors. <https://www.eclipse.org/>
 - There is also an Eclipse Plugin for Arduino:
http://playground.arduino.cc/Code/Eclipse#Eclipse_and_additional_plugins

- **Komodo Edit.** A good GUI-based choice for Mac OS X.
<http://komodoide.com/komodo-edit/>
- **emacs.** A classic GNU editor that is popular among Linux users.
<http://www.gnu.org/software/emacs/>
- **vim.** Another classic editor that relies heavily on keyboard commands. It also has a GUI mode called gVim. <http://www.vim.org/>

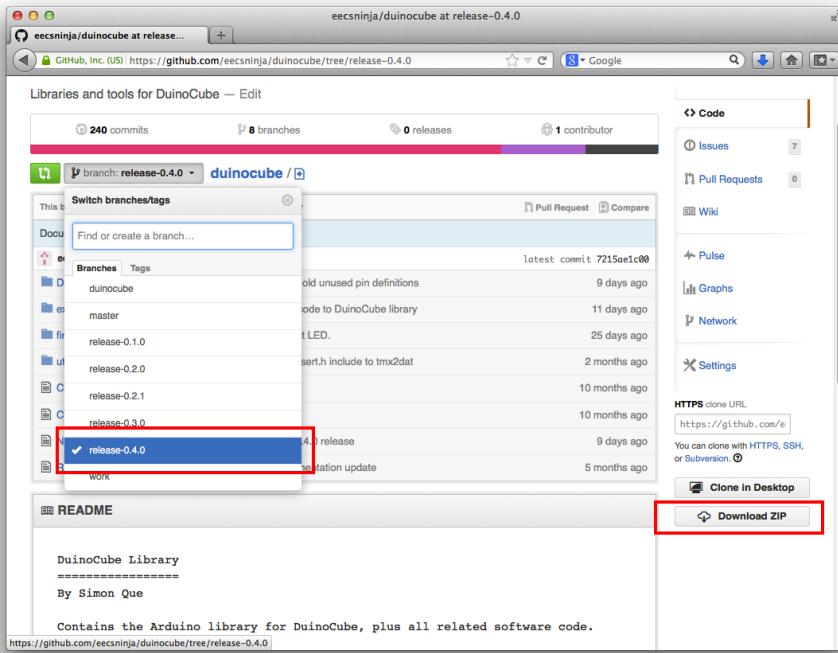
Once you've installed an external editor, notify the Arduino IDE of your decision. Open the Preferences menu in the IDE and check the "Use external editor" checkbox:



By setting this option, the Arduino IDE will reload the code from disk before compiling. (Otherwise, the IDE will compile the code that it is currently showing, which may not be the most recent code.)

Installing the DuinoCube library

To run DuinoCube sketches, you will have to download the DuinoCube code from GitHub: <https://github.com/eecsnilja/duinocube/>. Select the most recent release branch (0.4.0 as of this writing) and click "Download ZIP."



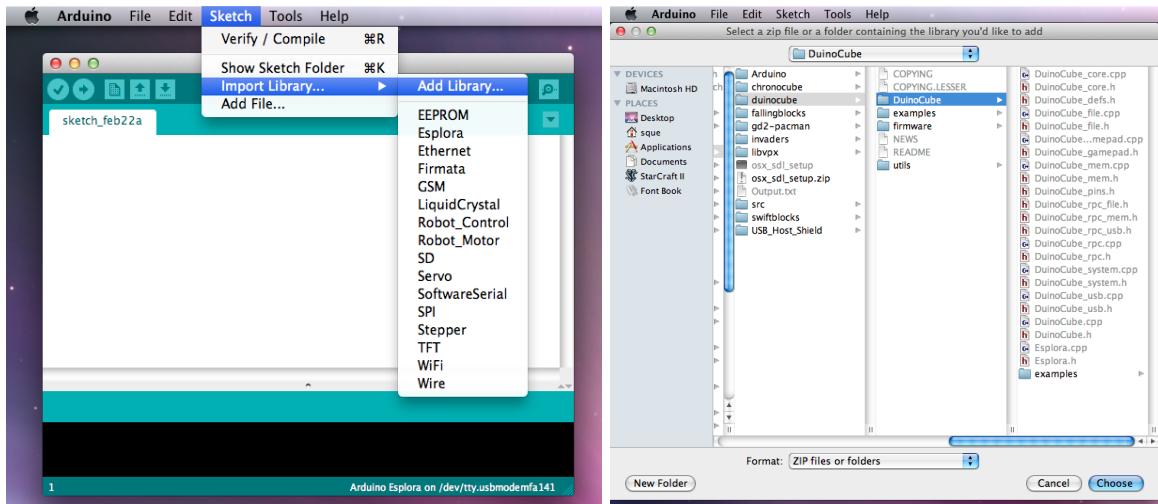
Advanced users can also get it by cloning the repository using git:

```
git clone https://github.com/eecsninja/duinocube.git
```

Check out the appropriate branch:

```
sque:duinocube $ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/release-0.1.0
  remotes/origin/release-0.2.0
  remotes/origin/release-0.2.1
  remotes/origin/release-0.3.0
  remotes/origin/release-0.4.0
sque:duinocube $ git checkout -b release-0.4.0 remotes/origin/release-0.4.0
```

The DuinoCube library is located in the DuinoCube/ folder of the code repository you just downloaded. Import the DuinoCube library into the Arduino environment as shown below. Be sure to select the DuinoCube/ library directory inside the duinocube repository.



Note that the DuinoCube library gets copied to a separate directory by the Arduino IDE. It is located under the following paths:

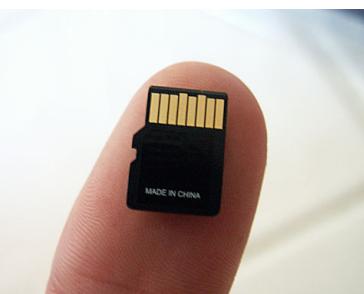
- Windows: C:\Users\<username>\Documents\Arduino\libraries\DuinoCube
- Linux: /home/<username>/Documents/Arduino/libraries/DuinoCube
- Mac OS X: /Users/<username>/Documents/Arduino/libraries/DuinoCube

If you update the DuinoCube library or download a newer version, be sure to delete the directories indicated above before importing the newer version.

Hardware setup

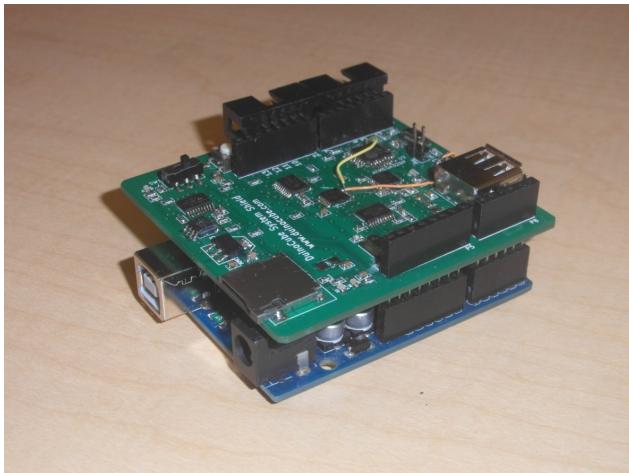
In addition to the DuinoCube UI and GFX shields, you will also need the following hardware to get started:

	<p>Compatible Arduino Board (Uno, Mega, Due, or Esplora). It should come with a USB cable.</p>
	<p>VGA cable. A VGA-to-DVI cable also works if your monitor has a DVI input. TBD: RCA cable for composite video</p>

