Gameduino Reference Manual

<<<to-do: add cover image>>>

Copyright © 2011 by Guy Garnett
guysphotos at me dot com

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ or send a letter to: *Creative Commons, 171 Second Street, Suite 300, San Francisco, CA 94105, USA*.

Contents

Code

Tables

Figures

Resources

8-Bit Gaming with Arduino and Gameduino

Acknowledgements

James Bowman created the Gameduino, and without his help, input, and corrections this document would not exist. Any valuable information in this manual ultimately comes from James, and any errors that were introduced in the process are mine.

The members of the Gameduino-Beta mailing list, particularly Colin Dooley provided valuable feedback and encouragement based on initial drafts of this manual.

Introduction

Modern microcontrollers, like the Atmel AVR processors that power the open-source Arduino boards, are at least as powerful as the 8-bit CPUs that powered arcade video games in the "golden age" of the 1980's. For example, the Zilog Z80 was an advanced 8-bit CPU that also featured limited 16-bit instructions. The Z80 ran at speeds of up to a million instructions per second, and powered classic games such as Pac Man, Donkey Kong, and Galaxian. By comparison, the ATmega328 that is at the heart of the Arduino Uno executes up to 16 million 8-bit instructions per second.

The Gameduino is an Arduino shield that provides VGA video and stereo audio output suitable for "classic" video gaming using custom FPGA hardware. Unlike other Arduino-based video game projects, the Gameduino provides extensive hardware support for foreground and background color graphics, collision detection, and sound generation. The Gameduino uses a simple SPI interface so it is adaptable to many different microcontrollers, and is powerful and flexible enough that it can be used for significant non-game applications.

Game Programming with Gameduino

Getting Started

Hardware

To do anything useful with a Gameduino, you will need a compatible microcontroller, a Gameduino board, as well as a monitor, speakers, and some sort of game controller or other input device. If you want to get started with the example code and sketches in this book, in addition to your computer, you will need:

- An Arduino Uno or Mega2560
- A Gameduino shield, and
- A SparkFun joystick shield; plus
- Speakers or headphones for sound output, and
- A monitor that supports 800x600 @ 72Hz and an analog VGA input.

The Hardware Sources text box below suggests some places where you can obtain the components mentioned in this book. Other hardware configurations are certainly possible – since this is an Arduino project, creative or home-built solutions are encouraged.

Resource 1: Hardware Sources

Arduino boards: http://arduino.cc/en/Main/Buy

Gameduino shield: To be determined!

Joystick shield kit: http://www.sparkfun.com/products/9760
Bare joystick shield board: http://www.sparkfun.com/products/9490

<<<To-do: reference a section on building your own interface hardware>>>

Hardware Assembly

Arrange the monitor and speakers in your work area so that they are accessible but out of the way. You will want to place the Arduino and Gameduino hardware somewhere convenient, so that you can use the joystick board to test your game sketches while keeping it connected to your computer and the monitor and speakers. Then put everything together:

1. Check the shields for bent pins, and gently straighten any that are out of place.
2. Stack the Gameduino shield on top of the Arduino, and push the shield's pins into the header. Check for bent pins and make sure the shield is securely seated.

3. Stack the joystick shield on top of the Gameduino, again checking for bent pins and ensuring that the shield is firmly seated. It would not do to have the stack come apart while playing your game sketch.
4. Connect the monitor and speakers to the ports on the Gameduino.

Smoke Test

A "smoke test" is the initial test of a new hardware system – the idea is to make sure that everything works (and doesn't have any short circuits or other show-stoppers) before doing anything more complicated. To test your hardware:

1. Power-on the monitor and speakers.
2. Boot up your computer. Once the computer is running,
3. Connect your computer to the Arduino using a USB cable (or optionally, you can use a wall-power adapter suitable for the Arduino).
4. Press the reset button on the joystick shield.

You should get some output from the Gameduino on the monitor and speakers. What you get will depend on what is loaded onto the system, either:

- The Asteroids game sketch will start with the "ASTEROIDS" logo scroll, if your Arduino came with the Gameduino, it should be pre-loaded with this sketch; or
- If you have a "stock" Arduino that doesn't (yet) have any Gameduino-aware sketches loaded onto it, you will get the Gameduino's boot screen, featuring the Gameduino logo and start-up sound.

If you don't get output from the Gameduino, or get something wildly different, try some of the troubleshooting steps below to isolate the problem.

Troubleshooting

<<<To-do: add troubleshooting information>>>

Development Environment

Now that the hardware is set up, you will also need a development environment before you can start creating your own video game sketches. The basic Gameduino development environment consists of:

- A computer running recent versions of either Windows, Mac OS X, or Linux,
- The Arduino integrated development environment (IDE), and
- The Gameduino software development kit (SDK).

Visit the download links listed below to get the software you need.

Resource 2: Software Download Links

Arduino IDE: http://arduino.cc/en/Main/Software
Gameduino SDK: http://excamera.com/sphinx/gameduino/samples/index.html

Installation and Set-Up

Once you have all of the pieces, you can set up your development environment. The exact steps will vary depending on your choice of operating system, but the following steps should serve to get you started:

1. Install the Arduino IDE following the instructions that came with the binary distribution for your platform.
2. Verify that the Arduino IDE is working correctly by launching the IDE and selecting an example sketch from the File menu. Do not attempt to compile or run this example; we are just verifying that the Arduino IDE is working.
3. Unpack the Gameduino SDK distribution by unzipping the Gameduino.ZIP archive file. This will create a "Gameduino" directory with multiple files and subdirectories inside of it. Note the location of the Gameduino directory for use in step 6 below.
4. Locate your Arduino sketchbook directory. The default location will vary depending on your operating system, but should be a subdirectory of your user or home directory. You can find and change the sketchbook location using the Preferences option of the Arduino IDE.
5. If it doesn't already exist, create a directory called "libraries" inside of the Arduino sketchbook directory. If the directory already exists, skip this step.
6. Move the Gameduino directory from step 3 into the libraries directory created in step 5 above.

IDE and SDK Testing

Once everything is installed, you should test the Gameduino development environment before starting to work on your own sketches. To test the tool chain:

1. Quit the Arduino IDE if it is still running, then launch the Arduino IDE. Quitting and re-starting the IDE forces it to re-scan the sketchbook and libraries.
2. Load a Gameduino demo program: go to the File menu, and using the submenus pick Examples, Gameduino, 4.Demo, and finally "ball". The ball sketch should open in a new window.
3. Compile the sketch using the Verify button on the Arduino IDE. It should compile with no errors or warnings.
4. Upload the sketch to the Arduino and run it, by using the Upload button on the IDE. After a brief pause, the bouncing-ball demo should be displayed on the monitor with accompanying audio output on the speakers.

Congratulations! You now have a functioning Gameduino development environment, and you are ready to start writing your first game sketch.

Developing a Game

<<<to-do: add information as the THUD game is developed>>>

Tools and Techniques

Initial Power-Up

Following a hard reset, such as initial power-on, the Gameduino requires a short amount of time to reboot, load the FPGA configuration, and initialize before it is ready to accept commands. This isn't normally an issue when developing code in the Arduino IDE, since downloading a new sketch or resetting the Arduino doesn't force a Gameduino reset. The Gameduino library normally handles this in the GD.begin() method by making sure that the Gameduino is ready to accept commands before initializing the board and returning.

If you are not using the library, your sketch will have to handle this manually. It takes the Gameduino between 175ms and 250ms to complete its power-on initialization. The Gameduino may ignore or fail to fully respond to SPI commands that arrive before this time, so sketches may have unpredictable sound and graphic artifacts until the Arduino is reset. Resetting the Arduino fixes the problem, since the reset forces the Arduino to re-run its setup() method without forcing a hard reset of the Gameduino. However, this solution isn't very practical for final code that is going to be embedded in a device.

Sketch 1: Wait for Gameduino Initialization

```
do {
  GD.wr(RAM_SPRIMG, 0xFF);
} while (GD.rd(RAM_SPRIMG) != 0xFF);
GD.wr(RAM_SPRIMG, 0x00);
```

The best solution to the problem is to ensure that the Gameduino is responding to commands before sending instructions to it. Sketch 1: Wait for Gameduino Initialization shows one such approach using GD library functions for brevity and clarity. This code fragment repeatedly writes a known, non-default value to Gameduino RAM, until attempting to read the RAM results in the value written – demonstrating that the Gameduino has accepted and processed both commands.

Sketch 2: Initialization with Splash Screen

```
void setup() {
  delay(2500);
  GD.begin();
}
```

Another approach is to insert a delay into the code before attempting to communicate with the Gameduino. A delay of at least 250ms will allow the Gameduino enough time to initialize and respond to commands; a longer delay will allow the power-on splash screen to display. As shown by, Sketch 2: Initialization with Splash Screeneven if you are using the Gameduino library, inserting a delay before calling GD.begin() will allow the splash screen to appear at power-up.

Gameduino Python Utilities

<<<to-do: add information about the Python utility package>>>

Conserving Arduino Memory

<<<to-do: add note about PROGMEM from the cookbook page>>>
<<<to-do: add notes from Gameduino-Beta list about conserving memory>>>

Gameduino Resource Library

<<<to-do: add description of the library and how to access it>>>

Other Applications

The Gameduino has a number of powerful non-game applications as a versatile video adapter for Arduino and potentially other microcontrollers. Since it uses standard VGA monitors for information output, it is a cost-effective and versatile display adapter. Compared to most other Arduino output devices, the Gameduino's video display can present a lot of information. A Gameduino can present at least ten times the character information as a typical multi-line LCD display, or almost twice as many on-screen pixels as a QVGA screen.

General-Purpose Display Adapter

As a general-purpose display adapter, the Gameduino VGA screen can contain 37 lines of 50 characters each, which compares well to a typical serial LCD. Serial LCDs commonly used with the Arduino display 1 to 4 lines of 8 to 32 characters. The two devices typically have similar interface requirements: an SPI or serial connection uses a few Ardino pins. Many LCD controllers will also accept downloadable character sets, allowing the programmer to define application-specific glyphs.

Unlike serial display adapters, the Gameduino requires more initialization: a complete character set (even the base ASCII glyphs) must be stored in the host microcontroller's program EEPROM and downloaded to the Gameduino before any data can be output. Most LCD displays include a built-in extended ASCII character set that can be used immediately.

Sprite graphics can be used to implement cursors or pointers. Alternatively, the Gameduino's sprite memory can be arranged as a high-resolution graphics overlay that can be addressed at the single pixel level, as described in the section on Bitmapped Graphics beginning on page 15. This allows an area of arbitrary high-resolution graphics on an otherwise text-based screen.

Industrial Process Control

The Gameduino can also be used for industrial process-control and systems status monitoring applications. Because it has a relatively simple, SPI-based host interface, it can be controlled by nearly any process controller or automation microcontroller that supports SPI communication. Because the entire character set is customizable on the Gameduino, specialized symbols specific to the application can be downloaded and easily displayed.

Character graphics in combination with the Gameduino's sprite graphics can be used to implement gauges and pointers, and provide a unique way of presenting information in an easy-to-understand visual format. For example, a bar graph scale can be built from a small number (3 or 4) of characters, and a sprite can be used as an indicator to visually represent the measurement value.

<<<to-do: insert screenshot and example sketch showing gauges>>>

Advanced Gameduino Programming

Video Timing

The Gameduino generates a SVGA video signal that conforms to the standards for 72Hz 800x600 video with a 50Mhz dot clock. Table 1: 800x600 SVGA Video Output summarizes the critical video timing parameters. See http://tinyvga.com/vga-timing/800x600@72Hz for detailed VESA VGA signal timing information.

Table 1: 800x600 SVGA Video Output

Horizontal Timing	
Pixel Clock	50.0Mhz
Frequency	48.077kHz
Resolution	800 pixels [1]
Active Video	16.0µs
Blanking	4.8µs
Vertical Timing	
Frequency	72Hz
Resolution	600 lines [2]
Active Video	12.480ms
Blanking	1.3728ms

Notes:
[1] Gameduino doubles pixels for a horizontal resolution of 400 double-size pixels.
[2] Gameduino doubles scan lines for a vertical resolution of 300 double-size pixels.

There are two keys to achieving smooth video game animation: frame rates and beam synchronization. Frame rate refers to how often we update the screen information, and beam synchronization dictates when we update that information.

Frame Rate

The Gameduino generates video at 72Hz, or 72 frames per second (fps) no matter what our game programming is doing. However, we can choose how often our program updates the screen, resulting in a logical frame rate that differs from the 72fps hardware frame rate. By updating the display every 2nd, 3rd, or 4th frame, Gameduino software can generate lower-frequency displays. The lower display frequencies allow the game program more computation time between updates, and also reduce the amount of information that has to be transferred each second.

Video displays depend on persistence of vision to smooth out the flicker between different frames. The point where individual frames smooth out varies between person to person, changes with different lighting conditions, and can also depend on the speed of the game or animation. A higher frame rate means more frequent (faster) updates mean that on-screen objects appear to move smoothly. A low frame rate makes on-screen objects appear to flicker or jerk across the screen. An inconsistent or variable frame rate may make things appear to stutter or move erratically. The point where things appear to be smooth is usually somewhere between 30fps and 60fps, but depends greatly on both the content of the screen and the viewer. For example, movie theaters use a 24fps frame rate for most cinematic releases, but still achieve the illusion of smooth continuous motion.

The Gameduino's 72fps hardware video generation is comfortably above this visual limit. Games that can run at this rate should be smooth and flicker-free, so most game sketches should strive for this goal. Software may achieve higher update frequencies, but the hardware that generates the VGA output signal only updates the screen 72 times per second.

A 36fps frame rate, or updates every other frame, is still fast enough to appear smooth under many conditions. Fast-moving objects (like a projectile) or high-contrast items (like white stars on a black background) may appear to flicker or stutter for some viewers. This rate is often a good compromise, since it allows twice the computation time between frames but is still smooth.

The 24fps frame rate is the same as movie projectors, and is the slowest rate that can appear smooth to most viewers. Objectionable flicker, jitter, or other visual artifacts may be apparent under some lighting conditions, on fast-moving objects, or when there is high contrast between sequential frames.

Frame rates of 18fps will appear to flicker, stutter, or have noticeable visual artifacts to most viewers. However, this frame rate allows for plenty of computation time between frames, and may be suitable for certain kinds of games, particularly strategic games that don't feature fast-moving action.

Some kinds of games can benefit from selective updates: fast-moving or high contrast portions of the display should be updated more frequently, while other items are updated less often. For example, a space game might update a scrolling background at 72fps, but could use a 36fps frame rate for sprite animations to save memory and processor time.

The main reason for choosing a logical frame rate lower than 72fps is to allow programs that require extensive computation between frames to complete their work and update the screen. The table below shows how much time is available.

Table 2: Video Frame Rate

				Cycles per Frame		
Rate	Div	Skip	Time	Arduino	SPI	J1
72 fps	1	0	13.88ms	222,222	13,888	694,444
36 fps	2	1	27.77ms	444,444	27,777	1,388,888
24 fps	3	2	41.66ms	666,666	41,666	2,083,333
18 fps	4	3	55.55ms	888,888	55,555	2,777,777

Rate: Logical frame rate, in frames per second (fps).
Div: Frame rate divisor. Logical frame rate = 72fps hardware frame rate ÷ Div.
Skip: Number of frames skipped between updates.

Time: Elapsed time between updates, in milliseconds (ms; 1ms = 1×10^{-3} seconds).
Arduino: Number of instruction cycles between updates assuming a 16Mhz ATmega8 processor. Some AVR instructions take more than one cycle.
SPI: Number of SPI byte transfers between updates assuming a 16Mhz SPI bus clock. The Gameduino SPI interface requires two address bytes to set up a transfer, plus one byte per data byte read or written to Gameduino memory.
J1: Number of instruction cycles between updates for the Gameduino's J1 coprocessor. Most coprocessor instructions take one cycle, although memory read and write instructions take two cycles.

Beam Synchronization

Although the Gameduino's double-ported RAM allows programs to update graphics data without preventing the video generator from fetching data, updating graphics data that is currently being drawn on the screen can cause tearing or other glitches in the video output. For example, if we update a sprite's Y position while it is being drawn on screen, a few lines of the sprite will appear at the old location, and the rest will appear at the new location, making the sprite look like it is torn in half.

<<<to-do: insert torn sprite image>>>

The solution is to update objects when they are not being drawn on the screen. This can be done via double buffering, updating off-screen objects, by waiting for vertical blanking, and by raster chasing techniques.

Double Buffering

One obvious solution to the problem is to keep two copies (buffers) of everything: one is used to draw the display while the other is being updated. When the update is done, the two buffers are swapped and the newly updated copy is used to draw the next frame of the display. The big drawback to double buffering is that it requires a lot of memory, and some way of quickly swapping the two buffers. The Gameduino hardware supports true double buffering for sprite control data, and limited double buffering for the sprite bitmaps and character picture.

Double buffering for sprites is controlled by the SPR_PAGE register at 0x280B (10251 decimal). When SPR_PAGE is 0, sprite control data page 0 is on screen: the data in locations 0x3000 to 0x33FF (12288 to 13311 decimal) is used to generate the display. When page 0 is being displayed, the sprite control data structures on page 1 at locations 0x3400 to 0x37FF (13312 to 14335 decimal) can be updated without causing visual glitches. Setting SPR_PAGE to 1 reverses the situation: page 1 data is used to draw the screen display, and page 0 can be updated.