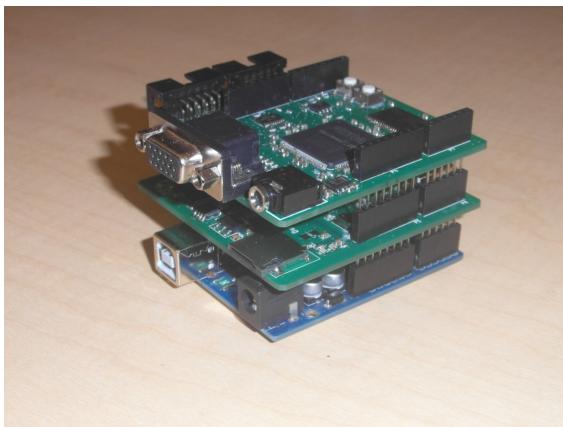
	Monitor or TV with supported input: VGA or DVI.
	Speakers or headphones. Optional for DuinoCube operation. <u>WARNING: Be extremely careful when using headphones. Make sure the sound is not too loud before putting them on.</u>
	Generic USB gamepad. Not needed if you have an Esplora.
	MicroSD card. You will also need some kind of microSD card reader for your computer. DuinoCube's file system driver only supports short filenames of the 8.3 format. Filenames that don't conform to the format will not appear with the correct name.
	10-pin IDC ribbon cable. Connects the data bus between the UI and GFX shields. Included with UI and GFX shield pair.

Follow these steps to assemble DuinoCube.

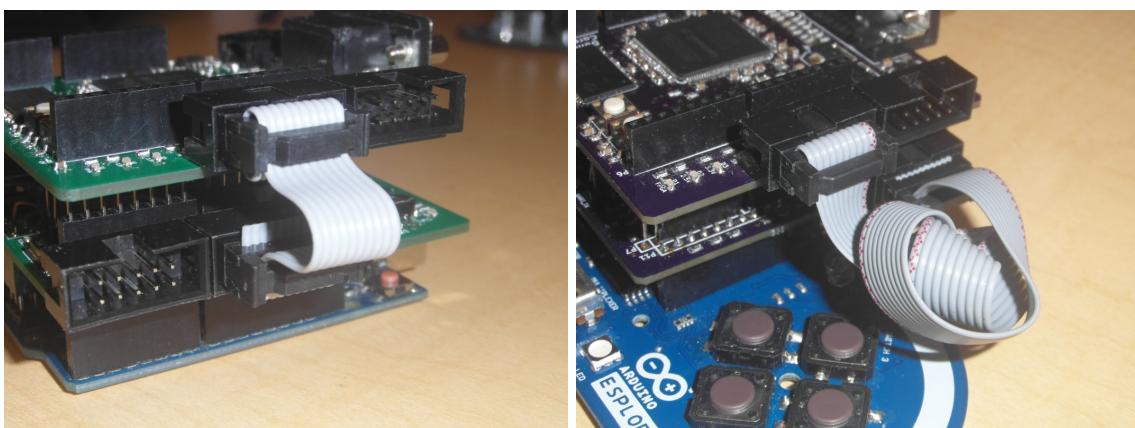
1. Attach the UI Shield to the Arduino board. Make sure all the headers line up: the inline headers on the sides and the 6-pin ISP header in between them.



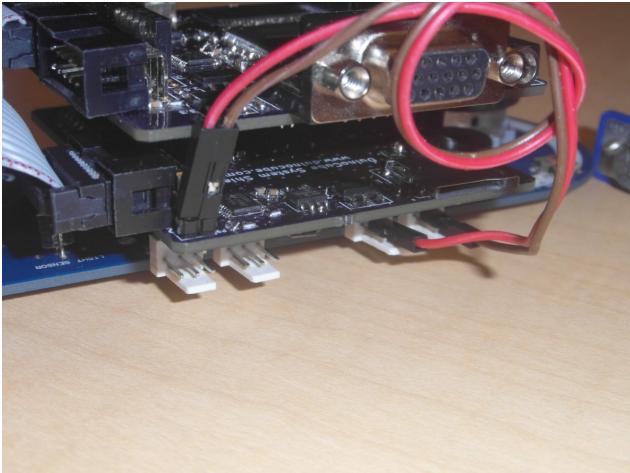
2. Stack the GFX Shield on top of the UI Shield.



3. Connect the two shields with the ribbon cable. The inter-shield connector on the GFX shield is the one that's not near the corner.



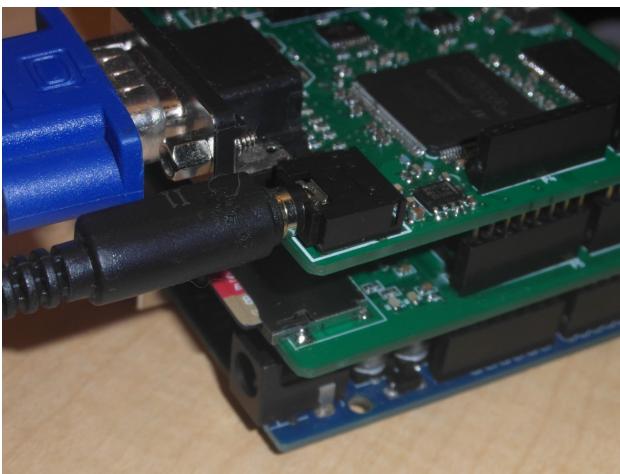
4. (Esplora only) Connect the two leftmost TinkerKit connectors on the Esplora board.



5. Connect the GFX Shield's VGA output to the input of the monitor or TV, using the VGA cable.



6. (optional) Connect the audio output to the speakers.



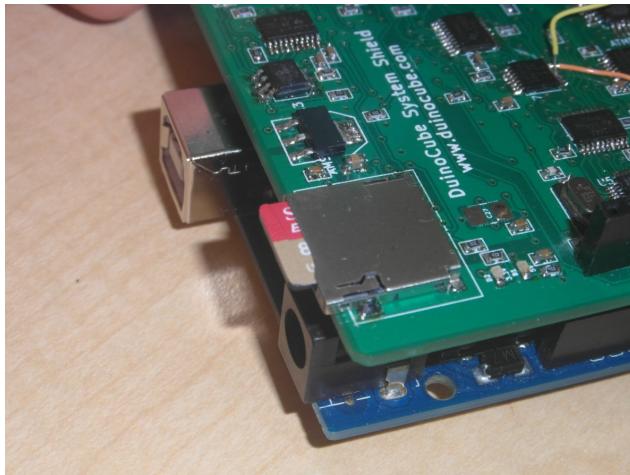
7. (not for Esplora) Attach the USB gamepad to the USB port on the UI Shield.



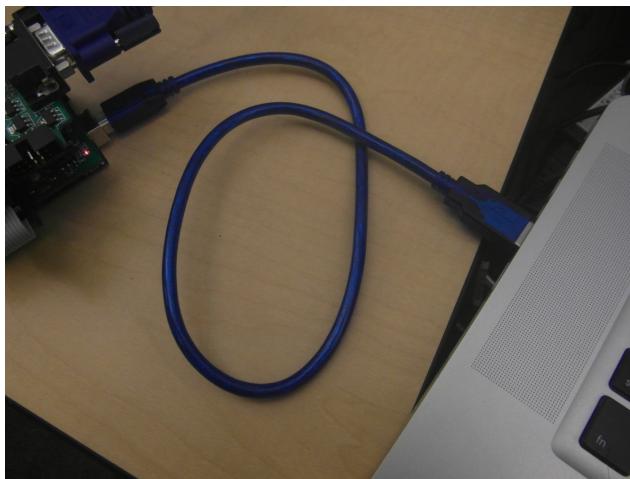
8. Insert the microSD card into the reader connected to your computer.
 - a. Format the microSD card using FAT32.
 - b. Load the contents of the zip file onto the SD card.
 - i. TODO: Add zip file.
 - c. Eject the microSD card.



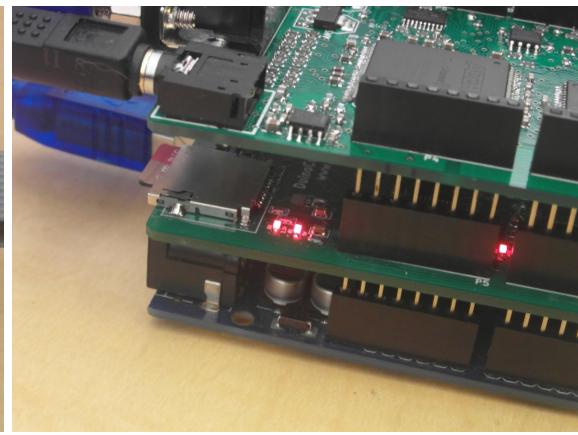
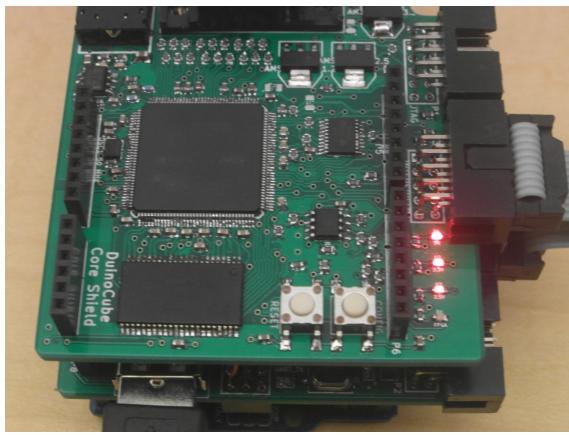
9. Insert microSD card into the microSD slot on the UI Shield. The GFX shield has been removed in the photo for visibility.



10. Connect the Arduino to your computer's USB port using the USB cable that came with it.



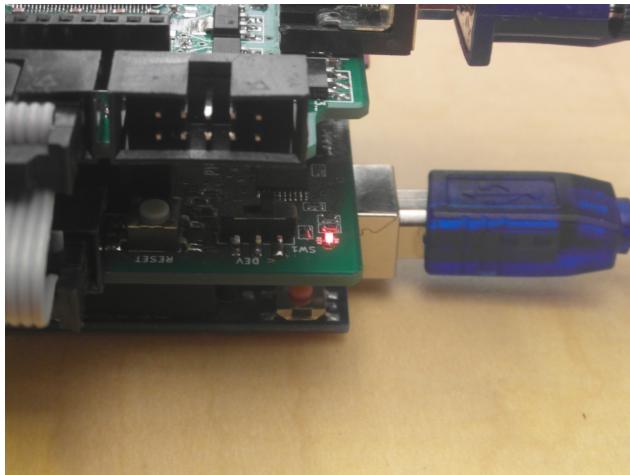
11. Make sure the power LEDs are turned on on both of the shields.



12. On the monitor or TV, select the VGA input. Make sure the display has picked up a valid video signal.



13. Set the switch on the UI shield labeled "DEV" in the direction indicated by the label. The "DEV" LED should turn on. This switch enables developer use of the Arduino while the UI shield is connected. Otherwise the UI shield may reset the Arduino on its own.



References:

http://en.wikipedia.org/wiki/8.3_filename

Loading a sketch

Programs written for Arduino are called **sketches**. Some facts about sketches:

- The main sketch source file has the **.ino** extension.
- Sketches are written in C++, with a few Arduino-specific rules.
- A sketch file must be placed in a directory with the same name. For example, if you have **Blink.ino**, it must be in a directory called **Blink**.
- When your sketch programs get big, you can break them into separate source files located in the same directory. These source files have the **.cpp** extension like normal

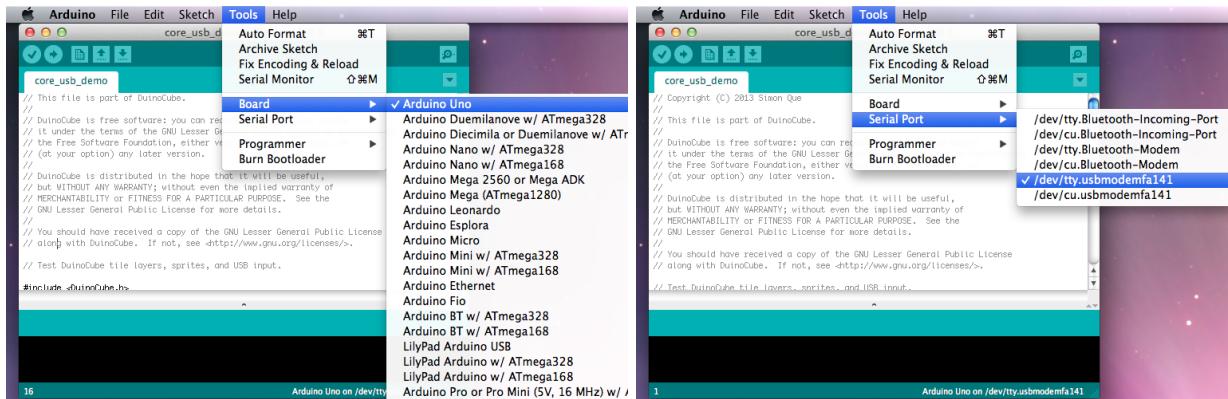
C++ files. They are automatically picked up by the Arduino IDE when you open the main sketch file.

Let's try loading a DuinoCube example sketch: **core_usb_demo**. It tests out both the graphics and the USB gamepad input. Open the sketch from the menus: **File > Examples > DuinoCube > core_usb_demo**.

Once you have the sketch open in the IDE, you will have to tell the IDE what Arduino board you have connected. Under the Tools menu, select the board name and the serial port.

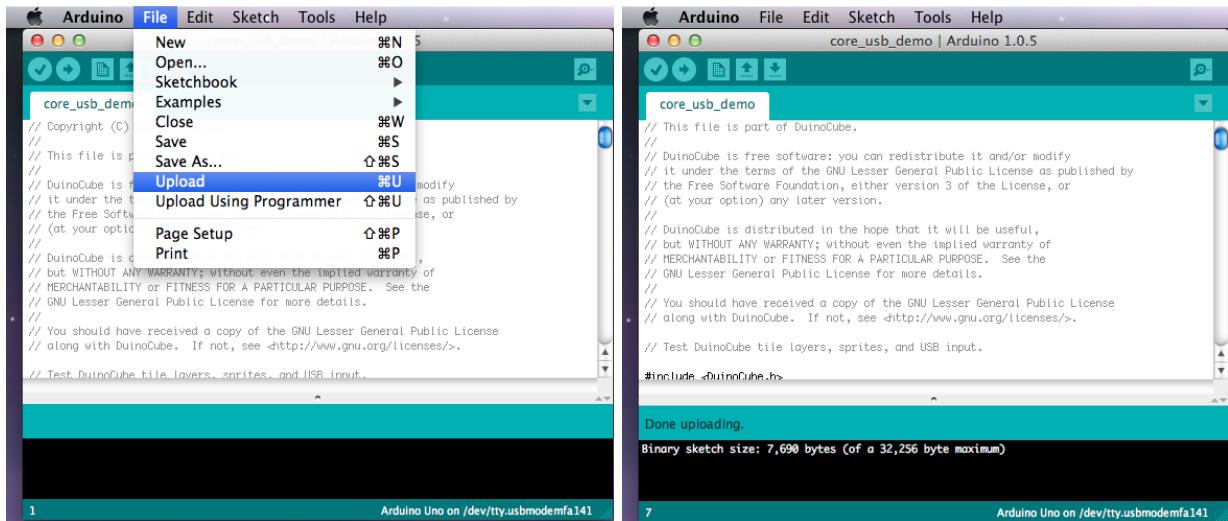
- On Windows, it will show up as a COM port. Look under your list of devices (Start Menu > Devices and Printers) to see which COM port it's on.
- On Linux and Mac OS X, it is called **/dev/tty.usbmodemXXXXXX**, where "XXXXXX" is some assigned letter/number combination. For older Arduino models, the name might be slightly different.