To implement sprite double buffering in your sketches, assuming that your sketch is fast enough to update the sprites every frame:

1. Initialize sprite control data pages 0 and 1 with the same data, defining the same sprites on each page.
2. Initialize SPR_PAGE to 0.
3. Wait for vertical blanking.

4. Toggle SPR_PAGE: set it to 1 if it is currently 0, or set it to 0 if it was 1.
5. Update sprite positions, bitmaps, and other attributes by changing values in the non-displayed page: if SPR_PAGE is 0, update the sprites starting at 0x3400; if SPR_PAGE is 1 update the sprites starting at 0x3000.
6. Go to Step 3 to draw the next frame.

If your game doesn't use all of the sprite bitmap memory, you can also double buffer sprite bitmaps. Without using this type of double buffering, the vertical blanking period is so short that it is difficult to redefine sprite image data in RAM_SPRIMG before the display is drawn. Because each bitmap is 256 bytes, there is enough time to update 4 or 5 sprite bitmaps if the SPI bus isn't used for anything else during vertical blanking. However, by double buffering sprite bitmaps, changes can be made during the entire video frame, allowing plenty of time to update 32 bitmaps. Bitmaps that aren't currently being used can be changed under program control, and then the sprite source image data (bits 25-30 of the sprite control data structure) can be changed to point to the modified bitmap. For example, a given sprite could use source images 0 and 32. When source image 0 is being used by the sprite, source image 32 can be updated. The sprite's data structure can be changed to swap source image bitmaps during the vertical blanking period, resulting in glitch-free sprite bitmap animation. This scheme works best when there are few enough sprites that two bitmaps can be allocated to each sprite. That way, low-numbered bitmaps (0 to 31) can be updated while high-numbered bitmaps (32 to 63) are on screen, and vice-versa.

Similarly, sketches can redefine the character bitmaps. If the character being redefined isn't present on-screen, the bitmap can be updated at any time. The same low/high scheme can be used with character bitmaps, but unlike sprites, each character on the display must be changed individually during the vertical blanking period to animate the bitmap, which severely limits the effectiveness of the approach.

When designing double-buffering schemes, it is a good idea to maximize SPI data transfer efficiency by keeping all of the data to be updated in a contiguous block. For example, updating 32 sprite bitmaps in a single block takes 8194 SPI cycles (2 to set up the address and 8192 to transfer 32 bitmaps of 256 bytes each). Updating 32 sprite bitmaps individually requires 8256 SPI cycles (32 bitmaps, each of which requires a 2-byte address followed by 256 data bytes), so keeping the data together saves 62 SPI cycles.

Updating Off-Screen Objects

The Gameduino has a logical 512×512 pixel display, but can only show a 400×300 pixel window on the monitor. Information that is not currently visible within the display window can be updated without causing visual glitches. This technique is frequently used by scrolling-screen video games. Portions of the playfield that are currently outside of the visible window are updated, and eventually scroll into the visible window. The game sketch constantly updates the playfield “ahead” of the scrolling.

Vertical Blanking

Video display timing is based around the need to scan an electron beam across a cathode-ray tube. During part of this scan, the beam is turned off and returned to the top of the screen so that it is be ready to draw a new frame of data. The period of time that the beam is turned off and being returned to the top of the screen is the vertical blanking interval. While modern LCD panels don’t use electron beams, the timing for that vertical blanking interval is still built into the VGA video standard.

Since no video is output during the vertical blanking interval, our game software can use this time to update the screen without causing visual glitches on the screen. The vertical blanking interval is nominally 1.3728ms. In a single vertical blanking interval:

- An Arduino’s ATmega CPU running at 16MHz can execute 21,964 machine cycles. Many AVR instructions take one cycle, but some take 2, 3, or 4 cycles.
- The Gameduino’s J1 coprocessor can execute 68,640 machine cycles. Almost all J1 instructions take one cycle, but some (including memory reads and writes) take 2 cycles.
- The Arduino can use its SPI bus can transfer 1,372 bytes, assuming the maximum SPI clock speed. SPI data transfers require sending 2 address bytes to set up a transfer plus 1 byte per data byte read from or written to the Gameduino.

The work that can be done is summarized in Table 3: Vertical Blanking.

Table 3: Vertical Blanking

Vertical Blanking	1.3728ms
Arduino CPU	21,964 cycles
J1 Coprocessor	68,640 cycles
SPI transfer	1,372 bytes

Although in theory all three types of activities can be going on at once, in practice there are some limitations. First, the Arduino CPU is responsible for initiating SPI transactions and keeping the SPI bus busy, so the amount of additional work that it can get done at the same is limited. Similarly, while the J1 coprocessor can execute its instructions independently of the Arduino, each SPI read or write operation steals a machine cycle from the J1, reducing its performance by about 2%. Finally, because of the way that the Gameduino’s video-composition hardware works, the actual time where it is safe to update the screen may vary by up to 2% from the nominal figure. See the discussion under Raster Chasing below for a more comprehensive discussion.

Detecting Vertical Blanking

Arduino sketches and coprocessor code can use the VBLANK register to synchronize their updates to the vertical blanking interval. VBLANK will read 1 when the Gameduino is generating a vertical blanking signal, or 0 otherwise.

The GD library provides a function, waitvblank() that will wait for VBLANK to go from 0 to 1 using a while loop. The general structure of a game sketch that uses waitvblank() is illustrated in Sketch 3: Game Loop using GD::waitvblank() at the right. Using this function guarantees that the Arduino has the maximum amount of time to update the screen: it waits for VBLANK to change from 0 to 1 before returning. If your sketch calls waitvblank() in the middle of a vertical blanking interval, it will wait for the next vertical blanking interval, skipping an entire frame, to ensure that your sketch has the full time available.

Sketch 3: Game Loop using GD::waitvblank()

```
loop()
{
  // Code that does game calculations here.
  waitvblank()
  // Code that updates the Gameduino screen here.
}
```

While this approach is sufficient for many game sketches, it is somewhat wasteful: the Arduino can do nothing except respond to interrupts while it is waiting for the vertical blanking interval to start. On top of that, continuous SPI read cycles used to repeatedly check VBLANK steal a few memory cycles from the coprocessor, causing it to run slightly slower. This can be an issue if your coprocessor code depends on precise timing –for example, if you are also doing raster chasing.

Another way of synchronizing Arduino code with vertical blanking is to have coprocessor microcode that allows the J1 coprocessor interrupt the Arduino at the start of the vertical blanking interval. An <<<example sketch>>> is shown at right. This sketch can be extended by addition additional coprocessor code after the interrupt is issued that will be executed during vertical blanking.

<<<to-do: insert interrupt sketch>>>

Raster Chasing

Just as the video beam needs to be returned to the top of the screen before starting each frame, it also needs to be returned to the left side of the screen prior to each line of video. Horizontal blanking is the period of time that the video signal is turned off while it is being returned to the left side of the screen. Since there are hundreds of lines per frame, the horizontal blanking time of 4.8µs is much shorter than the vertical blanking interval. If the action to be taken is based on the current line, then the technique is known as “raster chasing”. For example, a program could change the background color based on the current line.

However, the Gameduino doesn't output VGA video directly. Instead, each pixel is doubled horizontally and vertically. To do this, the Gameduino composes each line of video into one of two buffers by building it up from the background color, character graphics, and sprites. Once the line is built up in the buffer, it is output to the display, and the other buffer is available to compose the next line. The Gameduino outputs each pixel twice to double their size horizontally, and each line is output twice to double it vertically. There is a brief pause to switch buffers and synchronize the timing of the compositor before the Gameduino begins composing the next line.

So, instead of a standard VGA horizontal retrace, the Gameduino has a small time window between composing successive lines of the display. During this window, it is safe to update on-screen information, since no video composition is going on. Since each line of video can take a different amount of time to composed, based mainly on the number of sprites present in that line, the size of the window can be different from line to line: the window is at least 45 J1 cycles long, and can be as long as 1677 cycles. The time taken depends on the number of sprites that must be composited onto the line: the minimum time of 45 cycles corresponds to the maximum of 96 sprites per line.

There is no direct equivalent to the `VBLANK` register to indicate when it is safe to perform updates, but there *is* coprocessor-only register (`YLINE`, at 0x8000) that indicates the current horizontal line number. The Gameduino increments `YLINE` immediately after the rightmost pixel of a line is composed. J1 programs that wait for `YLINE` to change are guaranteed to have at least 45 cycles, and possibly longer, to make changes to the video data.

`YLINE` is a coprocessor-only register. There is no equivalent available to the Arduino that indicates when horizontal blanking is taking place – and not enough time to take advantage of it even if there were: even a single-byte SPI transfer requires 180 J1 cycles, so the window may well be closed before the data could be written. The coprocessor could interrupt the Arduino when a change in `YLINE` is detected using a program similar to the one described above for vertical blanking, but there is so little time available that it would be challenging to write an interrupt service routine that did anything useful.

The best approach for is to write a coprocessor routine that detects a change in the `YLINE` register does the necessary work. There is no indication when video compositing resumes, so coprocessor programs that update the screen during this window should do their work quickly. The <<example sketch>> to the right illustrates raster-chasing techniques that manipulate the BGCOLOR register to create a sky and ground background effect.

<<<to-do: insert raster-chasing example>>>

YLINE Validity

When coding J1 routines that work with YLINE, remember that the value of YLINE is undefined during the vertical blanking interval. Coprocessor code can count on YLINE going from 0 to 299 during the visible portion of the display, and a value greater than 299 during vertical blanking. However, during vertical blanking the YLINE register may take on unpredictable and arbitrary values above 299. For example, it may count from 0 to 299 during the visible frame, and then count 300, 301, 302, 303, 482, 483, etc. while the display is blanked for vertical retrace.

Bitmapped Graphics

<<<to-do: describe how to use sprites to make a bitmap graphics field>>>

Hardware Hacking

Circuit Board Modifications

Gameduino Modifications

The Gameduino board is designed to use Arduino pin 9 as the select (SEL or SS) pin for SPI communication. However, the Gameduino may need to co-exist with other hardware that uses pin 9 for its own purposes. It has been designed to enable this to be changed if needed by cutting a trace on the circuit board and installing a jumper wire.

Any available digital pin can be used as the new SPI SEL pin, but note that digital pins 11, 12, and 13 are used by the Gameduino's SPI interface, and digital pin 2 may also be in use by some sketches. To reconfigure the Gameduino to use a different pin for SPI SEL:

1. Break the circuit board trace at location X on the photograph, and

2. Solder a jumper wire from the through-hole at location A to the new SPI SEL pin; the exact pin will depend on your system's hardware configuration.

<<<To-do: insert photo here>>>

Joystick Shield Modifications

The SparkFun joystick shield is as close as anything gets to a standard gaming input device for the Arduino. From the point of view of using it with a Gameduino, it has one significant drawback: the joystick pushbutton is attached to Arduino digital pin 2. This may be an issue because pin 2 can be used by Gameduino coprocessor code to interrupt the Arduino, or pin 2 can be used to access the Gameduino's onboard Flash memory. Luckily, it is relatively easy to modify the joystick shield to remap this button to a different input. We recommend remapping the input to pin 7 with the following procedure:

1. Break the circuit board trace at location X in the photograph, and

2. Solder a jumper wire from location B to Arduino digital pin 7 at location C.

<<<to-do: insert photo here>>>

Do-It-Yourself Projects

Inputs and Controllers

<<<to-do: information about building your own controller>>>

Enclosures and Cases

<<<to-do: information about building your own box or case>>>

Gameduino Technical Reference

Hardware Interface

Although designed as an Arduino shield, the hardware is largely self-contained, and can be interfaced with any microprocessor or computer system that can provide power and a SPI interface. An additional output from the Gameduino is connected to Arduino digital pin 2 for use as an interrupt from the Gameduino to the host system. These connections are summarized in Table 4: Host Interface.

Table 4: Host Interface

Arduino Pin	Function
GND	Signal Ground
+3.3V	Supply Voltage, 3.3VDC
+5V	Supply Voltage, 5.0VDC, 23mA
2	Interrupt (optional)
9	SPI SEL or SS
11	SPI MOSI
12	SPI MISO
13	SPI SCK

Hardware Compatibility

The Gameduino is specifically intended for use with the Arduino Uno. It is physically and electrically compatible with most Arduino boards such as the including the Mega 2560, Duemilanove, Mega, and Diecimila. ATmaga168-based boards like Diecimila and earlier revisions of the Duemilanove may not have enough Flash memory or Static RAM (SRAM) for many Gameduino sketches presented in this manual. Arduino-derived designs such as the Freeduino and Seeeduino that incorporate an ATmega328, ATmega1280, or ATmega2560 should also work with the Gameduino. Other development boards with Arduino-compatible headers can also support the Gameduino; some of these like the Netduino, FEZ Domino, or FEZ Panda incorporate considerably more computational power than the standard Arduino platform. The status of known host boards for the Gameduino is presented in Table 5: Hardware Compatibility List.

Table 5: Hardware Compatibility List

Host Board				Compatibility and Support			
Name	Processor	Memory		Hardware	Software	Active	Note
Arduino Uno	16MHz ATmega328	32/2		Yes	Yes	Yes	
Arduino Mega2560	16MHz ATmaga2560	256/8		Yes	Yes	Yes	
Arduino Duemilanove	16MHz ATmega328	32/2		Yes	Yes		
Arduino Duemilanove	16MHz ATmega168	16/1		Yes	Maybe		1
Arduino Diecimila	16MHz ATmega168	16/1		Yes	Maybe		1
Arduino Mega	16MHz ATmega1280	128/8		Yes	Yes		
Freeduino v2.2	16MHz ATmega328	32/2					2
Seeduino v2.2	16MHz ATmega328	32/2					2

Host Board				Compatibility and Support			
Name	Processor	Memory		Hardware	Software	Active	Note
LeafLabs Maple	72MHz ARM Cortex	120/20		Yes	Yes	Yes	4
Netduino	48MHz ARM7	129/60			No		3
Netduino Plus	48MHz ARM7	64/28			No		3
FEZ Domino	72MHz ARM7	148/62			No		3
FEZ Panda	72MHz ARM7	148/62			No		3
FEZ Panda II	72MHz ARM7	148/62			No		3

Host Board: Name of the host board.
Processor: CPU speed and designation, program (Flash) and read/write (SRAM) memory in kbytes.
Compatibility and Support: Status of the Gameduino on each platform, see below.

***Hardware** – Physical and electrical compatibility with the Gameduino:*
- Yes = Fully compatible, connects to host board as a shield
- Maybe = Can be made to work by modifying boards or running wires
- No = Does not work
- Blank = Not tested, information needed.

***Software** – Compatibility with source code included in this book:*
- Yes = Compatible, full GD library support and all sketches work
- Maybe = Some sketches may not work due to memory or timing constraints, see notes
- No = Not Arduino source-code compatible, see notes.
- Blank = Not tested, information needed

***Active** – Availability of GD library and active community support:*
- Yes = Very active community and supported GD library
- Some = Limited activity but some work is being done, see notes
- 3rd Party = Active development community using a 3rd-party GD library, see notes
- No = No activity and no library, probably because of incompatibility
- Blank = Unknown, information needed.

***Note** – System-specific compatibility notes:*
1. These systems are mechanically and electrically compatible, but limited program storage and SRAM may prevent many sketches from working. Untested at this time.
2. Believed to be fully compatible, but untested at this time.
3. Boards based on .Net Micro Framework should be hardware compatible; untested and no community or library support at this time.
4. Boards based on Arduino/Processing IDE with ARM cross-compiler. Software is source-code (but not object-code) compatible.

SPI Communication

The Gameduino's programming interface is a 32-kilobyte address space that is available to the host computer via an SPI interface. The Gameduino expects data to be sent MSB first (DORD=0), using SPI Mode 0 for a clock signal that's low when idle (DORD=0) and sampled on the rising edge (CPHA=0). While the Gameduino should be able to keep pace with an SPI clock up to 16.7MHz, it has only been tested to the Arduino's maximum clock speed of 8MHz.

Library Support

The Arduino environment includes a standard library for handling SPI communications. Sketch 4: SPI Initialization shows the Arduino code to set up the SPI library to communicate with a Gameduino. The GD library for Arduino provides convenient wrapper functions around the SPI library for common Gameduino operations.

Sketch 4: SPI Initialization

```
#define GAMEDUINO_SS 9

pinMode(GAMEDUINO_SS,OUTPUT);
SPI.begin();
SPI.setClockDivider(SPI_CLOCK_DIV2);
SPI.setBitOrder(MSBFIRST);
SPI.setDataMode(MODE0);
```

Alternate SPI Address

The Gameduino board is designed to use Arduino pin 9 as the select (SEL or SS) pin for SPI communication. However, the board has been designed to enable this to be changed if needed; for example, to co-exist with other hardware that uses pin 9 for its own purposes. See Circuit Board Modifications

Gameduino Modifications on page 16 in the Hardware Hacking section for detailed instructions.

Memory Map

These memory maps present the Gameduino address space in ascending order from lowest address to highest address; the charts show addresses increasing from left to right and top to bottom. This means that ASCII character strings are presented in the order that they would be read as English text. For example, the string "HELLO WORLD" would be written to the upper-left corner of the character picture as:

Address		Byte Offset and Memory Contents										
(hex)	(dec)	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10
0x0000	0	0x48	0x45	0x4C	0x4C	0x4F	0x20	0x57	0x4F	0x52	0x4C	0x44
		H	E	L	L	O		W	O	R	L	D

However, when storing 16-bit binary numbers, the Gameduino is a little-endian device: 16-bit values are stored with the least-significant byte in the lower address. For example, the 16-bit value 0x2F8B would be stored in the 2-byte BG_COLOR register starting at 0x280E as:

Symbol		Address		Length		Byte Offset and Contents	
		(hex)	(dec)	(hex)	(dec)	+0	+1
BG_COLOR	RW	0x280E	10254	0x2	2	0x8B	0x2F

Many Gameduino memory locations are bit-mapped, with specific bits or groups of bits within a word having distinct functions. When mapping a byte or word, the individual bits are presented in as digits of a number, with the most-significant bit to the farthest left and the less-significant bits to the right. Individual bits within a multi-bit field are indicated with subscripts, with zero being the least-significant bit and higher numbers indicating progressively more significant bits. For example, B_0 is the least-significant bit in field B, while B_6 is the 7th and most-significant bit in a 7-bit field.

Gameduino Memory Map

Table 6: Gameduino Memory Map

Symbol		Address	Length	Purpose
RAM_PIC	RW	0x0000 0x0FFF	4096	**Character Picture,** character-mapped playfield Grid of 64×64 1-byte characters
RAM_CHR	RW	0x1000 0x1FFF	4096	**Character Bitmaps,** 256 characters 8×8 pixels, 2 bits per pixel (16 bytes/character)
RAM_PAL	RW	0x2000 0x27FF	2048	**Character Palettes,** 256 palettes of 4 colors each ARGB1555 format (2 bytes/color)
IDENT	R	0x2800	1	**Gameduino ID,** always reads 0x6D
REV	R	0x2801	1	**Hardware Revision,** always reads 0x10
FRAME	R	0x2802	1	**Video Frame Counter**
VBLANK	R	0x2803	1	**Vertical Blanking Flag,** 1 during vertical blank
SCROLL_X	RW	0x2804 0x2805	2	**Scroll X**, character picture horizontal scroll Scroll offset, 0 to 511 pixels
SCROLL_Y	RW	0x2806 0x2807	2	**Scroll Y**, character picture vertical scroll Scroll offset, 0 to 511 pixels
JK_MODE	RW	0x2808	1	**Sprite J/K Mode**, 0 to disable, 1 to enable
J1_RESET	RW	0x2809	1	**Coprocessor Reset,** 1 to reset and halt, 0 to run
SPR_DISABLE	RW	0x280A	1	**Sprite Disable,** 1 to disable sprints, 0 to enable
SPR_PAGE	RW	0x280B	1	**Sprite Page Select,** 0 for 0x3000, 1 for 0x3400
IOMODE	RW	0x280C	1	**Pin 2 I/O Mode,** 0x00=disable, 0x45=SPI, 0x4A=J1
	R	0x280D	1	**Unused Register** for future use, always reads 0
BG_COLOR	RW	0x280E 0x280F	2	**Background Color,** for transparent playfield pixels ARGB1555 format, high (alpha) bit is ignored
SAMPLE_L	RW	0x2810 0x2811	2	**Audio Sample,** left channel 16-bit signed audio sample value
SAMPLE_R	RW	0x2812 0x2813	2	**Audio Sample,** right channel 16-bit signed audio sample value
RING_MOD	RW	0x2814	1	**Ring Modulator**, voice used for modulation, affects all lower-numbered voices; 64 (default) to disable.
	R	0x2815 0x281D	9	**Unused Registers,** reserved for future use Always reads 0
SCREENSHOT_Y	RW	0x281E 0x281F	2	**Screenshot Line Select,** 0-311 Selects raster line to load into screenshot buffer
	R	0x2820 0x283F	32	**Unused Registers,** reserved for future use Always reads 0
PALETTE16A/B RAM_SPRPAL16	RW	0x2840 0x287F	64	**Sprite Palette 16,** 2 palettes (A&B), 16 colors each ARGB1555 format (2 bytes/color)

Symbol		Address	Length	Purpose
PALETTE4A/B RAM_SPRPAL4	RW	0x2880 0x288F	16	**Sprite Palette 4**, 2 palettes (A&B), 4 colors each ARGB1555 format (2 bytes/color)
COMM	RW	0x2890 0x28BF	48	**Coprocessor Communications Block** Specific use to be determined by the programmer
		0x28C0 0x28FF	64	**Unused Address Space** Always reads 0
COLLISION	R	0x2900 0x29FF	256	**Sprite Collision Detection,** valid when VBLANK=1 Sprite number of colliding sprite, FF=no collision
VOICES	RW	0x2A00 0x2AFF	256	**Voice Control,** audio synthesis control 64 voices, 4 bytes/voice
J1_CODE	RW	0x2B00 0x2BFF	256	**Coprocessor Microcode** 128 instructions, 2 bytes/instruction
SCREENSHOT	R	0x2C00 0x2F1F	800	**Screenshot Line Buffer**, selected display line data ARGB1555 format (2 bytes/pixel), alpha always 0
	R	0x2F20 0x2FFF	224	**Unused Address Space** Always reads 0
RAM_SPR	RW	0x3000 0x37FF	2048	**Sprite Control,** 2 pages of 256 sprites each 4 bytes/sprite
RAM_SPRPAL RAM_SPRPAL256	RW	0x3800 0x3FFF	2048	**Sprite Palette 256,** 4 palettes of 256 colors each ARGB1555 format (2 bytes/color)
RAM_SPRIMG	RW	0x4000 0x7FFF	16384	**Sprite Image Data** 64 images, 16×16 pixels, 1 byte/pixel
YLINE	R	0x8000	2	**Current Raster Line,** 0-299
ICAP_O	R	0x8002	2	**FPGA ICAP Port,** 8-bit output
		0x8004	2	**Unused Address Space**, always reads 0
ICAP	W	0x8006	2	**FPGA ICAP Port,** 8-bit input and ICAP control bits
		0x8008	2	**Unused Address Space**, always reads 0
FREQHZ	W	0x800A	2	**Timer Frequency**, 0Hz to 65535Hz
FREQTICK	R	0x800C	2	**Timer Tick**, toggles between 0 and 1 at FREQHZ
P2_V	RW	0x800E	2	**Pin 2 Value**, 0 or 1
P2_DIR	R	0x8010	2	**Pin 2 Direction,** 0=output, 1=input
RANDOM	R	0x8012	2	**16-bit Random Value,** from white-noise generator
CLOCK	R	0x8014	2	**16-bit Clock Value**, 50MHz machine cycles
FLASH_MISO	R	0x8016	2	**Onboard SPI Flash,** MISO line
FLASH_MOSI	W	0x8018	2	**Onboard SPI Flash,** MOSI line
FLASH_SCK	W	0x801A	2	**Onboard SPI Flash,** SCK line
FLASH_SSEL	W	0x801C	2	**Onboard SPI Flash,** SSEL line

Blue text indicates unofficial information (e.g. “RAM_SPRPAL4” is an unofficial label for 0x2880)
Lines shaded in grey are coprocessor-only (cannot be accessed by Arduino SPI interface).

Control Registers

The Gameduino is controlled through SPI reads and writes into its address space. This address space can be classified as memory and registers, where registers are one-byte or two-byte memory addresses that have specific functions. In contrast, memory areas are large sections of the address space that have the same function. For example, the Gameduino ID and hardware revision number are registers, while playfield image data and character bitmaps are considered memory.

Shared Registers

The Gameduino has two sets of registers: a group of shared registers at addresses 10240 through 10303 (0x2800 – 0x283F) that is accessible to both the SPI interface and the J1 coprocessor, and another group starting at 32768 (0x8000) that is only accessible to the coprocessor. The shared registers are described below, while the coprocessor-only registers are described in the

J1 Coprocessor chapter, starting on page 53.

Gameduino ID: IDENT is a single-byte, read-only register is used to identify a Gameduino when scanning SPI peripherals. If a Gameduino is present, issuing a read command for address 10240 (0x2800) will produce 109 (0x6D, ASCII "m").

Table 7: IDENT Register

Symbol		Address		Length		Contents	
		(hex)	(dec)	(hex)	(dec)	(hex)	(dec)
IDENT	R	0x2800	10240	0x01	1	0x6D	109

Hardware Version: REV is a single-byte, read-only register that contains the major and minor revision numbers (R and M respectively in a R.M version numbering scheme) of the Gameduino hardware and on-board firmware. R_3-R_0 is the major revision number; M_3-M_0 is the minor revision number.

Table 8: REV Register

Symbol		Address		Length		Contents	
		(hex)	(dec)	(hex)	(dec)	(hex)	(dec)
REV	R	0x2801	10241	0x01	1	$R_3R_2R_1R_0$ $M_3M_2M_1M_0$	R×16 + M

Video Frame Counter: The FRAME register contains a one-byte, read-only count of video frames generated. The count starts at power-on at 0, counts up to 255, then resets to 0. At the Gameduino's 72Hz frame rate, the register wraps around to 0 roughly every 3.56 seconds.

Table 9: FRAME Register

Symbol		Address		Length		Contents	
		(hex)	(dec)	(hex)	(dec)	(hex)	(dec)
FRAME	R	0x2802	10242	0x01	1	0x00-0xFF	0-255

Vertical Blanking Flag: VBLANK is a single-byte, read-only register that is set to 1 during the vertical blanking interval, or 0 otherwise. The waitvblank() method in the Gameduino library uses this flag to determine when to return. There are 72 vertical blanking intervals per second (one per frame of video generated). Each one lasts 1.3728ms. See the Video Timing section starting on page 8 for more on vertical blanking and to avoid visible glitches when updating the screen.

Table 10: VBLANK Register

Symbol		Address		Length		Contents
		(hex)	(dec)	(hex)	(dec)	
VBLANK	R	0x2803	10243	0x01	1	X X X X X X X V_0

V_0: Vertical blanking status: 0=not in vertical blanking, 1=vertical blanking.

Screen Scrolling: The Gameduino outputs a 400×300 pixel screen display window onto a logical 512×512 pixel image. The SCROLL_X and SCROLL_Y registers set the location of the upper left corner of the visible window relative to the larger logical image. If the visible window extends off of the right or bottom of the logical image, it automatically wraps around to the left or top (respectively) of the logical image. For more information on screen scrolling, see the explanation of Scrolling that starts on page 29 below.

Table 11: SCROLL_X and SCROLL_Y Registers

Symbol		Address		Length		Byte Offset and Register Contents		Default
		(hex)	(dec)	(hex)	(dec)	+0	+1	
SCROLL_X	RW	0x2804	10244	0x2	2	$S_7S_6S_5S_4S_3S_2S_1S_0$	X X X X X X X S_8	0x0000
SCROLL_Y	RW	0x2406	10246	0x2	2	$S_7S_6S_5S_4S_3S_2S_1S_0$	X X X X X X X S_8	0x0000

X: Unused bit position, ignored; should be 0 for compatibility reasons.
S_0–S_8: Scroll offset in pixels, 0-511.

Sprite J/K Mode: The JK_MODE register is a one-byte, read/write value that controls the behavior of the Gameduino's sprite collision-detection logic. When set to 0, J/K mode is disabled and collisions are detected between all sprites. Writing a 1 to JK_MODE enables grouped "J/K" collision detection. In this mode, the sprites are divided into two groups (labeled J and K for convenience), and collisions are only detected between sprites in different groups. Sprites that belong to the same group are considered "friendly" with one another and don't collide. Setting bit 31 of the sprite control word to 0 or 1 assigns the sprite to the J or K group, respectively. See the

Table 29: Sprite Palette Modes

Palette Mode P_0-P_3			Selected Mode				Pixel
(bin)	(dec)	(hex)	colors		palette		bits
0000	0	0x0	256	8	0	A	0-7
0001	1	0x1	256	8	1	B	0-7
0010	2	0x2	256	8	2	C	0-7
0011	3	0x3	256	8	3	D	0-7
0100	4	0x4	16	4	0	A	0-3
0101	5	0x5	16	4	1	B	0-3
0110	6	0x6	16	4	0	A	4-7
0111	7	0x7	16	4	1	B	4-7
1000	8	0x8	4	2	0	A	0-1
1001	9	0x9	4	2	1	B	0-1
1010	10	0xA	4	2	0	A	2-3
1011	11	0xB	4	2	1	B	2-3
1100	12	0xC	4	2	0	A	4-5
1101	13	0xD	4	2	1	B	4-5
1110	14	0xE	4	2	0	A	6-7
1111	15	0xF	4	2	1	B	6-7

Palette mode: The selected palette mode number in binary, decimal, and hexadecimal.
Colors: Number of colors and bit depth in selected palette mode.
Palette: The number and letter of the palette selected for the sprite.
Pixel bits: The bit positions of the sprite image used in the selected mode.

Table 30: Sprite Rotation

Rotation R_0-R_2			Selected Mode			Effect
(bin)	(dec)	(hex)	Yflip	Xflip	XYswap	
000	0	0x0	0	0	0	None
001	1	0x1	0	0	1	Mirrored left-to-right then rotated 270°
010	2	0x2	0	1	0	Mirrored left to right
011	3	0x3	0	1	1	270° clockwise rotation

100	4	0x4	1	0	0	Mirrored left-to-right then rotated 180°
101	5	0x5	1	0	1	90° clockwise rotation
110	6	0x6	1	1	0	180° clockwise rotation
111	7	0x7	1	1	1	Mirrored left-to-right then rotated 90°

Rotation: The selected rotation number in binary, decimal, and hexadecimal.
Y flip: Flip sprite image in Y direction (top-to-bottom), 0=no, 1=yes.
X flip: Flip sprite image in X direction (left-to-right), 0=no, 1=yes.
XY swap: Swap sprite X and Y axes, 0=no, 1=yes.
Effect: Effect of selected rotation on the sprite image.

Sprite Collision Detection topic on page 44 for more information.

Table 12: JK_MODE Register

Symbol		Address		Length		Contents	Default
		(hex)	(dec)	(hex)	(dec)		
JK_MODE	RW	0x2808	10248	0x01	1	X X X X X X X J_0	0x00

J_0: Sprite J/K mode collision detection: 0=disabled (detect all collisions), 1=enabled.

Coprocessor Reset: The J1_RESET is a one-byte, read/write register that allows the Arduino (or other host microcontroller) to stop and start the J1 coprocessor. Writing 0x01 to this register halts and resets the coprocessor, while 0x00 releases the coprocessor. Following a reset, the J1 begins executing code starting at J1_CODE, 0x2B00 (11008 decimal).

Table 13: J1_RESET Register

Symbol		Address		Length		Contents	Default
		(hex)	(dec)	(hex)	(dec)		
J1_RESET	RW	0x2809	10249	0x1	1	X X X X X X X R_0	0x00

R_0: Coprocessor control: 0=halt and reset, 1=run.

Sprite Disable: SPR_DISABLE is a one-byte, read/write register that controls the Gameduino's sprite graphics subsystem. The default of 0 allows sprite graphics to function normally. When SPR_DISABLE is set to 1, sprite graphics are disabled and do not appear on screen.

Table 14: SPR_DISABLE Register

Symbol		Address		Length		Contents	Default
		(hex)	(dec)	(hex)	(dec)		
SPR_DISABLE	RW	0x280A	10250	0x1	1	X X X X X X X D_0	0x00

D_0: Sprite graphics disable: 0=sprites enabled, 1=sprites disabled.

Sprite Page Select: The SPR_PAGE is a one-byte, read/write register used to implement double buffering for sprite control data. When SPR_PAGE is 0, the sprite graphics system uses the sprite control words at 0x3000-0x33FF (12288-13311 decimal) to generate the sprite display. When it is 1, the sprite control words at 0x3400-0x37FF (13312-14335 decimal) are used. See the section on Double Buffering starting on page 10 for more information and example code.

Table 15: SPR_PAGE Register

Symbol		Address		Length		Contents	Default
		(hex)	(dec)	(hex)	(dec)		
SPR_PAGE	RW	0x280B	10251	0x1	1	X X X X X X X P_0	0x00

P_0: Sprite graphics control page: 0=0x3000-0x33FF, 1=0x3400-0x37FF.

Pin 2 I/O Mode: The IOMODE register is used to assign pin 2 to one of two functions, or to disable it entirely. The default value of 0x00 causes the Gameduino to ignore pin 2. When set to 0x46 (decimal 70 or ASCII "F"), pin 2 is connected to the Gameduino's onboard SPI flash memory chip as the SEL or SS signal. Finally, when IOMODE is set to 0x4A (decimal 74 or ASCII "J"), the pin is controlled by the coprocessor-only P2_V and P2_DIR registers. Other values have no effect.

Table 16: IOMODE Register

Symbol		Address		Length		Contents	Default
		(hex)	(dec)	(hex)	(dec)		
IOMODE	RW	0x280C	10252	0x1	1	0x00, 0x46, or 0x4A	0x00

IOMODE must be one of the following three values:

- 0x00 (0 decimal, ASCII "NUL") – Pin 2 is ignored.
- 0x46 (70 decimal, ASCII "F") – Pin 2 is Gameduino SPI flash memory SEL.
- 0x4A (74 decimal, ASCII "J") – Pin 2 is controlled by the Gameduino coprocessor.