See these screenshots for an example:



If you're having trouble finding the right serial port, disconnect your Arduino from the computer and see which port(s) disappear. Then reconnect your Arduino and select one of the ports that reappear.

Now you're ready to upload the program. Select **File > Upload**. The keyboard shortcut is **Ctrl+U** or **Command+U**, depending on your operating system. The sketch will take about 10-20 seconds to compile and upload to the Arduino. When the upload is complete, the IDE will display the message "Done uploading."



Now your sketch is running on the Arduino! Use the gamepad's joystick or D-pad to move the sprite around the world. The camera will move to follow your sprite. Here's a screenshot of the demo in action:



Programming Guide

Now that you have loaded a sketch and seen it running on DuinoCube, it's time to dig into the code. This section will explain how to write a simple game-like demo. It uses core_usb_demo as an example.

Arduino programming model

The Arduino compiler replaces the traditional `main()` function with a pair of functions `setup()` and `loop()`. `setup()` runs once and then `loop()` is called repeatedly. This is because in a simple system like an Arduino, there is no operating system. The program never exits because it has nowhere to go. Hence, it ends up executing `loop()` repeatedly.

This shows the Arduino program structure and the equivalent C/C++ program:

<pre>void setup() { // Run setup code. } void loop() { // Code to run repeatedly. }</pre>	<pre>void setup() { // Run setup code. } void loop() { // Code to run repeatedly. } int main() { setup(); while (true) { loop(); } return 0; }</pre>
--	--

However, this model is designed for relatively simple code. For games, it needs to be slightly more complex, as we will see later.

Furthermore, your code will have to use `#include` to pick out dependent libraries. In the main `.ino` file of your sketch, the `#include` directive has two purposes:

- Includes the header file of that library (same as in regular C++)�．
- Tells the compiler to build and link that library. In regular C++, you would have to set the linker options manually.

Setting up the game

Every sketch for DuinoCube needs to #include the DuinoCube and SPI libraries:

```
#include <DuinoCube.h>
#include <SPI.h>
```

The SPI library is required by the DuinoCube library to access all the hardware using the SPI interface.

Below is the setup() function of core_usb_demo:

```
void setup() {
    Serial.begin(115200);

    DC.begin();

    load();
    draw();
}
```

Here's a breakdown of each part of the setup code.

```
Serial.begin(115200);
```

Enable the Arduino serial console to transmit and receive at a baud rate of 115200. This is used for logging debug messages to the console in the Arduino IDE. A sketch can run fine without it. But you will find it very useful to be able to print debug messages.

```
DC.begin();
```

Initialize the DuinoCube library. Call this before calling any other DuinoCube library functions. It resets the hardware in the system to start afresh. It also sets up the printf() function to send prints over the serial console.

```
load();
```

Calls a local helper function to load all the palettes, tilemaps, and images used in the demo.

```
draw();
```

Draw all the on-screen layers and objects for the first time.

Loading the graphics data

The following code specifies the files on the SD card from which to load image, palette, and tilemap data. Nothing fancy here, just arrays of filename strings.

```
const char* kImageFiles[] = {
    "data/tileset.raw",
    "data/clouds.raw",
    "data/sprite32.raw",
};

const char* kPaletteFiles[] = {
    "data/tileset.pal",
    "data/clouds.pal",
    "data/sprites.pal",
};

const char* kTilemapFiles[] = {
    "data/desert0.lay",
    "data/desert1.lay",
    "data/desert2.lay",
    "data/clouds.lay",
};
```

Here is the data loading function. It goes through each type of data: palette, tilemap, and image data, in that order.

```
static void load() {
    for (int i = 0; i < ARRAY_SIZE(kPaletteFiles); ++i) {
        const char* filename = kPaletteFiles[i];
        if (!DC.Core.loadPalette(filename, i)) {
            printf("Error loading palette from file %s.\n", filename);
            continue;
        }
        *palettes[i] = i;
    }

    for (int i = 0; i < ARRAY_SIZE(kTilemapFiles); ++i) {
        const char* filename = kTilemapFiles[i];
        if (!DC.Core.loadTilemap(filename, i)) {
            printf("Error loading tilemap from file %s.\n", filename);
            continue;
        }
    }
}
```

```

}

uint32_t vram_offset = 0;
for (int i = 0; i < ARRAY_SIZE(kImageFiles); ++i) {
    const char* filename = kImageFiles[i];
    uint32_t size_read = DC.Core.loadImageData(filename, vram_offset);
    if (size_read == 0) {
        printf("Could not open file %s.\n", filename);
        continue;
    }
    *vram_offsets[i] = vram_offset;
    vram_offset += size_read;
}
}

```

For each type of data, there is a library function to load it from a file into the Core:

- DC.Core.loadPalette()
- DC.Core.loadTilemap()
- DC.Core.loadImageData()

See the API reference for more details.

One thing you might have noticed is the error handling. If there was an error reading a file, the library loading function will return false (or zero bytes in the case of loadImageData()). This way, you can inspect the serial console for any error messages.

Also note the ARRAY_SIZE() macro, which returns the number of elements in a static array. It is defined in the DuinoCube library to make programming easier.

When loading image data, you also have to keep track of the image data addresses yourself. For example, suppose you want to load a 1 KB file (0x400 bytes) to the location 0x1000 in VRAM, and then load a second file into VRAM after it. You will have to store the return value of loadImageData(), which is 0x400, and add it to 0x1000 to get the next VRAM address to pass to loadImageData() when you call it on the second file.

Finally, note this code, which is used to assign a variable to each chunk of palette and image data:

```

// The order of these should match the order of image data being Loaded.
static uint16_t landscape_offset, clouds_offset, sprites_offset;
static uint16_t* vram_offsets[] = {
    &landscape_offset,
    &clouds_offset,
    &sprites_offset,
}

```

```

};

// The order of these should match the order of palette data being loaded.
static uint8_t landscape_pal, clouds_pal, sprites_pal;
static uint8_t* palettes[] = { &landscape_pal, &clouds_pal, &sprites_pal };

```

This code allow the image data addresses and palette indexes to be assigned to variables with names. Named variables are easier to keep track of than numbered elements in an array.

Defining strings in Arduino code

In the previous section, you saw some code that contained text string definitions. For more efficient string definitions, you need to use a special directive called **PROGMEM**.

When you define a string in Arduino code, the string text is stored in program memory (flash), of which the Arduinos have at least 32 kB. However, at startup time, the strings are copied from program memory into data memory (RAM) as static variables. It is these static variables that get referenced when the code attempts to print a string. Thus a string definition also requires program memory.

On the Uno and Esplora, there is only 2 kB and 2.5 kB of RAM, respectively. Loading strings into RAM at startup time is an inefficient use of a limited resource. For smaller programs, it's not a problem. But when you have dozens of strings, each with 20-50 characters, the space requirements can quickly add up.

One common solution is to store the strings in program memory without loading it into RAM at startup. Do this by adding the **PROGMEM** directive after a string definition.

```

#include <stdio.h>
#include <string.h>

#include <avr/pgmspace.h>      // Required for PROGMEM.

static const char kFilename[] PROGMEM = "input.dat";
static const char kErrorString[] PROGMEM =
    "Could not open file: %s\n";

```

To use a **PROGMEM** string, copy it to a local buffer when your program needs to use it locally. Use `strncpy_P()` to perform the copy.

```
char buf[256];
```

```

// Copy the string from program memory to local RAM. Limit
// the copy size to the size of the buffer.
strncpy_P(buf, kFileName, sizeof(buf));
bool success = !DC.Core.loadPalette(buf, MAIN_PALETTE_INDEX));

```

To print a PROGMEM string to the console, use `printf_P()`:

```

// Continuation of above code example.
if (!success) {
    // This should print: "Could not open file: input.dat"
    // buf still contains the filename string.
    printf_P(kErrorString, buf);
}

```

When defining an array of strings, define the individual strings as variables first.

```

static const char kPaletteFile0[] PROGMEM = "tileset.pal"
static const char kPaletteFile1[] PROGMEM = "clouds.pal"
static const char kPaletteFile2[] PROGMEM = "sprites.pal"

static const char* kPaletteFiles[] =
    { kPaletteFile0, kPaletteFile1, kPaletteFile2 };

for (int i = 0; i < ARRAY_SIZE(kPaletteFiles); ++i) {
    char buf[256];
    strncpy_P(buf, kPaletteFiles[i], sizeof(buf));
    DC.Core.loadPalette(buf, i);
}

```

Here the individual strings are stored in only program memory, but the array of their pointers is loaded into normal RAM. We could also add the `PROGMEM` directive to `kPaletteFiles`. But it would require extra code to read the array from program memory. All that work just to save six bytes of RAM in the static space (each address value is two bytes). It isn't worth the trouble. In contrast, the strings are 11-12 bytes each, including the null terminator. Adding the `PROGMEM` directive saves 35 bytes of RAM.

Just remember: code space is cheaper than RAM.

Drawing the tile layers and sprites

With all the data loaded, it's time to enable the drawing. The `draw()` function draws both the tile layers and sprites. In our program, there are four layers and one sprite.

Before turning on the rendering of these objects, the camera needs to be reset and the sprite rendering depth needs to be set.

In the DuinoCube rendering space, there are two types of coordinates:

- **World coordinates**: The coordinates of the virtual world that is being rendered. These are the coordinates relative to the grid of the virtual world. The locations of tile layers and sprites are specified in world coordinates. Objects that are stationary should have unchanging world coordinates.
- **Screen coordinates**: The location of the objects on the screen. The **camera** indicates where the top-left corner of the screen is in world coordinate space.

During initialization in our program, the camera is set to (0, 0):

```
DC.Core.moveCamera(0, 0);
```

The rendering order of the sprites and layers needs to be specified. The tile layers are rendered in order of their indexes. The `DC.Core.setSpriteDepth()` function inserts all the sprites after a particular layer, so that the sprites are rendered immediately on top of it and below the next layer.

In our program, the sprites are above all layers except for the clouds:

```
DC.Core.setSpriteDepth(CLOUD_LAYER - 1);
```

Every tile layer or sprite is either **enabled** or **disabled**. They are disabled by default.

- **enabled**: The layer or sprite is drawn by the Core's renderer.
- **disabled**: The layer or sprite is not drawn by the Core's renderer.

To draw a layer or sprite, it first needs to be configured by setting its properties and location. Then, call `DC.Core.enableTileLayer()` or `DC.Core.enableSprite()` to set it from disabled to enabled.

Here is the code to draw each of the tile layers:

```
for (int layer = 0; layer < ARRAY_SIZE(kTilemapFiles); ++layer) {
    uint8_t palette = (layer == CLOUD_LAYER) ? clouds_pal : landscape_pal;
    uint16_t offset = (layer == CLOUD_LAYER) ? clouds_offset : landscape_offset;

    // Set Layer properties.
    DC.Core.setTileLayerProperty(layer, TILE_PROP_FLAGS,
        (TILE_FLAGS_ENABLE_EMPTY |
        TILE_FLAGS_ENABLE_TRANSPIRANCE |
        TILE_FLAGS_ENABLE_FLIP));
```

```

DC.Core.setTileLayerProperty(layer, TILE_PROP_PALETTE, palette);
DC.Core.setTileLayerProperty(layer, TILE_PROP_EMPTY_VALUE, EMPTY_TILE);
DC.Core.setTileLayerProperty(layer, TILE_PROP_TRANSP_VALUE, COLOR_KEY);
DC.Core.setTileLayerProperty(layer, TILE_PROP_DATA_OFFSET, offset);
DC.Core.moveTileLayer(layer, 0, 0);

// Turn the Layer on.
DC.Core.enableTileLayer(layer);
}

```

The code sets the properties of each tile layer before enabling it for rendering. These properties are necessary to render the layers properly. They include:

- Which color palette to use for.
- The value of an empty tile in the tilemap.
- The transparent pixel value in the tile image data.
- The offset in VRAM at which tile image data is stored.

See the API reference for a complete list of tile layer properties that can be set.

Similarly, here is code for rendering the player's sprite for the first time:

```

// Set to 32x32 size.
DC.Core.setSpriteProperty(PLAYER_SPRITE, SPRITE_PROP_WIDTH, SPRITE_SIZE_32);
DC.Core.setSpriteProperty(PLAYER_SPRITE, SPRITE_PROP_HEIGHT, SPRITE_SIZE_32);
// Set image data offset.
DC.Core.setSpriteProperty(PLAYER_SPRITE, SPRITE_PROP_DATA_OFFSET,
                        sprites_offset);
// Set transparency.
DC.Core.setSpriteProperty(PLAYER_SPRITE, SPRITE_PROP_TRANSP_VALUE, COLOR_KEY);
// Set location.
DC.Core.moveSprite(PLAYER_SPRITE, 0, 0);
// Set palette.
DC.Core.setSpriteProperty(PLAYER_SPRITE, SPRITE_PROP_PALETTE, sprites_pal);
// Set flags.
DC.Core.setSpriteProperty(PLAYER_SPRITE, SPRITE_PROP_FLAGS,
                        SPRITE_FLAGS_ENABLE_TRANSP);

// Enable the sprite.
DC.Core.enableSprite(PLAYER_SPRITE);

```

Once again, the code specifies various rendering properties before enabling the sprite for rendering. These properties are similar to the tile layer properties. But there are some properties of sprites that don't exist for tile layers:

- Size: Each dimension can be 8, 16, 32, or 64 pixels.
- Orientation: Flip the sprite horizontally, vertically, diagonally, or a combination of these.

See the API reference for more details on sprite parameters.

The tile layer and sprite code described here does not need to be called every cycle. It only needs to be called again except to update the rendered object. For example, to change its location or to change the VRAM data offset to point at a new set of images.

The main game loop

The main body of a game program is the main loop. This is a loop that runs continuously until some exit condition is reached. Even if the game is waiting for the user to select a menu option, the game loop is running and polling for the user input. It does not block on waiting for user input because it still needs to update certain aspects of the game like background music and animation.