Background Color: The BG_COLOR register defines a background color that appears behind all of the other video sources. On power-up, the background color is the only image displayed, and defaults to 0x0000 (black). The BG_COLOR value is a standard Gameduino 16-bit color value except that Bit 15, the alpha (A) channel or transparency bit, is ignored if set.

Table 17: BG_COLOR Register

Symbol		Address		Length		Byte Offset and Register Contents		Default
		(hex)	(dec)	(hex)	(dec)	+0	+1	
BG_COLOR	RW	0x280E	10254	0x2	2	$G_2G_1G_0B_4B_3B_2B_1B_0$	X $R_4R_3R_2R_1R_0G_4G_3$	0x0000

X: Alpha channel color information, ignored.
R_4–R_0: Red channel color information, 0-32.
B_4–B_0: Green channel color information, 0-32.
B_4–B_0: Blue channel color information, 0-32.

Left and Right Audio Samples: The Gameduino has the ability to play back stereo audio wave data using the SAMPLE_L and SAMPLE_R registers. Each register accepts a signed 16-bit integer reflecting the sample. The Gameduino updates audio samples every 64 cycles of it's 50MHz clock, corresponding to an update frequency of 781.25kHz, so values loaded into these registers are reflected on the audio output channels within 1.28μs.

Table 18: SAMPLE_L and SAMPLE_R Registers

Symbol		Address		Length		Byte Offset and Register Contents		Default
		(hex)	(dec)	(hex)	(dec)	+0	+1	
SAMPLE_L	RW	0x2810	10256	0x2	2	$L_7L_6L_5L_4L_3L_2L_1L_0$	$L_{15}L_{14}L_{13}L_{12}L_{11}L_{10}L_9L_8$	0x0000

SCROLL_R	RW	0x2812	10258	0x2	2	$R_7R_6R_5R_4R_3R_2R_1R_0$	$R_{15}R_{14}R_{13}R_{12}R_{11}R_{10}R_9R_8$	0x0000

L_0–L_{15}: Left audio channel sample value, 16-bit signed integer (L_{15} is the sign bit).
R_0–R_{15}: Right audio channel sample value, 16-bit signed integer (R_{15} is the sign bit).

Screenshot Line Select: The SCREENSHOT_Y register is used to capture a line of raster data from the SCREENSHOT buffer at 0x2C00 (11264 decimal). To take a screenshot, load a value between 0x8000 and 0x812B (decimal 32768 and 33067), corresponding to the flag bit F=1 and the S_0-S_8 equal to the line number (0 to 299) to be captured. When the flag bit of the register reads 1, the selected line of data is available in memory locations 0x2C00-0x2F1F (11264-12063 decimal). Writing a 0 disables the screenshot feature, while the low bits S_0-S_8 will always read the current Gameduino scan line.

Table 19: SCREENSHOT_Y Register

Symbol		Address		Length		Byte Offset and Register Contents		Default
		(hex)	(dec)	(hex)	(dec)	+0	+1	
SCREENSHOT_Y	RW	0x2406	10246	0x2	2	$S_7S_6S_5S_4S_3S_2S_1S_0$	F X X X X X X S_8	0x0000

X: Unused bit position, ignored; should be 0 for compatibility reasons.
S_0–S_8: Raster line to capture, 0-299.
F: Screenshot flag, 0 or 1.

Unused Registers and **Unused Address Space:** Several portions of the Gameduino memory map are unused. These addresses have no function, and are not mapped to storage or control. When written, these locations have no effect and the data written is discarded. When read, the locations have no specified value and should read 0, but some addresses may read other, unpredictable values instead. In particular, unused registers and unused address space is not connected to RAM or other scratchpad storage.

Video Display

The Gameduino presents a 400×300 pixel screen display at 72Hz (72 frames per second). The pixels are square on standard display devices. The Gameduino generates its 400×300 video display by combining image data from a number of image sources: background color, a character graphics playfield, and the sprites. Each image source has a different priority – that is, the video sources are arranged in order from back to front, and sources that closer to the front can cover up information from one that is farther to the back. The character graphics and sprites may include transparent pixels that allow lower-priority pixels to show through. The background doesn't allow

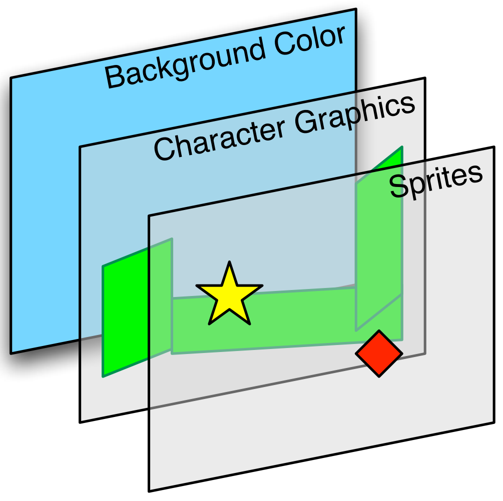

Figure 1: Video Display Layers

transparent pixels, since there is no lower-priority image to show through.

The sources for the Gameduino's video display, in order from back to front, are:

- Background color,
- Character graphics, and
- Sprites in numeric order.

For example, image data on the playfield covers up the background color, and sprite 0 image data would in turn cover up the playfield. Sprite 255 would cover up pixels from any lower-numbered sprite (including sprite 0) as well as playfield and background pixels.

Scrolling

Each image source is a logical 512×512 pixel image. However, the Gameduino's VGA output is limited to a 400x300 pixel window. The SCROLL_X and SCROLL_Y registers are used to define where the 400×300 display window starts within the background image. This is position of the top-left corner of the visible window, as measured in pixels from the top-left corner of the logical image.

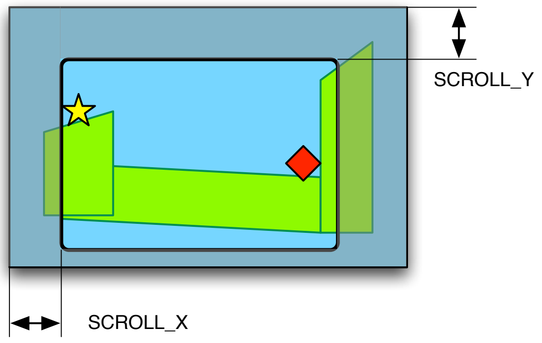

Figure 2: Scrolling

Wraparound

SCROLL_X and SCROLL_Y can store any scroll value from 0 to 511. When SCROLL_X and SCROLL_Y are both 0 (the default), the display window starts at the top corner of the logical image; the right-most 212 pixels and the bottom 112 pixels are off screen and invisible. Scroll values of 112 and 212 (for X and Y respectively) would align the visible screen with the bottom-right corner of the logical image. Larger scroll values would cause the visible window to wrap around the logical image: once pixel 511 of the logical image is output, the subsequent pixels begin with pixel 0 of the same line. Similarly, once row 511 has been output to the VGA screen, the next row to be output will be row 0 of the logical image.

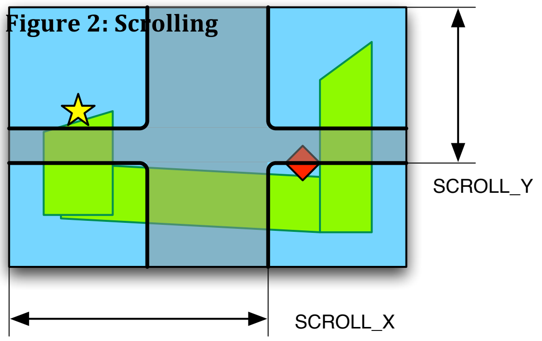

Figure 3: Scroll Wraparound

Table 20: SCROLL Registers

Symbol		Address		Length		Byte Offset and Register Contents		Default
		(hex)	(dec)	(hex)	(dec)	+0	+1	
SCROLL_X	RW	0x2804	10244	0x2	2	$S_7S_6S_5S_4S_3S_2S_1S_0$	X X X X X X X S_8	0x0000
SCROLL_Y	RW	0x2406	10246	0x2	2	$S_7S_6S_5S_4S_3S_2S_1S_0$	X X X X X X X S_8	0x0000

X: Unused bit position, ignored; should be 0 for compatibility reasons.

S_0–S_8: Scroll offset in pixels, 0-511.

Programmers can take advantage of this wrap-around feature to implement continuously scrolling images. Characters and sprites that are currently off-screen (the greyed-out areas of the figures above) can be updated while the screen scrolls, giving the illusion of an infinite playfield.

<<<to-do: insert example code here>>>

Split Screen Scrolling

The scroll registers affect the entire frame, so changing the scroll values in the middle of generating a frame on the display will cause parts of the display to scroll independently of the rest of the display. The Gameduino's J1 coprocessor can reload the scroll registers in the middle of video generation to create a split-screen or multi-pane scrolling effect.

<<<to-do: insert example code here>>>

Background Color

The background color is defined by the contents of the 16-bit BG_COLOR register. This color appears behind all of the other video sources. On power-up, the background color is the only image displayed, and defaults to 0x0000 (black).

The BG_COLOR value is a standard Gameduino 16-bit color value (see the section on Color), except that Bit 15, the alpha (A) channel or transparency bit, is ignored if set. The background can never be transparent, since there are no lower-priority pixels to show through. The background is normally a solid color, but see the section on raster chasing for examples involving multicolored backgrounds.

Table 21: BG_COLOR Register

Symbol		Address		Length		Byte Offset and Register Contents		Default
		(hex)	(dec)	(hex)	(dec)	+0	+1	
BG_COLOR	RW	0x280E	10254	0x2	2	$G_2G_1G_0B_4B_3B_2B_1B_0$	X $R_4R_3R_2R_1R_0G_4G_3$	0x0000

X: Alpha channel color information, ignored.
R_4–R_0: Red channel color information, 0-32.
B_4–B_0: Green channel color information, 0-32.
B_4–B_0: Blue channel color information, 0-32.

Character Graphics

Together, the memory areas RAM_PIC, RAM_CHR, and RAM_PAL define the character graphics picture that is displayed behind the game sprites and in front of the background color. This picture consists of a logical 512×512 pixel image, but a full bitmap of 2-byte colors would occupy a half-megabyte – obviously an issue for a device with only 32kbytes of address space. Instead, RAM_PIC contains a 64×64 grid of one-byte characters, where each character in the grid represents one of 256 8×8 pixel bitmaps stored in RAM_CHR. Each character in the grid also has its own 4-color palette in RAM_PAL.

RAM_PIC: Playfield Picture

RAM_PIC contains a 64×64 grid of characters that defines the character graphics picture. Each byte in RAM_PIC represents a character bitmap to be looked up in RAM_CHR. Character bitmaps are 8×8 pixels and can select one of four colors; color information for each pixel is drawn from RAM_PAL. Character graphics pixels may be transparent, allowing the background color to show through. The result is a 512×512 pixel character graphics picture that forms the playfield for game sketches. The organization of the playfield is shown in Table 25: RAM_PIC Memory Map.

A 400×300 pixel (50×37½ character) window of this picture is visible on screen. The window may be scrolled around the picture on a single-pixel basis, so that parts of as many as 51×39 characters may be fully or partially visible. The Gameduino's memory is dual-ported, so characters that are not currently on screen can be updated by the Arduino SPI interface or by the J1 coprocessor without causing visible artifacts.

RAM_CHR: Playfield Character Set

Each character code in RAM_PIC is looked up in RAM_CHR to determine the corresponding 8×8 pixel character bitmap that should be displayed at the character position. There are 256 bitmaps, one for each possible character code stored in RAM_PIC. Table 26: RAM_CHR Memory Map shows the organization of the character set memory. Character codes are listed in hexadecimal and decimal, along with the ASCII glyph for characters 33 through 126 (0x31 through 0x7F). Glyphs that are not printable ASCII characters are shaded grey.

The RAM_CHR bitmaps determine the graphic that is displayed at each character location. Character bitmaps contain 4 colors and are therefore 2 bits deep; each character bitmap occupies 16 bytes. The format of each character bitmap is shown in Table 22: RAM_CHR Bitmap Organization. Pixels in the character bitmap have coordinates starting at (0,0) for the upper left-hand corner of the bitmap, to (7,7) at the lower right. Each pixel is two bits, the value 0-3 selects one of the four colors stored in the corresponding character palette in RAM_PAL.

Table 22: RAM_CHR Bitmap Organization

Row Offset		Byte Offset and Bitmap Contents	
(hex)	(dec)	+0	+1
+0x00	+0	$P00_1P00_0$ $P10_1P10_0$ $P20_1P20_0$ $P30_1P30_0$	$P40_1P40_0$ $P50_1P50_0$ $P60_1P60_0$ $P70_1P70_0$
+0x02	+2	$P01_1P01_0$ $P11_1P11_0$ $P21_1P21_0$ $P31_1P31_0$	$P41_1P41_0$ $P51_1P51_0$ $P61_1P61_0$ $P71_1P71_0$
+0x04	+4	$P02_1P02_0$ $P12_1P12_0$ $P22_1P22_0$ $P32_1P32_0$	$P42_1P42_0$ $P52_1P52_0$ $P62_1P62_0$ $P72_1P72_0$
+0x06	+6	$P03_1P03_0$ $P13_1P13_0$ $P23_1P23_0$ $P33_1P33_0$	$P43_1P43_0$ $P53_1P53_0$ $P63_1P63_0$ $P73_1P73_0$
+0x08	+8	$P04_1P04_0$ $P14_1P14_0$ $P24_1P24_0$ $P34_1P34_0$	$P44_1P44_0$ $P54_1P54_0$ $P64_1P64_0$ $P74_1P74_0$
+0x0A	+10	$P05_1P05_0$ $P15_1P15_0$ $P25_1P25_0$ $P35_1P35_0$	$P45_1P45_0$ $P55_1P55_0$ $P65_1P65_0$ $P75_1P75_0$
+0x0C	+12	$P06_1P06_0$ $P16_1P16_0$ $P26_1P26_0$ $P36_1P36_0$	$P46_1P46_0$ $P56_1P56_0$ $P66_1P66_0$ $P76_1P76_0$
+0x0D	+14	$P07_1P07_0$ $P17_1P17_0$ $P27_1P27_0$ $P37_1P37_0$	$P47_1P47_0$ $P57_1P57_0$ $P67_1P67_0$ $P77_1P77_0$

Pxy_1Pxy_0: Pixel value at bitmap location (x,y), 0-3.

Character Sets: Table 26: RAM_CHR Memory Map shows ASCII glyphs for convenience. In ASCII, the control characters at 0 through 31 (0x00 through 0x1F) and 127 (0x7F) aren't printable ASCII characters, while character codes 128 through 255 (0x80 through 0xFF) are undefined. This makes these characters good candidates for use as playfield graphics instead of alphabetic and numeric symbols.

ASCII is a relatively old standard; several more modern standards exist, and all are supersets of the ASCII standard. Table 23: Character Sets shows the relationship between six popular code pages. All share the same definition for characters (code points) between 0 and 127 (0x00 and 0x7F) with ASCII.

Table 23: Character Sets

Character Set	Character Codes				
	`0x00-0x1F`	`0x20-0x7E`	`0x7F`	`0x80-0x9F`	`0xA0-0xFF`
ASCII	Control	Printable	Control	Undefined	Undefined
Windows-1252	Control	Printable	Control	Printable	Printable
Mac OS Roman	Control	Printable	Control	Printable	Printable
ISO-8859-1	Control	Printable	Control	Control	Printable
ISO-8859-15	Control	Printable	Control	Control	Printable
UTF-8	Control	Printable	Control	Multi-byte	Multi-byte

Most Windows systems configured for a Western alphabet uses the Windows-1252 character set. Apple products use Mac OS Roman for the same purpose. The two character sets of course differ on the characters above code 127 (0x7F); after all, when have Redmond and Cupertino been able to completely agree on anything?

The relevant standards are ISO-8859-1 and its updated version, ISO-8859-15 (which includes the Euro character and a few other modifications). UTF-8 is an 8-bit encoding for Unicode that maximizes ASCII compatibility: codes 0 through 127 (0x00 through 0x7F) match ASCII, while the higher codes are only encountered as part of multi-byte sequences that encode the rest of the Unicode character set.

RAM_PAL: Playfield Character Palettes

Each character on the playfield has its own 4-color palette in RAM_PAL. This means that each character has its own color palette, and also that characters that have identical bitmaps but require different colors must be stored as separate characters. Color data is stored in standard ARGB1555 format, resulting in 256 character palettes, each palette occupying 8 bytes. The overall structure of RAM_PAL is shown in Table 27: RAM_PAL Memory Map.

The individual palettes are composed of 4 color entries, numbered 0 through 4 as shown in Table 24: RAM_PAL Palette Format. Character bitmap pixels map directly to the palette entries: bitmap pixel values of 0 are displayed in the color entry 0, pixel value 1 is displayed using color entry 1, and so on. Transparency can be set for any color or colors in the palette. It is customary to code character sets so that color entry 0 (pixel value 0) corresponds to the background or transparent color.