In the Arduino programming model, the `loop()` function itself is usually insufficient to serve as the game loop. Variables like the player's location and the current score need to sit outside the main loop. In the Arduino model, they will have to be declared as global variables. And it is poor programming practice to have many global variables that are used in a local scope.

An alternative is to have an inner loop within the `loop()` function. This code lets you have a large number of game loop variables defined outside the game loop itself but still as local variables:

```
void loop() {
    // Define some game loop variables here.
    bool done = false;
    while (!done) {
        // Actual game loop goes here.
    }
}
```

That said, here are the loop variables in `core_usb_demo`:

```
// Initialize the player sprite location and image address.
int16_t player_x = 0;
int16_t player_y = 0;
```

```

uint16_t player_offset = sprites_offset;

// Current camera location.
int16_t scroll_x = 0;
int16_t scroll_y = 0;

// Keep a copy of the previous gamepad state to detect button
// press and release events.
GamepadState prev_gamepad;
prev_gamepad.buttons = 0;
prev_gamepad.x = 0;
prev_gamepad.y = 0;

// Keep track of changes in orientation.
uint16_t old_flip_flags = SPRITE_FLIP_NONE;

// Player movement speed based on gamepad input.
int8_t dx = 0;
int8_t dy = 0;

// Adjustable sprite rendering depth relative to tile layers.
uint8_t sprite_z = CLOUD_LAYER - 1;

// Counter for moving the clouds.
uint16_t movement_count = 0;

```

These variables will be described in more details as we look at their usage in the main loop.

Here is the main loop, with large sections removed for brevity:

```

bool done = false;
while (!done) {
    // Wait for visible, non-vblanked region to do computations.
    DC.Core.waitForEvent(CORE_EVENT_VBLANK_END);

    // Perform game logic. (not shown here)

    // Wait for Vblank.
    DC.Core.waitForEvent(CORE_EVENT_VBLANK_BEGIN);

    // Update rendering parameters. (not shown here)
}

```

First there is a `done` flag. This flag is initialized to false. The game loop ends when it is set to true. This could be when the player exits the game to go to the start menu, for example.

Inside the loop, the code is divided into two sections.

The first section performs the game logic: read user input and update the objects and counters in the game. The game logic takes place during the **visible period** of the monitor's refresh cycle, when it receives the image data that is being displayed.

The second section updates the rendering to reflect the changes made in the first part of the code: scroll the camera, move the sprite, update sprite image and orientation, etc. This takes place during the **vertical blanking period** of the monitor's refresh cycle, when it transitions to the next frame. By updating the rendering settings during the vertical blanking period, you can avoid tearing effects.

References:

http://www.brainbell.com/tutors/A+/Hardware/Basic_Monitor_Operation.htm
http://en.wikipedia.org/wiki/Screen_tearing

Reading user input

During the visible period, the main loop reads the current state of the gamepad:

```
GamepadState gamepad = DC.Gamepad.readGamepad();
```

The gamepad has four thumb buttons on the right side. core_usb_demo uses these to change the orientation of the player's sprite. Here, only one orientation is shown. The others are not shown for brevity:

```
uint16_t new_flip_flags = old_flip_flags;
if ((gamepad.buttons & (1 << GAMEPAD_BUTTON_1)) &&
    !(prev_gamepad.buttons & (1 << GAMEPAD_BUTTON_1))) {
    new_flip_flags = SPRITE_FLIP_NONE;
}
```

gamepad.buttons is a bit field that contains the state of the gamepad buttons. A 1 bit means the button is pressed. The above code sets an orientation if it detects that button #1 is pressed. Note that it only sets the orientation when the button state goes from unpressed to pressed. This way, holding down the button won't cause the code to interpret it as a series of repeating presses.

The next part uses the L1 and R1 buttons to change the image of the sprite. The different images are arranged in memory consecutively. The code below shows how to interpret the gamepad input to cycle through the images.

```

int sprite_image_index =
    (player_offset - sprites_offset) / SPRITE_SIZE;
if ((gamepad.buttons & (1 << GAMEPAD_BUTTON_L1)) &&
    !(prev_gamepad.buttons & (1 << GAMEPAD_BUTTON_L1))) {
    --sprite_image_index;
}
if ((gamepad.buttons & (1 << GAMEPAD_BUTTON_R1)) &&
    !(prev_gamepad.buttons & (1 << GAMEPAD_BUTTON_R1))) {
    ++sprite_image_index;
}
// Adjust for valid image index values.
sprite_image_index =
    (sprite_image_index + NUM_SPRITE_IMAGES) % NUM_SPRITE_IMAGES;
player_offset = sprites_offset + sprite_image_index * SPRITE_SIZE;

```

The L2 and R2 buttons of the gamepad control the rendering depth of the sprite. By pressing L2 and R2, the player can move the sprite higher or lower in the stack of tile layers:

```

// L2 and R2 buttons to change sprite Z-level.
if ((gamepad.buttons & (1 << GAMEPAD_BUTTON_R2)) &&
    !(prev_gamepad.buttons & (1 << GAMEPAD_BUTTON_R2)) &&
    sprite_z < NUM_TILE_LAYERS - 1) {
    ++sprite_z;
} else if ((gamepad.buttons & (1 << GAMEPAD_BUTTON_L2)) &&
           !(prev_gamepad.buttons & (1 << GAMEPAD_BUTTON_L2)) &&
           sprite_z > 0) {
    --sprite_z;
}

```

The directional pad or joystick moves the sprite. The directional pad state is stored as two signed integers: `gamepad.x` and `gamepad.y`, where 0 means no direction pressed or neutral position. The code shown here is for the X-axis. The Y-axis has similar code.

```

if (gamepad.x < 0) {
    // Use acceleration.
    if (prev_gamepad.x != gamepad.x)
        dx = -1;
    else if (dx > -MAX_MOVEMENT_SPEED)
        --dx;
    player_x += dx;
}
else if (gamepad.x > 0) {
    // Use acceleration.
    if (prev_gamepad.x != gamepad.x)

```

```
dx = 1;
else if (dx < MAX_MOVEMENT_SPEED)
    ++dx;
player_x += dx;
}
```

The code is a little more complex than just increasing or decreasing the player's location. It applies acceleration so that the player sprite moves faster the longer you hold down a direction, up to a maximum speed. This looks a lot smoother than just having one speed.

The above code also detects the first time the user has pressed the gamepad in that direction. That's when it resets the acceleration value to 1 or -1.

After all this, the gamepad state needs to be saved so it can become compared to the next cycle's gamepad state:

```
prev_gamepad = gamepad;
```

And we don't want the sprite to move off the screen. There's some code to adjust the camera scrolling so that it moves with the sprite when the sprite is on the edge of the screen:

```
if (player_x < scroll_x)
    scroll_x = player_x;
else if (player_x >= scroll_x + SCREEN_WIDTH - SPRITE_WIDTH)
    scroll_x = player_x + SPRITE_WIDTH - SCREEN_WIDTH;

if (player_y < scroll_y)
    scroll_y = player_y;
else if (player_y >= scroll_y + SCREEN_HEIGHT - SPRITE_HEIGHT)
    scroll_y = player_y + SPRITE_HEIGHT - SCREEN_HEIGHT;
```

The cloud layer needs to be moved in its own direction. The following code updates the cloud layer offset. It uses the movement counter divided by 8 and 16 along the two axes:

```
// Update the cloud movement.
uint16_t clouds_x = (movement_count / 8);
uint16_t clouds_y = -(movement_count / 16);
movement_count += MOVEMENT_STEP; // Has the value of 8.
```

Each frame cycle, the clouds move one pixel to the right. Every other frame, the clouds move one pixel up, due to dividing MOVEMENT_STEP by 16.

That was a lot of game logic. In a real game, there is a lot more to keep track of. You will be better off writing your helper functions to handle the various parts, so that your main loop code doesn't become too long and unmaintainable.

Here is the last part of the main loop, the update code that runs during the vertical blanking period. It just updates the rendering settings using all of the parameters that were updated during the visible period.

```
// Scroll the camera.  
DC.Core.moveCamera(scroll_x, scroll_y);  
  
// Scroll the cloud layer independently.  
DC.Core.moveTileLayer(CLOUD_LAYER, clouds_x, clouds_y);  
  
// Update the sprite.  
if (new_flip_flags != old_flip_flags) {  
    DC.Core.setSpriteProperty(PLAYER_SPRITE, SPRITE_PROP_ORIENTATION,  
        new_flip_flags);  
    old_flip_flags = new_flip_flags;  
}  
DC.Core.setSpriteProperty(PLAYER_SPRITE, SPRITE_PROP_DATA_OFFSET,  
    player_offset);  
DC.Core.moveSprite(PLAYER_SPRITE, player_x, player_y);  
DC.Core.setSpriteDepth(sprite_z);
```

API Reference

The DuinoCube library is divided into several modules:

- DC.Core: Graphics and audio core functions.
- DC.File: File system (SD card) functions.
- DC.Gamepad: Gamepad input functions.
- DC.Mem: Shared memory functions.

To use it, include the DuinoCube library at the start of your code as follows:

```
#include <DuinoCube.h>
```

Top-level module

void begin();

Initializes DuinoCube. Run this as part of your `setup()` routine.

Example:

```
void setup() {  
    DC.begin();  
    ...  
}
```

printf()

The library sets up `printf()` to write logging over the Serial interface. Just use `printf()` as you normally would.

Core module

System control functions

void moveCamera(int16_t x, int16_t y);

Sets the view camera to a pixel offset (x, y) relative to world coordinates.

Example:

```
DC.Core.moveCamera(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2);
```

```
void waitForEvent(uint16_t event);
```

Loops until one of the events indicated by `event` has occurred.

Supported events:

- `CORE_EVENT_VBLANK_BEGIN`: The start of the vertical blanking period.
- `CORE_EVENT_VBLANK_END`: The end of the vertical blanking period.
- `CORE_EVENT_HBLANK_BEGIN`: The start of the horizontal blanking period.
- `CORE_EVENT_HBLANK_END`: The end of the horizontal blanking period.

These events can be combined using the bitwise OR operator.

Example:

```
void loop() {  
    DC.Core.waitForEvent(CORE_EVENT_VBLANK_END);  
    ... // Do game logic while previous frame is drawn.  
  
    DC.Core.waitForEvent(CORE_EVENT_VBLANK_BEGIN);  
    ... // Update rendering objects when blanked.  
}
```

Data loading functions

```
bool loadPalette(const char* filename, uint8_t palette_index);
```

Loads palette data from file to a palette indicated by `palette_index`.

Returns true if successful. Returns false if file could not be opened or read.

Example:

```
#define BG_PALETTE      0  
if (!DC.Core.loadPalette("angels.pal", BG_PALETTE)) {  
    printf("Could not load palette!\n");  
}
```

```
bool loadTilemap(const char* filename, uint8_t tilemap_index);
```

Loads tilemap data from file to a tilemap indicated by `tilemap_index`.

Returns true if successful. Returns false if file could not be opened or read.

The tilemap is 32x32 tiles and each tile is 2 bytes, unless in text mode. The total expected size is $32 * 32 * 2 = 2 \text{ kB}$ per tilemap.

Example:

```
#define MAZE_TILE_LAYER      1
if (!DC.Core.loadTilemap("maze.map", MAZE_TILE_LAYER)) {
    printf("Could not load tilemap!\n");
}

uint32_t loadImageData(const char* filename, uint32_t vram_offset);
```

Loads image data from file to a video RAM offset given by `vram_offset`.

Returns the number of bytes read, or 0 if there was an error.

Example:

```
uint32_t vram_offset = 0x1200;
uint32_t size_read =
    DC.Core.loadImageData("sprites.raw", vram_offset));
if (size_read == 0) {
    printf("Could not load image data!\n");
}
vram_offset += size_read;
```

Tile layer functions

```
void enableTileLayer(uint8_t layer_index);
```

Enables a tile layer given by `tile_index`.

Example:

```
#define TEXT_LAYER_INDEX      3
DC.Core.enableTileLayer(TEXT_LAYER_INDEX);
```

```
void disableTileLayer(uint8_t layer_index);
```

Turns off the tile layer given by `tile_index`.

Example:

```
DC.Core.disableTileLayer(cloud_layer_index);
```

```
void moveTileLayer(uint8_t layer_index, int16_t x, int16_t y);
```

Sets the tile layer given by `tile_index` to the location (`x`, `y`). When the location is (0, 0), the top left corner of the tilemap lines up with the top left corner of the world.

Example:

```
DC.Core.moveTileLayer(layer_index, layer_x, layer_y);

void setTileLayerProperty(uint8_t layer_index, uint16_t property,
                      uint16_t value);
```

Sets a property of the tile layer given by `tile_index` to a particular value.

Property types:

The flags can be a combination of these:

- `TILE_PROP_FLAGS`: Bit flags that can be a bitwise OR combination of these:
 - `TILE_FLAGS_ENABLE_TEXT`: For text-based tiling (8x8, 8-bit)
 - `TILE_FLAGS_ENABLE_TRANSP`: Use one color as the transparent color.
 - `TILE_FLAGS_ENABLE_FLIP`: Allow the upper bits of each tile value to be interpreted as selecting a tile orientation.
- `TILE_PROP_PALETTE`: The index of the palette to use for rendering the layer.
- `TILE_PROP_DATA_OFFSET`: The beginning of the tile image data in VRAM.
- `TILE_PROP_TRANSP_VALUE`: The value to use for transparent pixels, if enabled.
- `TILE_PROP_EMPTY_VALUE`: The value of a tile slot that is empty.

Example:

```
DC.Core.setTileLayerProperty(current_layer,
                             TILE_FLAGS,
                             TILE_ENABLE_TEXT |
                             TILE_ENABLE_TRANSP);

DC.Core.setTileLayerProperty(current_layer,
                             TILE_DATA_OFFSET,
                             vram_offset);

DC.Core.setTileLayerProperty(current_layer,
                             TILE_TRANSP_VALUE,
                             transparent_pixel_value);
```

Sprite functions

```
void setSpriteDepth(uint8_t depth);
```

Sets the rendering depth of all sprites to `depth`. The depth value can range from 0 to 3. They are defined relative to the tile layers. e.g. if depth is set to 2, then the sprites are rendered immediately after tile layer #2 has been rendered.

Example:

```
#define WORLD_MAP_LAYER_INDEX      0
#define CLOUD_LAYER_INDEX          1
#define TEXT_LAYER_INDEX           3

// Set sprites to be above the world map but below the
// clouds and text.
DC.Core.setSpriteDepth(WORLD_MAP_LAYER_INDEX);

void enableSprite(uint8_t sprite_index);
```

Enables a sprite given by `sprite_index`.

Example:

```
for (int index = GHOSTS_INDEX_BEGIN;
     index < GHOSTS_INDEX_BEGIN + NUM_GHOSTS;
     ++index) {
    DC.Core.enableSprite(index);
}

void disableSprite(uint8_t sprite_index);
```

Turns off the sprite object given by `sprite_index`.