Table 24: RAM_PAL Palette Format

Color Entry	Entry Offset	Byte Offset	
		+0	+1
0	+0	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
1	+2	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
2	+4	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
3	+6	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$

A: Alpha channel color information, 0=opaque, 1=transparent.
R_0–R_4: Red channel color information, 0-32; ignored when A=1.
B_0–B_4: Green channel color information, 0-32; ignored when A=1.
B_0–B_4: Blue channel color information, 0-32; ignored when A=1.

Table 25: RAM_PIC Memory Map

Row Number		Address		Contents	Default
(hex)	(dec)	(hex)	(dec)		
0x00	0	0x0000	0	64 character codes	0x00
0x01	1	0x0040	64	64 character codes	0x00
0x02	2	0x0080	128	64 character codes	0x00
0x03	3	0x00C0	192	64 character codes	0x00
0x04	4	0x0100	256	64 character codes	0x00
0x05	5	0x0140	320	64 character codes	0x00
0x06	6	0x0180	384	64 character codes	0x00
0x07	7	0x01C0	448	64 character codes	0x00
0x08	8	0x0200	512	64 character codes	0x00
0x09	9	0x0240	576	64 character codes	0x00
0x0A	10	0x0280	640	64 character codes	0x00
0x0B	11	0x02C0	704	64 character codes	0x00
0x0C	12	0x0300	768	64 character codes	0x00
0x0D	13	0x0340	832	64 character codes	0x00
0x0E	14	0x0380	896	64 character codes	0x00
0x0F	15	0x03C0	960	64 character codes	0x00
0x10	16	0x0400	1024	64 character codes	0x00
0x11	17	0x0440	1088	64 character codes	0x00
0x12	18	0x0480	1152	64 character codes	0x00
0x13	19	0x04C0	1216	64 character codes	0x00
0x14	20	0x0500	1280	64 character codes	0x00
0x15	21	0x0540	1344	64 character codes	0x00
0x16	22	0x0580	1408	64 character codes	0x00
0x17	23	0x05C0	1472	64 character codes	0x00
0x18	24	0x0600	1536	64 character codes	0x00
0x19	25	0x0640	1600	64 character codes	0x00
0x1A	26	0x0680	1664	64 character codes	0x00
0x1B	27	0x06C0	1728	64 character codes	0x00
0x1C	28	0x0700	1792	64 character codes	0x00
0x1D	29	0x0740	1856	64 character codes	0x00
0x1E	30	0x0780	1920	64 character codes	0x00
0x1F	31	0x07C0	1984	64 character codes	0x00
0x20	32	0x0800	2048	64 character codes	0x00
0x21	33	0x0840	2112	64 character codes	0x00
0x22	34	0x0880	2176	64 character codes	0x00
0x23	35	0x08C0	2240	64 character codes	0x00
0x24	36	0x0900	2304	64 character codes	0x00
0x25	37	0x0940	2368	64 character codes	0x00
0x26	38	0x0980	2432	64 character codes	0x00
0x27	39	0x09C0	2496	64 character codes	0x00
0x28	40	0x0A00	2560	64 character codes	0x00
0x29	41	0x0A40	2624	64 character codes	0x00
0x2A	42	0x0A80	2688	64 character codes	0x00
0x2B	43	0x0AC0	2752	64 character codes	0x00
0x2C	44	0x0B00	2816	64 character codes	0x00
0x2D	45	0x0B40	2880	64 character codes	0x00
0x2E	46	0x0B80	2944	64 character codes	0x00
0x2F	47	0x0BC0	3008	64 character codes	0x00
0x30	48	0x0C00	3072	64 character codes	0x00
0x31	49	0x0C40	3136	64 character codes	0x00
0x32	50	0x0C80	3200	64 character codes	0x00
0x33	51	0x0CC0	3264	64 character codes	0x00
0x34	52	0x0D00	3328	64 character codes	0x00
0x35	53	0x0D40	3392	64 character codes	0x00
0x36	54	0x0D80	3456	64 character codes	0x00
0x37	55	0x0DC0	3520	64 character codes	0x00
0x38	56	0x0E00	3584	64 character codes	0x00
0x39	57	0x0E40	3648	64 character codes	0x00
0x3A	58	0x0E80	3712	64 character codes	0x00
0x3B	59	0x0EC0	3776	64 character codes	0x00
0x3C	60	0x0F00	3840	64 character codes	0x00
0x3D	61	0x0F40	3904	64 character codes	0x00
0x3E	62	0x0F80	3968	64 character codes	0x00
0x3F	63	0x0FC0	4032	64 character codes	0x00

Table 26: RAM_CHR Memory Map

Character			Address		Contents
(hex)	(dec)	(g)	(hex)	(dec)	
0x00	0		0x1000	4096	8×8×2 bitmap, 16 bytes
0x01	1		0x1010	4112	8×8×2 bitmap, 16 bytes
0x02	2		0x1020	4128	8×8×2 bitmap, 16 bytes
0x03	3		0x1030	4144	8×8×2 bitmap, 16 bytes
0x04	4		0x1040	4160	8×8×2 bitmap, 16 bytes
0x05	5		0x1050	4176	8×8×2 bitmap, 16 bytes
0x06	6		0x1060	4192	8×8×2 bitmap, 16 bytes
0x07	7		0x1070	4208	8×8×2 bitmap, 16 bytes
0x08	8		0x1080	4224	8×8×2 bitmap, 16 bytes
0x09	9		0x1090	4240	8×8×2 bitmap, 16 bytes
0x0A	10		0x10A0	4256	8×8×2 bitmap, 16 bytes
0x0B	11		0x10B0	4272	8×8×2 bitmap, 16 bytes
0x0C	12		0x10C0	4288	8×8×2 bitmap, 16 bytes
0x0D	13		0x10D0	4304	8×8×2 bitmap, 16 bytes
0x0E	14		0x10E0	4320	8×8×2 bitmap, 16 bytes
0x0F	15		0x10F0	4336	8×8×2 bitmap, 16 bytes
0x10	16		0x1100	4352	8×8×2 bitmap, 16 bytes
0x11	17		0x1110	4368	8×8×2 bitmap, 16 bytes
0x12	18		0x1120	4384	8×8×2 bitmap, 16 bytes
0x13	19		0x1130	4400	8×8×2 bitmap, 16 bytes
0x14	20		0x1140	4416	8×8×2 bitmap, 16 bytes
0x15	21		0x1150	4432	8×8×2 bitmap, 16 bytes
0x16	22		0x1160	4448	8×8×2 bitmap, 16 bytes
0x17	23		0x1170	4464	8×8×2 bitmap, 16 bytes
0x18	24		0x1180	4480	8×8×2 bitmap, 16 bytes
0x19	25		0x1190	4496	8×8×2 bitmap, 16 bytes
0x1A	26		0x11A0	4512	8×8×2 bitmap, 16 bytes
0x1B	27		0x11B0	4528	8×8×2 bitmap, 16 bytes
0x1C	28		0x11C0	4544	8×8×2 bitmap, 16 bytes
0x1D	29		0x11D0	4560	8×8×2 bitmap, 16 bytes
0x1E	30		0x11E0	4576	8×8×2 bitmap, 16 bytes
0x1F	31		0x11F0	4592	8×8×2 bitmap, 16 bytes
0x20	32		0x1200	4608	8×8×2 bitmap, 16 bytes
0x21	33	!	0x1210	4624	8×8×2 bitmap, 16 bytes
0x22	34	"	0x1220	4640	8×8×2 bitmap, 16 bytes
0x23	35	#	0x1230	4656	8×8×2 bitmap, 16 bytes
0x24	36	$	0x1240	4672	8×8×2 bitmap, 16 bytes
0x25	37	%	0x1250	4688	8×8×2 bitmap, 16 bytes
0x26	38	&	0x1260	4704	8×8×2 bitmap, 16 bytes
0x27	39	'	0x1270	4720	8×8×2 bitmap, 16 bytes
0x28	40	(	0x1280	4736	8×8×2 bitmap, 16 bytes
0x29	41	)	0x1290	4752	8×8×2 bitmap, 16 bytes
0x2A	42	*	0x12A0	4768	8×8×2 bitmap, 16 bytes
0x2B	43	+	0x12B0	4784	8×8×2 bitmap, 16 bytes
0x2C	44	,	0x12C0	4800	8×8×2 bitmap, 16 bytes
0x2D	45	-	0x12D0	4816	8×8×2 bitmap, 16 bytes
0x2E	46	.	0x12E0	4832	8×8×2 bitmap, 16 bytes
0x2F	47	/	0x12F0	4848	8×8×2 bitmap, 16 bytes
0x30	48	0	0x1300	4864	8×8×2 bitmap, 16 bytes
0x31	49	1	0x1310	4880	8×8×2 bitmap, 16 bytes
0x32	50	2	0x1320	4896	8×8×2 bitmap, 16 bytes
0x33	51	3	0x1330	4912	8×8×2 bitmap, 16 bytes
0x34	52	4	0x1340	4928	8×8×2 bitmap, 16 bytes
0x35	53	5	0x1350	4944	8×8×2 bitmap, 16 bytes
0x36	54	6	0x1360	4960	8×8×2 bitmap, 16 bytes
0x37	55	7	0x1370	4976	8×8×2 bitmap, 16 bytes
0x38	56	8	0x1380	4992	8×8×2 bitmap, 16 bytes
0x39	57	9	0x1390	5008	8×8×2 bitmap, 16 bytes
0x3A	58	:	0x13A0	5024	8×8×2 bitmap, 16 bytes
0x3B	59	;	0x13B0	5040	8×8×2 bitmap, 16 bytes
0x3C	60	<	0x13C0	5056	8×8×2 bitmap, 16 bytes
0x3D	61	=	0x13D0	5072	8×8×2 bitmap, 16 bytes
0x3E	62	>	0x13E0	5088	8×8×2 bitmap, 16 bytes
0x3F	63	?	0x13F0	5104	8×8×2 bitmap, 16 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0x40	64	@		0x1400	5120		8×8×2 bitmap, 16 bytes
0x41	65	A		0x1410	5136		8×8×2 bitmap, 16 bytes
0x42	66	B		0x1420	5152		8×8×2 bitmap, 16 bytes
0x43	67	C		0x1430	5168		8×8×2 bitmap, 16 bytes
0x44	68	D		0x1440	5184		8×8×2 bitmap, 16 bytes
0x45	69	E		0x1450	5200		8×8×2 bitmap, 16 bytes
0x46	70	F		0x1460	5216		8×8×2 bitmap, 16 bytes
0x47	71	G		0x1470	5232		8×8×2 bitmap, 16 bytes
0x48	72	H		0x1480	5248		8×8×2 bitmap, 16 bytes
0x49	73	I		0x1490	5264		8×8×2 bitmap, 16 bytes
0x4A	74	J		0x14A0	5280		8×8×2 bitmap, 16 bytes
0x4B	75	K		0x14B0	5296		8×8×2 bitmap, 16 bytes
0x4C	76	L		0x14C0	5312		8×8×2 bitmap, 16 bytes
0x4D	77	M		0x14D0	5328		8×8×2 bitmap, 16 bytes
0x4E	78	N		0x14E0	5344		8×8×2 bitmap, 16 bytes
0x4F	79	O		0x14F0	5360		8×8×2 bitmap, 16 bytes
0x50	80	P		0x1500	5376		8×8×2 bitmap, 16 bytes
0x51	81	Q		0x1510	5392		8×8×2 bitmap, 16 bytes
0x52	82	R		0x1520	5408		8×8×2 bitmap, 16 bytes
0x53	83	S		0x1530	5424		8×8×2 bitmap, 16 bytes
0x54	84	T		0x1540	5440		8×8×2 bitmap, 16 bytes
0x55	85	U		0x1550	5456		8×8×2 bitmap, 16 bytes
0x56	86	V		0x1560	5472		8×8×2 bitmap, 16 bytes
0x57	87	W		0x1570	5488		8×8×2 bitmap, 16 bytes
0x58	88	X		0x1580	5504		8×8×2 bitmap, 16 bytes
0x59	89	Y		0x1590	5520		8×8×2 bitmap, 16 bytes
0x5A	90	Z		0x15A0	5536		8×8×2 bitmap, 16 bytes
0x5B	91	[		0x15B0	5552		8×8×2 bitmap, 16 bytes
0x5C	92	\		0x15C0	5568		8×8×2 bitmap, 16 bytes
0x5D	93	]		0x15D0	5584		8×8×2 bitmap, 16 bytes
0x5E	94	^		0x15E0	5600		8×8×2 bitmap, 16 bytes
0x5F	95	_		0x15F0	5616		8×8×2 bitmap, 16 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0x60	96	`		0x1600	5632		8×8×2 bitmap, 16 bytes
0x61	97	a		0x1610	5648		8×8×2 bitmap, 16 bytes
0x62	98	b		0x1620	5664		8×8×2 bitmap, 16 bytes
0x63	99	c		0x1630	5680		8×8×2 bitmap, 16 bytes
0x64	100	d		0x1640	5696		8×8×2 bitmap, 16 bytes
0x65	101	e		0x1650	5712		8×8×2 bitmap, 16 bytes
0x66	102	f		0x1660	5728		8×8×2 bitmap, 16 bytes
0x67	103	g		0x1670	5744		8×8×2 bitmap, 16 bytes
0x68	104	h		0x1680	5760		8×8×2 bitmap, 16 bytes
0x69	105	i		0x1690	5776		8×8×2 bitmap, 16 bytes
0x6A	106	j		0x16A0	5792		8×8×2 bitmap, 16 bytes
0x6B	107	k		0x16B0	5808		8×8×2 bitmap, 16 bytes
0x6C	108	l		0x16C0	5824		8×8×2 bitmap, 16 bytes
0x6D	109	m		0x16D0	5840		8×8×2 bitmap, 16 bytes
0x6E	110	n		0x16E0	5856		8×8×2 bitmap, 16 bytes
0x6F	111	o		0x16F0	5872		8×8×2 bitmap, 16 bytes
0x70	112	p		0x1700	5888		8×8×2 bitmap, 16 bytes
0x71	113	q		0x1710	5904		8×8×2 bitmap, 16 bytes
0x72	114	r		0x1720	5920		8×8×2 bitmap, 16 bytes
0x73	115	s		0x1730	5936		8×8×2 bitmap, 16 bytes
0x74	116	t		0x1740	5952		8×8×2 bitmap, 16 bytes
0x75	117	u		0x1750	5968		8×8×2 bitmap, 16 bytes
0x76	118	v		0x1760	5984		8×8×2 bitmap, 16 bytes
0x77	119	w		0x1770	6000		8×8×2 bitmap, 16 bytes
0x78	120	x		0x1780	6016		8×8×2 bitmap, 16 bytes
0x79	121	y		0x1790	6032		8×8×2 bitmap, 16 bytes
0x7A	122	z		0x17A0	6048		8×8×2 bitmap, 16 bytes
0x7B	123	{		0x17B0	6064		8×8×2 bitmap, 16 bytes
0x7C	124	|		0x17C0	6080		8×8×2 bitmap, 16 bytes
0x7D	125	}		0x17D0	6096		8×8×2 bitmap, 16 bytes
0x7E	126	~		0x17E0	6112		8×8×2 bitmap, 16 bytes
0x7F	127			0x17F0	6128		8×8×2 bitmap, 16 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0x80	128			0x1800	6144		8×8×2 bitmap, 16 bytes
0x81	129			0x1810	6160		8×8×2 bitmap, 16 bytes
0x82	130			0x1820	6176		8×8×2 bitmap, 16 bytes
0x83	131			0x1830	6192		8×8×2 bitmap, 16 bytes
0x84	132			0x1840	6208		8×8×2 bitmap, 16 bytes
0x85	133			0x1850	6224		8×8×2 bitmap, 16 bytes
0x86	134			0x1860	6240		8×8×2 bitmap, 16 bytes
0x87	135			0x1870	6256		8×8×2 bitmap, 16 bytes
0x88	136			0x1880	6272		8×8×2 bitmap, 16 bytes
0x89	137			0x1890	6288		8×8×2 bitmap, 16 bytes
0x8A	138			0x18A0	6304		8×8×2 bitmap, 16 bytes
0x8B	139			0x18B0	6320		8×8×2 bitmap, 16 bytes
0x8C	140			0x18C0	6336		8×8×2 bitmap, 16 bytes
0x8D	141			0x18D0	6352		8×8×2 bitmap, 16 bytes
0x8E	142			0x18E0	6368		8×8×2 bitmap, 16 bytes
0x8F	143			0x18F0	6384		8×8×2 bitmap, 16 bytes
0x90	144			0x1900	6400		8×8×2 bitmap, 16 bytes
0x91	145			0x1910	6416		8×8×2 bitmap, 16 bytes
0x92	146			0x1920	6432		8×8×2 bitmap, 16 bytes
0x93	147			0x1930	6448		8×8×2 bitmap, 16 bytes
0x94	148			0x1940	6464		8×8×2 bitmap, 16 bytes
0x95	149			0x1950	6480		8×8×2 bitmap, 16 bytes
0x96	150			0x1960	6496		8×8×2 bitmap, 16 bytes
0x97	151			0x1970	6512		8×8×2 bitmap, 16 bytes
0x98	152			0x1980	6528		8×8×2 bitmap, 16 bytes
0x99	153			0x1990	6544		8×8×2 bitmap, 16 bytes
0x9A	154			0x19A0	6560		8×8×2 bitmap, 16 bytes
0x9B	155			0x19B0	6576		8×8×2 bitmap, 16 bytes
0x9C	156			0x19C0	6592		8×8×2 bitmap, 16 bytes
0x9D	157			0x19D0	6608		8×8×2 bitmap, 16 bytes
0x9E	158			0x19E0	6624		8×8×2 bitmap, 16 bytes
0x9F	159			0x19F0	6640		8×8×2 bitmap, 16 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0xA0	160			0x1A00	6656		8×8×2 bitmap, 16 bytes
0xA1	161			0x1A10	6672		8×8×2 bitmap, 16 bytes
0xA2	162			0x1A20	6688		8×8×2 bitmap, 16 bytes
0xA3	163			0x1A30	6704		8×8×2 bitmap, 16 bytes
0xA4	164			0x1A40	6720		8×8×2 bitmap, 16 bytes
0xA5	165			0x1A50	6736		8×8×2 bitmap, 16 bytes
0xA6	166			0x1A60	6752		8×8×2 bitmap, 16 bytes
0xA7	167			0x1A70	6768		8×8×2 bitmap, 16 bytes
0xA8	168			0x1A80	6784		8×8×2 bitmap, 16 bytes
0xA9	169			0x1A90	6800		8×8×2 bitmap, 16 bytes
0xAA	170			0x1AA0	6816		8×8×2 bitmap, 16 bytes
0xAB	171			0x1AB0	6832		8×8×2 bitmap, 16 bytes
0xAC	172			0x1AC0	6848		8×8×2 bitmap, 16 bytes
0xAD	173			0x1AD0	6864		8×8×2 bitmap, 16 bytes
0xAE	174			0x1AE0	6880		8×8×2 bitmap, 16 bytes
0xAF	175			0x1AF0	6896		8×8×2 bitmap, 16 bytes
0xB0	176			0x1B00	6912		8×8×2 bitmap, 16 bytes
0xB1	177			0x1B10	6928		8×8×2 bitmap, 16 bytes
0xB2	178			0x1B20	6944		8×8×2 bitmap, 16 bytes
0xB3	179			0x1B30	6960		8×8×2 bitmap, 16 bytes
0xB4	180			0x1B40	6976		8×8×2 bitmap, 16 bytes
0xB5	181			0x1B50	6992		8×8×2 bitmap, 16 bytes
0xB6	182			0x1B60	7008		8×8×2 bitmap, 16 bytes
0xB7	183			0x1B70	7024		8×8×2 bitmap, 16 bytes
0xB8	184			0x1B80	7040		8×8×2 bitmap, 16 bytes
0xB9	185			0x1B90	7056		8×8×2 bitmap, 16 bytes
0xBA	186			0x1BA0	7072		8×8×2 bitmap, 16 bytes
0xBB	187			0x1BB0	7088		8×8×2 bitmap, 16 bytes
0xBC	188			0x1BC0	7104		8×8×2 bitmap, 16 bytes
0xBD	189			0x1BD0	7120		8×8×2 bitmap, 16 bytes
0xBE	190			0x1BE0	7136		8×8×2 bitmap, 16 bytes
0xBF	191			0x1BF0	7152		8×8×2 bitmap, 16 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0xC0	192			0x1C00	7168		8×8×2 bitmap, 16 bytes
0xC1	193			0x1C10	7184		8×8×2 bitmap, 16 bytes
0xC2	194			0x1C20	7200		8×8×2 bitmap, 16 bytes
0xC3	195			0x1C30	7216		8×8×2 bitmap, 16 bytes
0xC4	196			0x1C40	7232		8×8×2 bitmap, 16 bytes
0xC5	197			0x1C50	7248		8×8×2 bitmap, 16 bytes
0xC6	198			0x1C60	7264		8×8×2 bitmap, 16 bytes
0xC7	199			0x1C70	7280		8×8×2 bitmap, 16 bytes
0xC8	200			0x1C80	7296		8×8×2 bitmap, 16 bytes
0xC9	201			0x1C90	7312		8×8×2 bitmap, 16 bytes
0xCA	202			0x1CA0	7328		8×8×2 bitmap, 16 bytes
0xCB	203			0x1CB0	7344		8×8×2 bitmap, 16 bytes
0xCC	204			0x1CC0	7360		8×8×2 bitmap, 16 bytes
0xCD	205			0x1CD0	7376		8×8×2 bitmap, 16 bytes
0xCE	206			0x1CE0	7392		8×8×2 bitmap, 16 bytes
0xCF	207			0x1CF0	7408		8×8×2 bitmap, 16 bytes
0xD0	208			0x1D00	7424		8×8×2 bitmap, 16 bytes
0xD1	209			0x1D10	7440		8×8×2 bitmap, 16 bytes
0xD2	210			0x1D20	7456		8×8×2 bitmap, 16 bytes
0xD3	211			0x1D30	7472		8×8×2 bitmap, 16 bytes
0xD4	212			0x1D40	7488		8×8×2 bitmap, 16 bytes
0xD5	213			0x1D50	7504		8×8×2 bitmap, 16 bytes
0xD6	214			0x1D60	7520		8×8×2 bitmap, 16 bytes
0xD7	215			0x1D70	7536		8×8×2 bitmap, 16 bytes
0xD8	216			0x1D80	7552		8×8×2 bitmap, 16 bytes
0xD9	217			0x1D90	7568		8×8×2 bitmap, 16 bytes
0xDA	218			0x1DA0	7584		8×8×2 bitmap, 16 bytes
0xDB	219			0x1DB0	7600		8×8×2 bitmap, 16 bytes
0xDC	220			0x1DC0	7616		8×8×2 bitmap, 16 bytes
0xDD	221			0x1DD0	7632		8×8×2 bitmap, 16 bytes
0xDE	222			0x1DE0	7648		8×8×2 bitmap, 16 bytes
0xDF	223			0x1DF0	7664		8×8×2 bitmap, 16 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0xE0	224			0x1E00	7680		8×8×2 bitmap, 16 bytes
0xE1	225			0x1E10	7696		8×8×2 bitmap, 16 bytes
0xE2	226			0x1E20	7712		8×8×2 bitmap, 16 bytes
0xE3	227			0x1E30	7728		8×8×2 bitmap, 16 bytes
0xE4	228			0x1E40	7744		8×8×2 bitmap, 16 bytes
0xE5	229			0x1E50	7760		8×8×2 bitmap, 16 bytes
0xE6	230			0x1E60	7776		8×8×2 bitmap, 16 bytes
0xE7	231			0x1E70	7792		8×8×2 bitmap, 16 bytes
0xE8	232			0x1E80	7808		8×8×2 bitmap, 16 bytes
0xE9	233			0x1E90	7824		8×8×2 bitmap, 16 bytes
0xEA	234			0x1EA0	7840		8×8×2 bitmap, 16 bytes
0xEB	235			0x1EB0	7856		8×8×2 bitmap, 16 bytes
0xEC	236			0x1EC0	7872		8×8×2 bitmap, 16 bytes
0xED	237			0x1ED0	7888		8×8×2 bitmap, 16 bytes
0xEE	238			0x1EE0	7904		8×8×2 bitmap, 16 bytes
0xEF	239			0x1EF0	7920		8×8×2 bitmap, 16 bytes
0xF0	240			0x1F00	7936		8×8×2 bitmap, 16 bytes
0xF1	241			0x1F10	7952		8×8×2 bitmap, 16 bytes
0xF2	242			0x1F20	7968		8×8×2 bitmap, 16 bytes
0xF3	243			0x1F30	7984		8×8×2 bitmap, 16 bytes
0xF4	244			0x1F40	8000		8×8×2 bitmap, 16 bytes
0xF5	245			0x1F50	8016		8×8×2 bitmap, 16 bytes
0xF6	246			0x1F60	8032		8×8×2 bitmap, 16 bytes
0xF7	247			0x1F70	8048		8×8×2 bitmap, 16 bytes
0xF8	248			0x1F80	8064		8×8×2 bitmap, 16 bytes
0xF9	249			0x1F90	8080		8×8×2 bitmap, 16 bytes
0xFA	250			0x1FA0	8096		8×8×2 bitmap, 16 bytes
0xFB	251			0x1FB0	8112		8×8×2 bitmap, 16 bytes
0xFC	252			0x1FC0	8128		8×8×2 bitmap, 16 bytes
0xFD	253			0x1FD0	8144		8×8×2 bitmap, 16 bytes
0xFE	254			0x1FE0	8160		8×8×2 bitmap, 16 bytes
0xFF	255			0x1FF0	8176		8×8×2 bitmap, 16 bytes