Example:

```
for (int index = GHOSTS_INDEX_BEGIN;
     index < GHOSTS_INDEX_BEGIN + NUM_GHOSTS;
     ++index) {
    DC.Core.disableSprite(index);
}

void moveSprite(uint8_t sprite_index, int16_t x, int16_t y);
```

Sets the sprite indicated by `sprite_index` to the location (`x`, `y`).

When the location is (0, 0), the top left corner of the sprite object lines up with the top left corner of the world.

Example:

```
DC.Core.moveSprite(enemy.sprite_index, enemy.x, enemy.y);

void setSpriteProperty(uint8_t sprite_index, uint16_t property,
                      uint16_t value);
```

Sets a property of the the sprite object given by `sprite_index` to a particular value.

Property types:

- `SPRITE_PROP_FLAGS`, which can be a combination of these:
 - `SPRITE_ENABLE_TRANSP`: Use one color as the transparent color.
- `SPRITE_PROP_ORIENTATION`: Set the orientation of the sprite. The orientation can be a bitwise OR combination of the following:
 - `SPRITE_FLIP_HORIZ`: Flip along X axis.
 - `SPRITE_FLIP_VERT`: Flip along Y axis.
 - `SPRITE_FLIP_DIAG`: Flip along diagonal X=Y line.
- `SPRITE_PROP_WIDTH`, `SPRITE_PROP_HEIGHT`: The dimensions of the sprite.
 - `SPRITE_SIZE_8`: dimension is 8 pixels.
 - `SPRITE_SIZE_16`: dimension is 16 pixels.
 - `SPRITE_SIZE_32`: dimension is 32 pixels.
 - `SPRITE_SIZE_64`: dimension is 64 pixels.
- `SPRITE_PROP_PALETTE`: Set the index of the palette to use for drawing this sprite.
- `SPRITE_PROP_DATA_OFFSET`: The beginning of the sprite object image data in VRAM.
- `SPRITE_PROP_TRANSP_VALUE`: The value to use for transparent pixels, if enabled.

Example:

```
DC.Core.setSpriteProperty(player_sprite,
                          SPRITE_DATA_OFFSET,
                          vram_offset);

DC.Core.setSpriteProperty(player_sprite,
                          SPRITE_ORIENTATION,
                          FLIP_X | FLIP_Y | FLIP_DIAG);
```

File module

These functions allow access to the files on the SD card. All filenames must follow the old FAT16 format of filenames being limited to 8 characters plus 3 characters for the extension. However, the SD card can be formatted as FAT16 or FAT32.

```
uint16_t open(const char* filename, uint8_t mode);
```

Opens a file and returns the handle, or 0 if the file could not be opened.

Modes:

- FILE_READ: Open an existing file for reading. The file will not be created.

More file access modes will be added in the future, including write capability.

Example:

```
uint16_t file_handle = DC.File.open("input.txt",
FILE_READ);
if (!file_handle) {
    return false;
}
printf("File is %ld bytes\n", DC.File.size(file_handle));
char buf[128];
uint32_t size_read = DC.File.read(file_handle, buf,
sizeof(buf));
printf("%ld bytes read.\n", size_read);
DC.File.close(file_handle);

uint32_t size(uint16_t handle);
```

Returns the size of an open file.

Example: See example for `open()`.

```
uint32_t read(uint16_t handle, void* dest, uint32_t size);
```

Attempts to read data from file into a buffer. Returns the number of bytes read.

Args:

- handle: Open file handle.
- dest: Read data into this memory location.
- size: Maximum number of bytes to read.

Example: See example for `open()`.

```
uint32_t write(uint16_t handle, const void* src, uint32_t size);
```

Attempts to write data from buffer into file. Returns the number of bytes written.

Args:

- `handle`: Open file handle.
- `src`: Read data from this memory location.
- `size`: Maximum number of bytes to write.

Not yet implemented.

```
void close(uint16_t handle);
```

Closes an open file indicated by `handle`.

Example: See example for `open()`.

Gamepad module

For accessing a USB gamepad or the Esplora gamepad interface. The same code works on both system configurations.

```
GamepadState readInput();
```

Reads and returns the current state of the gamepad. The return value is a structure with the format:

```
struct GamepadState {
    uint16_t buttons;    // Button states, one bit set for each
                         // button pressed.
    int16_t x, y;        // Position of joystick or D-pad.
};
```

Example:

```
GamepadState input = DC.Gamepad.readInput();

int direction = NONE;
if (input.x < 0)
    direction = LEFT;
if (input.x > 0)
    direction = RIGHT;
if (input.y < 0)
    direction = UP;
if (input.y < 0)
    direction = DOWN;
```

Mem module

These functions are used to access the 32 kB of external shared RAM on the UI Shield. The memory is accessed over the SPI interface. It cannot be simply read from and written to like normal internal memory. Instead, you must copy buffers of data between the external and internal memories.

Additionally, the UI shield controller chip has a heap system for external shared memory. It reserves the first 512 bytes for passing remote procedure call arguments.

The remaining 31.5 kB is available for general use as dynamically allocated memory.

```
void stat(uint16_t* free_size, uint16_t* largest_size);
```

Returns the total free heap memory and size of the largest block of heap memory.

Args:

- `free_size`: Pass in the address of an integer variable. The function will store the total free heap memory in bytes in the variable.
- `largest_size`: Pass in the address of an integer variable. The function will store the largest block of free heap memory in bytes in the variable.

Example:

```
uint16_t free_size = 0;
uint16_t largest_size = 0;
DC.Mem.stat(&free_size, &largest_size);

printf("Total bytes free: %d, largest free block: %d\n",
       free_size, largest_size);

uint16_t alloc(uint16_t size);
```

Attempts to allocate external memory of `size` bytes. Memory is allocated in blocks of 256 bytes, so the size is rounded up to the next multiple of 256 bytes.

If there is available memory, the function returns the starting address of the allocated space. If there is not enough room to allocate the requested size, the function returns 0.

Example:

```
// Should allocate 1024 bytes if available.
uint16_t addr = DC.Mem.alloc(1000);
```

```

    if (!addr) {
        printf("Unable to allocate memory.\n");
    } else {
        // Do something with the memory.
        ...
        DC.Mem.free(addr);
    }

void free(uint16_t addr);

```

Frees a shared memory block that was previously allocated by `alloc()`. If the address does not point to the beginning of an allocated block, the function does nothing.

Example:

See the example for `alloc()`.

```
void read(uint16_t src_addr, void* dest, uint16_t size);
```

Copies a block of data from shared memory to internal memory.

Args:

- `src_addr`: The address of the source data in external shared memory.
- `dest`: A pointer to the destination in internal memory.
- `size`: The number of bytes to copy. Does not check the for the end of shared memory.

Example:

```

struct Rect {
    int x, y;
    int width, height;
};

Rect rect;
// src_addr was previously allocated using DC.Mem.alloc().
DC.Mem.read(src_addr, &rect, sizeof(rect));

```

```
void write(uint16_t dest_addr, const void* src, uint16_t size);
```

Copies a block of data from internal memory to shared memory.

Args:

- `dest_addr`: The address of destination in external shared memory.
- `src`: A pointer to the source data in internal memory.

- `size`: The number of bytes to copy. Does not check for the end of shared memory..

Example:

```
struct Rect {  
    int x, y;  
    int width, height;  
};  
Rect rect;  
rect.x = 30;  
rect.y = 12;  
rect.width = 100;  
rect.height = 200;  
// dest_addr was previously allocated using DC.Mem.alloc()  
DC.Mem.write(dest_addr , &rect, sizeof(rect));
```