Table 27: RAM_PAL Memory Map

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0x00	0			0x2000	8192		4 color palette, 8 bytes
0x01	1			0x2008	8200		4 color palette, 8 bytes
0x02	2			0x2010	8208		4 color palette, 8 bytes
0x03	3			0x2018	8216		4 color palette, 8 bytes
0x04	4			0x2020	8224		4 color palette, 8 bytes
0x05	5			0x2028	8232		4 color palette, 8 bytes
0x06	6			0x2030	8240		4 color palette, 8 bytes
0x07	7			0x2038	8248		4 color palette, 8 bytes
0x08	8			0x2040	8256		4 color palette, 8 bytes
0x09	9			0x2048	8264		4 color palette, 8 bytes
0x0A	10			0x2050	8272		4 color palette, 8 bytes
0x0B	11			0x2058	8280		4 color palette, 8 bytes
0x0C	12			0x2060	8288		4 color palette, 8 bytes
0x0D	13			0x2068	8296		4 color palette, 8 bytes
0x0E	14			0x2070	8304		4 color palette, 8 bytes
0x0F	15			0x2078	8312		4 color palette, 8 bytes
0x10	16			0x2080	8320		4 color palette, 8 bytes
0x11	17			0x2088	8328		4 color palette, 8 bytes
0x12	18			0x2090	8336		4 color palette, 8 bytes
0x13	19			0x2098	8344		4 color palette, 8 bytes
0x14	20			0x20A0	8352		4 color palette, 8 bytes
0x15	21			0x20A8	8360		4 color palette, 8 bytes
0x16	22			0x20B0	8368		4 color palette, 8 bytes
0x17	23			0x20B8	8376		4 color palette, 8 bytes
0x18	24			0x20C0	8384		4 color palette, 8 bytes
0x19	25			0x20C8	8392		4 color palette, 8 bytes
0x1A	26			0x20D0	8400		4 color palette, 8 bytes
0x1B	27			0x20D8	8408		4 color palette, 8 bytes
0x1C	28			0x20E0	8416		4 color palette, 8 bytes
0x1D	29			0x20E8	8424		4 color palette, 8 bytes
0x1E	30			0x20F0	8432		4 color palette, 8 bytes
0x1F	31			0x20F8	8440		4 color palette, 8 bytes
0x20	32			0x2100	8448		4 color palette, 8 bytes
0x21	33	!		0x2108	8456		4 color palette, 8 bytes
0x22	34	"		0x2110	8464		4 color palette, 8 bytes
0x23	35	#		0x2118	8472		4 color palette, 8 bytes
0x24	36	$		0x2120	8480		4 color palette, 8 bytes
0x25	37	%		0x2128	8488		4 color palette, 8 bytes
0x26	38	&		0x2130	8496		4 color palette, 8 bytes
0x27	39	'		0x2138	8504		4 color palette, 8 bytes
0x28	40	(		0x2140	8512		4 color palette, 8 bytes
0x29	41	)		0x2148	8520		4 color palette, 8 bytes
0x2A	42	*		0x2150	8528		4 color palette, 8 bytes
0x2B	43	+		0x2158	8536		4 color palette, 8 bytes
0x2C	44	,		0x2160	8544		4 color palette, 8 bytes
0x2D	45	-		0x2168	8552		4 color palette, 8 bytes
0x2E	46	.		0x2170	8560		4 color palette, 8 bytes
0x2F	47	/		0x2178	8568		4 color palette, 8 bytes
0x30	48	0		0x2180	8576		4 color palette, 8 bytes
0x31	49	1		0x2188	8584		4 color palette, 8 bytes
0x32	50	2		0x2190	8592		4 color palette, 8 bytes
0x33	51	3		0x2198	8600		4 color palette, 8 bytes
0x34	52	4		0x21A0	8608		4 color palette, 8 bytes
0x35	53	5		0x21A8	8616		4 color palette, 8 bytes
0x36	54	6		0x21B0	8624		4 color palette, 8 bytes
0x37	55	7		0x21B8	8632		4 color palette, 8 bytes
0x38	56	8		0x21C0	8640		4 color palette, 8 bytes
0x39	57	9		0x21C8	8648		4 color palette, 8 bytes
0x3A	58	:		0x21D0	8656		4 color palette, 8 bytes
0x3B	59	;		0x21D8	8664		4 color palette, 8 bytes
0x3C	60	<		0x21E0	8672		4 color palette, 8 bytes
0x3D	61	=		0x21E8	8680		4 color palette, 8 bytes
0x3E	62	>		0x21F0	8688		4 color palette, 8 bytes
0x3F	63	?		0x21F8	8696		4 color palette, 8 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0x40	64	@		0x2200	8704		4 color palette, 8 bytes
0x41	65	A		0x2208	8712		4 color palette, 8 bytes
0x42	66	B		0x2210	8720		4 color palette, 8 bytes
0x43	67	C		0x2218	8728		4 color palette, 8 bytes
0x44	68	D		0x2220	8736		4 color palette, 8 bytes
0x45	69	E		0x2228	8744		4 color palette, 8 bytes
0x46	70	F		0x2230	8752		4 color palette, 8 bytes
0x47	71	G		0x2238	8760		4 color palette, 8 bytes
0x48	72	H		0x2240	8768		4 color palette, 8 bytes
0x49	73	I		0x2248	8776		4 color palette, 8 bytes
0x4A	74	J		0x2250	8784		4 color palette, 8 bytes
0x4B	75	K		0x2258	8792		4 color palette, 8 bytes
0x4C	76	L		0x2260	8800		4 color palette, 8 bytes
0x4D	77	M		0x2268	8808		4 color palette, 8 bytes
0x4E	78	N		0x2270	8816		4 color palette, 8 bytes
0x4F	79	O		0x2278	8824		4 color palette, 8 bytes
0x50	80	P		0x2280	8832		4 color palette, 8 bytes
0x51	81	Q		0x2288	8840		4 color palette, 8 bytes
0x52	82	R		0x2290	8848		4 color palette, 8 bytes
0x53	83	S		0x2298	8856		4 color palette, 8 bytes
0x54	84	T		0x22A0	8864		4 color palette, 8 bytes
0x55	85	U		0x22A8	8872		4 color palette, 8 bytes
0x56	86	V		0x22B0	8880		4 color palette, 8 bytes
0x57	87	W		0x22B8	8888		4 color palette, 8 bytes
0x58	88	X		0x22C0	8896		4 color palette, 8 bytes
0x59	89	Y		0x22C8	8904		4 color palette, 8 bytes
0x5A	90	Z		0x22D0	8912		4 color palette, 8 bytes
0x5B	91	[		0x22D8	8920		4 color palette, 8 bytes
0x5C	92	\		0x22E0	8928		4 color palette, 8 bytes
0x5D	93	]		0x22E8	8936		4 color palette, 8 bytes
0x5E	94	^		0x22F0	8944		4 color palette, 8 bytes
0x5F	95	_		0x22F8	8952		4 color palette, 8 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0x60	96	`		0x2300	8960		4 color palette, 8 bytes
0x61	97	a		0x2308	8968		4 color palette, 8 bytes
0x62	98	b		0x2310	8976		4 color palette, 8 bytes
0x63	99	c		0x2318	8984		4 color palette, 8 bytes
0x64	100	d		0x2320	8992		4 color palette, 8 bytes
0x65	101	e		0x2328	9000		4 color palette, 8 bytes
0x66	102	f		0x2330	9008		4 color palette, 8 bytes
0x67	103	g		0x2338	9016		4 color palette, 8 bytes
0x68	104	h		0x2340	9024		4 color palette, 8 bytes
0x69	105	i		0x2348	9032		4 color palette, 8 bytes
0x6A	106	j		0x2350	9040		4 color palette, 8 bytes
0x6B	107	k		0x2358	9048		4 color palette, 8 bytes
0x6C	108	l		0x2360	9056		4 color palette, 8 bytes
0x6D	109	m		0x2368	9064		4 color palette, 8 bytes
0x6E	110	n		0x2370	9072		4 color palette, 8 bytes
0x6F	111	o		0x2378	9080		4 color palette, 8 bytes
0x70	112	p		0x2380	9088		4 color palette, 8 bytes
0x71	113	q		0x2388	9096		4 color palette, 8 bytes
0x72	114	r		0x2390	9104		4 color palette, 8 bytes
0x73	115	s		0x2398	9112		4 color palette, 8 bytes
0x74	116	t		0x23A0	9120		4 color palette, 8 bytes
0x75	117	u		0x23A8	9128		4 color palette, 8 bytes
0x76	118	v		0x23B0	9136		4 color palette, 8 bytes
0x77	119	w		0x23B8	9144		4 color palette, 8 bytes
0x78	120	x		0x23C0	9152		4 color palette, 8 bytes
0x79	121	y		0x23C8	9160		4 color palette, 8 bytes
0x7A	122	z		0x23D0	9168		4 color palette, 8 bytes
0x7B	123	{		0x23D8	9176		4 color palette, 8 bytes
0x7C	124	|		0x23E0	9184		4 color palette, 8 bytes
0x7D	125	}		0x23E8	9192		4 color palette, 8 bytes
0x7E	126	~		0x23F0	9200		4 color palette, 8 bytes
0x7F	127	▨		0x23F8	9208		4 color palette, 8 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0x80	128			0x2400	9216		4 color palette, 8 bytes
0x81	129			0x2408	9224		4 color palette, 8 bytes
0x82	130			0x2410	9232		4 color palette, 8 bytes
0x83	131			0x2418	9240		4 color palette, 8 bytes
0x84	132			0x2420	9248		4 color palette, 8 bytes
0x85	133			0x2428	9256		4 color palette, 8 bytes
0x86	134			0x2430	9264		4 color palette, 8 bytes
0x87	135			0x2438	9272		4 color palette, 8 bytes
0x88	136			0x2440	9280		4 color palette, 8 bytes
0x89	137			0x2448	9288		4 color palette, 8 bytes
0x8A	138			0x2450	9296		4 color palette, 8 bytes
0x8B	139			0x2458	9304		4 color palette, 8 bytes
0x8C	140			0x2460	9312		4 color palette, 8 bytes
0x8D	141			0x2468	9320		4 color palette, 8 bytes
0x8E	142			0x2470	9328		4 color palette, 8 bytes
0x8F	143			0x2478	9336		4 color palette, 8 bytes
0x90	144			0x2480	9344		4 color palette, 8 bytes
0x91	145			0x2488	9352		4 color palette, 8 bytes
0x92	146			0x2490	9360		4 color palette, 8 bytes
0x93	147			0x2498	9368		4 color palette, 8 bytes
0x94	148			0x24A0	9376		4 color palette, 8 bytes
0x95	149			0x24A8	9384		4 color palette, 8 bytes
0x96	150			0x24B0	9392		4 color palette, 8 bytes
0x97	151			0x24B8	9400		4 color palette, 8 bytes
0x98	152			0x24C0	9408		4 color palette, 8 bytes
0x99	153			0x24C8	9416		4 color palette, 8 bytes
0x9A	154			0x24D0	9424		4 color palette, 8 bytes
0x9B	155			0x24D8	9432		4 color palette, 8 bytes
0x9C	156			0x24E0	9440		4 color palette, 8 bytes
0x9D	157			0x24E8	9448		4 color palette, 8 bytes
0x9E	158			0x24F0	9456		4 color palette, 8 bytes
0x9F	159			0x24F8	9464		4 color palette, 8 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0xA0	160			0x2500	9472		4 color palette, 8 bytes
0xA1	161			0x2508	9480		4 color palette, 8 bytes
0xA2	162			0x2510	9488		4 color palette, 8 bytes
0xA3	163			0x2518	9496		4 color palette, 8 bytes
0xA4	164			0x2520	9504		4 color palette, 8 bytes
0xA5	165			0x2528	9512		4 color palette, 8 bytes
0xA6	166			0x2530	9520		4 color palette, 8 bytes
0xA7	167			0x2538	9528		4 color palette, 8 bytes
0xA8	168			0x2540	9536		4 color palette, 8 bytes
0xA9	169			0x2548	9544		4 color palette, 8 bytes
0xAA	170			0x2550	9552		4 color palette, 8 bytes
0xAB	171			0x2558	9560		4 color palette, 8 bytes
0xAC	172			0x2560	9568		4 color palette, 8 bytes
0xAD	173			0x2568	9576		4 color palette, 8 bytes
0xAE	174			0x2570	9584		4 color palette, 8 bytes
0xAF	175			0x2578	9592		4 color palette, 8 bytes
0xB0	176			0x2580	9600		4 color palette, 8 bytes
0xB1	177			0x2588	9608		4 color palette, 8 bytes
0xB2	178			0x2590	9616		4 color palette, 8 bytes
0xB3	179			0x2598	9624		4 color palette, 8 bytes
0xB4	180			0x25A0	9632		4 color palette, 8 bytes
0xB5	181			0x25A8	9640		4 color palette, 8 bytes
0xB6	182			0x25B0	9648		4 color palette, 8 bytes
0xB7	183			0x25B8	9656		4 color palette, 8 bytes
0xB8	184			0x25C0	9664		4 color palette, 8 bytes
0xB9	185			0x25C8	9672		4 color palette, 8 bytes
0xBA	186			0x25D0	9680		4 color palette, 8 bytes
0xBB	187			0x25D8	9688		4 color palette, 8 bytes
0xBC	188			0x25E0	9696		4 color palette, 8 bytes
0xBD	189			0x25E8	9704		4 color palette, 8 bytes
0xBE	190			0x25F0	9712		4 color palette, 8 bytes
0xBF	191			0x25F8	9720		4 color palette, 8 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0xC0	192			0x2600	9728		4 color palette, 8 bytes
0xC1	193			0x2608	9736		4 color palette, 8 bytes
0xC2	194			0x2610	9744		4 color palette, 8 bytes
0xC3	195			0x2618	9752		4 color palette, 8 bytes
0xC4	196			0x2620	9760		4 color palette, 8 bytes
0xC5	197			0x2628	9768		4 color palette, 8 bytes
0xC6	198			0x2630	9776		4 color palette, 8 bytes
0xC7	199			0x2638	9784		4 color palette, 8 bytes
0xC8	200			0x2640	9792		4 color palette, 8 bytes
0xC9	201			0x2648	9800		4 color palette, 8 bytes
0xCA	202			0x2650	9808		4 color palette, 8 bytes
0xCB	203			0x2658	9816		4 color palette, 8 bytes
0xCC	204			0x2660	9824		4 color palette, 8 bytes
0xCD	205			0x2668	9832		4 color palette, 8 bytes
0xCE	206			0x2670	9840		4 color palette, 8 bytes
0xCF	207			0x2678	9848		4 color palette, 8 bytes
0xD0	208			0x2680	9856		4 color palette, 8 bytes
0xD1	209			0x2688	9864		4 color palette, 8 bytes
0xD2	210			0x2690	9872		4 color palette, 8 bytes
0xD3	211			0x2698	9880		4 color palette, 8 bytes
0xD4	212			0x26A0	9888		4 color palette, 8 bytes
0xD5	213			0x26A8	9896		4 color palette, 8 bytes
0xD6	214			0x26B0	9904		4 color palette, 8 bytes
0xD7	215			0x26B8	9912		4 color palette, 8 bytes
0xD8	216			0x26C0	9920		4 color palette, 8 bytes
0xD9	217			0x26C8	9928		4 color palette, 8 bytes
0xDA	218			0x26D0	9936		4 color palette, 8 bytes
0xDB	219			0x26D8	9944		4 color palette, 8 bytes
0xDC	220			0x26E0	9952		4 color palette, 8 bytes
0xDD	221			0x26E8	9960		4 color palette, 8 bytes
0xDE	222			0x26F0	9968		4 color palette, 8 bytes
0xDF	223			0x26F8	9976		4 color palette, 8 bytes

Character				Address			Contents
(hex)	(dec)	(g)		(hex)	(dec)		
0xE0	224			0x2700	9984		4 color palette, 8 bytes
0xE1	225			0x2708	9992		4 color palette, 8 bytes
0xE2	226			0x2710	10000		4 color palette, 8 bytes
0xE3	227			0x2718	10008		4 color palette, 8 bytes
0xE4	228			0x2720	10016		4 color palette, 8 bytes
0xE5	229			0x2728	10024		4 color palette, 8 bytes
0xE6	230			0x2730	10032		4 color palette, 8 bytes
0xE7	231			0x2738	10040		4 color palette, 8 bytes
0xE8	232			0x2740	10048		4 color palette, 8 bytes
0xE9	233			0x2748	10056		4 color palette, 8 bytes
0xEA	234			0x2750	10064		4 color palette, 8 bytes
0xEB	235			0x2758	10072		4 color palette, 8 bytes
0xEC	236			0x2760	10080		4 color palette, 8 bytes
0xED	237			0x2768	10088		4 color palette, 8 bytes
0xEE	238			0x2770	10096		4 color palette, 8 bytes
0xEF	239			0x2778	10104		4 color palette, 8 bytes
0xF0	240			0x2780	10112		4 color palette, 8 bytes
0xF1	241			0x2788	10120		4 color palette, 8 bytes
0xF2	242			0x2790	10128		4 color palette, 8 bytes
0xF3	243			0x2798	10136		4 color palette, 8 bytes
0xF4	244			0x27A0	10144		4 color palette, 8 bytes
0xF5	245			0x27A8	10152		4 color palette, 8 bytes
0xF6	246			0x27B0	10160		4 color palette, 8 bytes
0xF7	247			0x27B8	10168		4 color palette, 8 bytes
0xF8	248			0x27C0	10176		4 color palette, 8 bytes
0xF9	249			0x27C8	10184		4 color palette, 8 bytes
0xFA	250			0x27D0	10192		4 color palette, 8 bytes
0xFB	251			0x27D8	10200		4 color palette, 8 bytes
0xFC	252			0x27E0	10208		4 color palette, 8 bytes
0xFD	253			0x27E8	10216		4 color palette, 8 bytes
0xFE	254			0x27F0	10224		4 color palette, 8 bytes
0xFF	255			0x27F8	10232		4 color palette, 8 bytes

Sprites

Sprites are graphical objects with their own bitmaps and palettes that can be positioned anywhere over the underlying character graphics playfield. They the most complex part of the Gameduino's video subsystem, using six distinct areas of memory for control, collision detection, image data, and color palettes.

Sprites are visually positioned above the other elements of the video display, so that opaque pixels of a sprite will obscure character graphics playfield and background pixel information underneath. Sprite pixels may be transparent, allowing the underlying graphical layers to show through. Each sprite is numbered from 0 to 255 (0x00 to 0xFF hexadecimal), and are rendered in order, so that sprite 0 is visually beneath sprite 1, and so on to sprite 255, which is the highest-priority visual element on the display.

The Gameduino hardware supports 256 sprites, with some limitations:

- **Sprites per Line:** There is a limit of 96 sprites on a single horizontal line. If too many sprites occupy the same line, the Gameduino hardware runs out of time to compose the line, and some sprites may not be drawn.
- **Sprite Bitmap Memory:** RAM_SPRIMG contains enough memory to store 64 sprite images at the full 8-bit (256-color) pixel depth. However, two sprites can share one bitmap in 4-bit (16-color) mode: one sprite uses the upper nybble of the byte, while the other sprite uses the lower nybble. Similarly, in 2-bit (4-color) mode, a total of 4 sprites can share one bitmap.

Sprite Control

Each sprite is controlled by a 32-bit sprite control word, diagrammed in Table 28: Sprite Control Word. The hardware supports a maximum of 256 sprites, so sprite control data occupies 1024 bytes of memory. The Gameduino has two pages of sprite control data: one at 0x3000 to 0x33FF (1228 to 13311 decimal) and one at 0x3400 to 0x37FF (13312 to 14335 decimal) to support double buffering. The active page is set by the SPR_PAGE register at 0x280B (10251 decimal). If SPR_PAGE is 0, the lower bank of sprite control data is used; the upper bank is used when SPR_PAGE is 1. The inactive page can be modified without affecting the screen display.

Table 28: Sprite Control Word

Sprite Control	Byte Offset and Control Word Contents			
	+0	+1	+2	+3
Sprite Control	$X_7X_6X_5X_4X_3X_2X_1X_0$	$P_3P_2P_1P_0$ $R_2R_1R_0$ X_8	$Y_7Y_6Y_5Y_4Y_3Y_2Y_1Y_0$	C $S_5S_4S_3S_2S_1S_0$ Y_8

X_0–X_8: Sprite X position, 0-511 (0x000-0x1FF hexadecimal).

P_0–R_3: Palette mode select, 0-15 (0x0-0xF hexadecimal), see Table 29: Sprite Palette Modes.

R0–R2: Sprite rotation, 0-8; see Table 30: Sprite Rotation.

Y_0–Y_8: Sprite Y position, 0-511 (0x000-0x1FF hexadecimal).

C: Collision class for J/K mode, 0=J 1=K.

S_0–S_5: Source image bitmap, 0-63 (0x00-0x3F hexadecimal).

Table 29: Sprite Palette Modes

Palette Mode P_0-P_3			Selected Mode				Pixel
(bin)	(dec)	(hex)	colors		palette		bits
0000	0	0x0	256	8	0	A	0-7
0001	1	0x1	256	8	1	B	0-7
0010	2	0x2	256	8	2	C	0-7
0011	3	0x3	256	8	3	D	0-7
0100	4	0x4	16	4	0	A	0-3
0101	5	0x5	16	4	1	B	0-3
0110	6	0x6	16	4	0	A	4-7
0111	7	0x7	16	4	1	B	4-7
1000	8	0x8	4	2	0	A	0-1
1001	9	0x9	4	2	1	B	0-1
1010	10	0xA	4	2	0	A	2-3
1011	11	0xB	4	2	1	B	2-3
1100	12	0xC	4	2	0	A	4-5
1101	13	0xD	4	2	1	B	4-5
1110	14	0xE	4	2	0	A	6-7
1111	15	0xF	4	2	1	B	6-7

Palette mode: The selected palette mode number in binary, decimal, and hexadecimal.
Colors: Number of colors and bit depth in selected palette mode.
Palette: The number and letter of the palette selected for the sprite.
Pixel bits: The bit positions of the sprite image used in the selected mode.

Table 30: Sprite Rotation

Rotation R_0-R_2			Selected Mode			Effect
(bin)	(dec)	(hex)	Yflip	Xflip	XYswap	
000	0	0x0	0	0	0	None
001	1	0x1	0	0	1	Mirrored left-to-right then rotated 270°
010	2	0x2	0	1	0	Mirrored left to right
011	3	0x3	0	1	1	270° clockwise rotation
100	4	0x4	1	0	0	Mirrored left-to-right then rotated 180°
101	5	0x5	1	0	1	90° clockwise rotation
110	6	0x6	1	1	0	180° clockwise rotation
111	7	0x7	1	1	1	Mirrored left-to-right then rotated 90°

Rotation: The selected rotation number in binary, decimal, and hexadecimal.
Y flip: Flip sprite image in Y direction (top-to-bottom), 0=no, 1=yes.
X flip: Flip sprite image in X direction (left-to-right), 0=no, 1=yes.
XY swap: Swap sprite X and Y axes, 0=no, 1=yes.
Effect: Effect of selected rotation on the sprite image.

Sprite Collision Detection

The Gameduino sprite engine detects collisions while the on-screen display is being scanned out. If a given sprite covered up (obscured pixels from) another sprite, then a collision is detected. The results are available in COLLISION, a 256-byte area of memory from 0x2900-0x29FF (10496-10751 decimal). This area of memory is valid only during vertical blanking (VBLANK=1) and will read 0xFF (decimal 255) at all other times.

Each byte of COLLISION represents the collision status of the corresponding sprite. The byte is 0xFF (255 decimal) if the sprite did not collide with any other sprite, or contains the sprite number, 0x00-0xFE (0 to 254) of the sprite that was obscured. Since sprite 255 (0xFF hexadecimal) is the highest-priority sprite, it can never be obscured by another sprite. Only one collision is reported per sprite; if a sprite collides with multiple other sprites, one of the collisions will be reported, and the others are discarded.

If J/K collision mode is enabled, by setting the JK_MODE register to 1, collisions will only be detected between sprites belonging to different collision classes. For example, in J/K mode, if two sprites belong to class J (C=0), they cannot collide even if one sprite obscures pixels from the other. Only sprites with class bits can collide. If J/K mode is disabled, all collisions will be reported regardless of class membership.

Sprite Image Data

Sprite images are stored in 16kbytes of RAM beginning at 0x4000 (16384 decimal). This is enough storage for 64 image bitmaps: each sprite image is a 16×16 bitmap where each pixel occupies a single byte. The resulting bitmap is 256 bytes, and is diagrammed in Table 31: Sprite Image Bitmap.

Table 31: Sprite Image Bitmap

Row		Byte Offset and Bitmap Contents															
(h)	(d)	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
+0x00	+0	P_{01}	P_{02}	P_{00}	P_{03}	P_{04}	P_{05}	P_{06}	P_{07}	P_{08}	P_{09}	P_{0A}	P_{0B}	P_{0C}	P_{0D}	P_{0E}	P_{0F}
+0x10	+0	P_{11}	P_{12}	P_{11}	P_{13}	P_{14}	P_{15}	P_{16}	P_{17}	P_{18}	P_{19}	P_{1A}	P_{1B}	P_{1C}	P_{1D}	P_{1E}	P_{1F}
+0x20	+0	P_{21}	P_{22}	P_{22}	P_{23}	P_{24}	P_{25}	P_{26}	P_{27}	P_{28}	P_{29}	P_{2A}	P_{2B}	P_{2C}	P_{2D}	P_{2E}	P_{2F}
+0x30	+0	P_{31}	P_{32}	P_{33}	P_{33}	P_{34}	P_{35}	P_{36}	P_{37}	P_{38}	P_{39}	P_{3A}	P_{3B}	P_{3C}	P_{3D}	P_{3E}	P_{3F}
+0x40	+0	P_{41}	P_{42}	P_{40}	P_{43}	P_{44}	P_{45}	P_{46}	P_{47}	P_{48}	P_{49}	P_{4A}	P_{4B}	P_{4C}	P_{4D}	P_{4E}	P_{4F}
+0x50	+0	P_{51}	P_{52}	P_{50}	P_{53}	P_{54}	P_{55}	P_{56}	P_{57}	P_{58}	P_{59}	P_{5A}	P_{5B}	P_{5C}	P_{5D}	P_{5E}	P_{5F}
+0x60	+0	P_{61}	P_{62}	P_{60}	P_{63}	P_{64}	P_{65}	P_{66}	P_{67}	P_{68}	P_{69}	P_{6A}	P_{6B}	P_{6C}	P_{6D}	P_{6E}	P_{6F}
+0x70	+0	P_{71}	P_{72}	P_{70}	P_{73}	P_{74}	P_{75}	P_{76}	P_{77}	P_{78}	P_{79}	P_{7A}	P_{7B}	P_{7C}	P_{7D}	P_{7E}	P_{7F}
+0x80	+0	P_{81}	P_{82}	P_{80}	P_{83}	P_{84}	P_{85}	P_{86}	P_{87}	P_{88}	P_{89}	P_{8A}	P_{8B}	P_{8C}	P_{8D}	P_{8E}	P_{8F}
+0x90	+0	P_{91}	P_{92}	P_{90}	P_{93}	P_{94}	P_{95}	P_{96}	P_{97}	P_{98}	P_{99}	P_{9A}	P_{9B}	P_{9C}	P_{9D}	P_{9E}	P_{9F}
+0xA0	+0	P_{A1}	P_{A2}	P_{A0}	P_{A3}	P_{A4}	P_{A5}	P_{A6}	P_{A7}	P_{A8}	P_{A9}	P_{AA}	P_{AB}	P_{AC}	P_{AD}	P_{AE}	P_{AF}
+0xB0	+0	P_{B1}	P_{B2}	P_{B0}	P_{B3}	P_{B4}	P_{B5}	P_{B6}	P_{B7}	P_{B8}	P_{B9}	P_{BA}	P_{BB}	P_{BC}	P_{BD}	P_{BE}	P_{BF}
+0xC0	+0	P_{C1}	P_{C2}	P_{C0}	P_{C3}	P_{C4}	P_{C5}	P_{C6}	P_{C7}	P_{C8}	P_{C9}	P_{CA}	P_{CB}	P_{CC}	P_{CD}	P_{CE}	P_{CF}
+0xD0	+0	P_{D1}	P_{D2}	P_{D0}	P_{D3}	P_{D4}	P_{D5}	P_{D6}	P_{D7}	P_{D8}	P_{D9}	P_{DA}	P_{DB}	P_{DC}	P_{DD}	P_{DE}	P_{DF}
+0xE0	+0	P_{E1}	P_{E2}	P_{E0}	P_{E3}	P_{E4}	P_{E5}	P_{E6}	P_{E7}	P_{E8}	P_{E9}	P_{EA}	P_{EB}	P_{EC}	P_{ED}	P_{EE}	P_{EF}
+0xF0	+0	P_{F1}	P_{F2}	P_{F0}	P_{F3}	P_{F4}	P_{F5}	P_{F6}	P_{F7}	P_{F8}	P_{F9}	P_{FA}	P_{FB}	P_{FC}	P_{FD}	P_{FE}	P_{FF}

P_{XY}: Pixel data for position X,Y, where X and Y are 0x0-0xF (0 to 15 decimal).

Up to 256 different sprite images are supported, by allowing sprites to share a bitmap. In 16-color (4 bit per pixel) mode, each sprite bitmap can contain two complete sprite images, one in the high nybble (bits 4-7) and one in the low nybble (bits 0-3). In 4-color (2 bit per pixel) mode, each sprite bitmap can contain four complete sprite images. Each image occupies 2 of the 8 bits. See Table 28: Sprite Control Word and Table 29: Sprite Palette Modes for more information.

Sprite Palettes

There are a total of eight sprite palettes available on the Gameduino: 4 256-color palettes, plus 2 16-color and 2 4-color palettes. All eight palettes are fully independent, and there is no restriction, other than available sprite image data memory, on mixing sprite modes or palettes.

256-Color Palettes: The four 256-color palettes are each 512 bytes long, and are located at RAM_SPRPAL starting at 0x3800 (14336 decimal). The structure of RAM_SPRPAL is diagrammed in Table 32: RAM_SPRPAL Memory Map, while each palette consists of 256 color entries as described in Table 37: Palette Entry Format.

Table 32: RAM_SPRPAL Memory Map

Symbol		Palette		Address		Length		Contents
	alternate			(hex)	(dec)	(hex)	(dec)	
RAM_SPRPAL	PALETTE256A	0	A	0x3800	14336	0x200	512	256-color palette A
	PALETTE256B	1	B	0x3A00	14848	0x200	512	256-color palette B
	PALETTE256C	2	C	0x3C00	15360	0x200	512	256-Color palette C
	PALETTE256C	3	D	0x3F00	15872	0x200	512	256-Color palette D

16-Color Palettes: There are two 16-color palettes, each 32 bytes long, located at PALETTE16 starting at 0x2840 (10304 decimal). The structure of PALETTE16 is diagrammed in Table 33: PALETTE16 Memory Map, while each palette consists of 16 color entries as described in Table 37: Palette Entry Format, and detailed in Table 34: 16-Color Sprite Palette.

Table 33: PALETTE16 Memory Map

Symbol	Palette		Address		Length		Contents
			(hex)	(dec)	(hex)	(dec)	
PALETTE16A	0	A	0x2840	10304	0x20	32	16-Color sprite palette A
PALETTE16B	1	B	0x2860	10336	0x20	32	16-Color sprite palette B

Table 34: 16-Color Sprite Palette

Color	Entry Offset		Byte Offset	
Entry	(hex)	(dec)	+0	+1
0	+0x00	+0	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
1	+0x02	+2	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
2	+0x04	+4	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
3	+0x06	+6	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
4	+0x08	+8	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
5	+0x0A	+10	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
6	+0x0C	+12	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
7	+0x0E	+14	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
8	+0x10	+16	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
9	+0x12	+18	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
10	+0x14	+20	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
11	+0x16	+22	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
12	+0x18	+24	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
13	+0x1A	+26	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
14	+0x1C	+28	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
15	+0x1E	+30	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$

A: Alpha channel color information, 0=opaque, 1=transparent.
R_0–R_4: Red channel color information, 0-32; ignored when A=1.
B_0–B_4: Green channel color information, 0-32; ignored when A=1.
B_0–B_4: Blue channel color information, 0-32; ignored when A=1.

4-Color Palettes: There are two 4-color palettes, each 8 bytes long, located at PALETTE4 starting at 0x2880 (10368 decimal). The structure of PALETTE4 is diagrammed in Table 35: PALETTE4 Memory Map, while each palette consists of 16 color entries as described in Table 37: Palette Entry Format, and detailed in Table 36: 4-Color Sprite Palette.

Table 35: PALETTE4 Memory Map

Symbol	Palette		Address		Length		Contents
			(hex)	(dec)	(hex)	(dec)	
PALETTE4A	0	A	0x2880	10368	0x8	8	4-Color sprite palette A
PALETTE4B	1	B	0x2888	10376	0x8	8	4-Color sprite palette B

Table 36: 4-Color Sprite Palette

Color Entry	Entry Offset	Byte Offset	
		+0	+1
0	+0	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
1	+2	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
2	+4	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
3	+6	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$

A: Alpha channel color information, 0=opaque, 1=transparent.
R_0–R_4: Red channel color information, 0-32; ignored when A=1.
B_0–B_4: Green channel color information, 0-32; ignored when A=1.
B_0–B_4: Blue channel color information, 0-32; ignored when A=1.

Color

The Gameduino uses a number of different formats to represent color values: 8-bit logical color space values used by the GD Library, 5-bit color values used by the hardware, and 3-bit color values combined with dithering to generate video output to the display. Table 38: Color Space shows the relationship between these three color spaces.

Logical Color

The GD Library implements a 24-bit RGB888 logical color space. It provides 8 bits per channel via an RGB() macro that accepts one byte of color data for each channel. The macro discards the least significant 3 bits of each color channel, and returns a 15-bit hardware representation of the color, with the alpha channel set to 0 (opaque).

Hardware Color

Internally, the Gameduino uses an ARGB1555 color format in its palettes and color registers. Color values are stored in a single 16-bit word, with one bit of alpha (transparency) information, and 5 bits each of red, green, and blue channel information. The lower (least significant) byte of the word contains the blue channel data and the least significant bits of the green channel, while the upper byte contains the most significant bits of the green channel, all of the red channel, and the alpha channel. A standard color value is diagrammed below:

Table 37: Palette Entry Format

Address	+0	+1
ARGB1555 Color Data Template	$G_2G_1G_0B_4B_3B_2B_1B_0$	A $R_4R_3R_2R_1R_0G_4G_3$
ARGB1555 Display Colors	$G_2G_1G_0B_4B_3B_2B_1B_0$	0 $R_4R_3R_2R_1R_0G_4G_3$
ARGB1555 Transparent	X X X X X X X X	1 X X X X X X X

A: Alpha channel color information, 0=opaque, 1=transparent.
R_0–R_4: Red channel color information, 0-32; ignored when A=1.
B_0–B_4: Green channel color information, 0-32; ignored when A=1.
B_0–B_4: Blue channel color information, 0-32; ignored when A=1.

As shown above, the Gameduino implements an alpha (transparency) color channel in hardware. Transparent values are a special case of the standard color value: when the most significant bit is set, the low 15 bits containing the red, green, and blue channels in the color value are ignored. The result is values of 0x8000-0xFFFF (decimal 32768 through 65535), are transparent.

By convention, 0x8000 is the standard transparent color value. However, storing 0x80 (decimal 128) in the high byte of a color word is sufficient to make a color entry transparent, regardless of the contents of the low byte.

The BG_COLOR register ignores alpha channel information, because the background color is the bottom-most color plane. There is no lower-priority video information to show through a transparent background color, so all color values are treated as opaque. Similarly, pixel data read back through the SCREENSHOT memory area are always opaque, since they represent the final color values output to the display.

Output Color

The Gameduino display output has only 3 bits of resolution per channel. Only the colors where the three least significant bits are all zero are output directly. The other colors, where the least significant bits are nonzero, are approximated using a 2×2 dithering algorithm in the video output hardware.

Table 38: Color Space

8-bit Logical Color			5-bit Hardware Color			3-bit Output Color		
(hex)	(dec)	(bin)	(hex)	(dec)	(bin)	(hex)	(dec)	(bin)
0x00	0	00000000	0x00	0	00000	0x00	0	000
0x08	8	00001000	0x01	1	00001	0x00	0	000
0x10	16	00010000	0x02	2	00010	0x00	0	000
0x18	24	00011000	0x03	3	00011	0x00	0	000
0x20	32	00100000	0x04	4	00100	0x01	1	001
0x28	40	00101000	0x05	5	00101	0x01	1	001
0x30	48	00110000	0x06	6	00110	0x01	1	001
0x38	56	00111000	0x07	7	00111	0x01	1	001
0x40	64	01000000	0x08	8	01000	0x02	2	010

8-bit Logical Color			5-bit Hardware Color			3-bit Output Color		
(hex)	(dec)	(bin)	(hex)	(dec)	(bin)	(hex)	(dec)	(bin)
0x48	72	01001000	0x09	9	01001	0x02	2	010
0x50	80	01010000	0x0A	10	01010	0x02	2	010
0x58	88	01011000	0x0B	11	01011	0x02	2	010
0x60	96	01100000	0x0C	12	01100	0x03	3	011
0x68	104	01101000	0x0D	13	01101	0x03	3	011
0x70	112	01110000	0x0E	14	01110	0x03	3	011
0x78	120	01111000	0x0F	15	01111	0x03	3	011
0x80	128	10000000	0x10	16	10000	0x04	4	100
0x88	136	10001000	0x11	17	10001	0x04	4	100
0x90	144	10010000	0x12	18	10010	0x04	4	100
0x98	152	10011000	0x13	19	10011	0x04	4	100
0xA0	160	10100000	0x14	20	10100	0x05	5	101
0xA8	168	10101000	0x15	21	10101	0x05	5	101
0xB0	176	10110000	0x16	22	10110	0x05	5	101
0xB8	184	10111000	0x17	23	10111	0x05	5	101
0xC0	192	11000000	0x18	24	11000	0x06	6	110
0xC8	200	11001000	0x19	25	11001	0x06	6	110
0xD0	208	11010000	0x1A	26	11010	0x06	6	110
0xD8	216	11011000	0x1B	27	11011	0x06	6	110
0xE0	224	11100000	0x1C	28	11100	0x07	7	111
0xE8	232	11101000	0x1D	29	11101	0x07	7	111
0xF0	240	11110000	0x1E	30	11110	0x07	7	111
0xF8	248	11111000	0x1F	31	11111	0x07	7	111

Lines shaded in grey are dithered on output.

Audio

The Gameduino implements two independent audio systems: an additive synthesis based on the summation of separate waveforms, and a sample playback mechanism.

Sound Synthesis Registers

The Gameduino additive sound synthesis system is based on 64 sound control words and is supplemented by a ring modulation system.

Voice Control

Sound control data is located at VOICES, 0x2A00 (10752 decimal). Each sound control word consists of 32 bits that specify one component of the final sound to be output.

Table 39: VOICES Memory Map

V	Address		Byte Offset and Control Word Contents			
	(hex)	(dec)	+0	+1	+2	+3
0	0x2A00	10752	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
1	0x2A04	10756	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
2	0x2A08	10760	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
3	0x2A0C	10764	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
4	0x2A10	10768	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
5	0x2A14	10772	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
6	0x2A18	10776	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$

V	Address		Byte Offset and Control Word Contents			
	(hex)	(dec)	+0	+1	+2	+3
7	0x2A1C	10780	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
8	0x2A20	10784	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
9	0x2A24	10788	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
10	0x2A28	10792	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
11	0x2A2C	10796	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
12	0x2A30	10800	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
13	0x2A34	10804	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
14	0x2A38	10808	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
15	0x2A3C	10812	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
16	0x2A40	10816	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
17	0x2A44	10820	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
18	0x2A48	10824	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
19	0x2A4C	10828	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
20	0x2A50	10832	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
21	0x2A54	10836	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
22	0x2A58	10840	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
23	0x2A5C	10844	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
24	0x2A60	10848	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
25	0x2A64	10852	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
26	0x2A68	10856	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
27	0x2A6C	10860	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
28	0x2A70	10864	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
29	0x2A74	10868	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
30	0x2A78	10872	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
31	0x2A7C	10876	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
32	0x2A80	10880	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
33	0x2A84	10884	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
34	0x2A88	10888	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
35	0x2A8C	10892	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
36	0x2A90	10896	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
37	0x2A94	10900	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
38	0x2A98	10904	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
39	0x2A9C	10908	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
40	0x2AA0	10912	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
41	0x2AA4	10916	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
42	0x2AA8	10920	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
43	0x2AAC	10924	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
44	0x2AB0	10928	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
45	0x2AB4	10932	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
46	0x2AB8	10936	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
47	0x2ABC	10940	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$

V	Address		Byte Offset and Control Word Contents			
	(hex)	(dec)	+0	+1	+2	+3
48	0x2AC0	10944	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
49	0x2AC4	10948	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
50	0x2AC8	10952	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
51	0x2ACC	10956	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
52	0x2AD0	10960	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
53	0x2AD4	10964	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
54	0x2AD8	10968	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
55	0x2ADC	10972	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
56	0x2AE0	10976	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
57	0x2AE4	10980	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
58	0x2AE8	10984	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
59	0x2AEC	10988	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
60	0x2AF0	10992	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
61	0x2AF4	10996	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
62	0x2AF8	11000	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$
63	0x2AFC	11004	$F_7F_6F_5F_4F_3F_2F_1F_0$	W $F_EF_DF_CF_BF_AF_9F_8$	$L_7L_6L_5L_4L_3L_2L_1L_0$	$R_7R_6R_5R_4R_3R_2R_1R_0$

V: Voice number, 0-64 (0x00-0x40 hexadecimal).
F_0–F_E: Frequency, in units of ¼Hz (so 4=1Hz or 1760=440Hz).
W: Waveform, 0=sine, 1=noise.
L_0-L_7: Left volume, 0-255 (0x00-0xFF hexadecimal).
R_0-R_7: Right volume, 0-255 (0x00-0xFF hexadecimal).

Ring Modulator

In addition to simple additive synthesis, the Gameduino's audio output system implements a ring modulator. Writing values to the RING_MOD register at 0x2814 (10260 decimal) control the ring modulator. Writing 64 (0x40) to the RING_MOD register disables the feature. Writing any lower value enables the ring modulator. The value stored in RING_MOD determines which voice control register is used as the ring modulator control. All lower-numbered voices are modulated, while higher-numbered voices are unaffected. For example, writing 32 (0x20 hexadecimal) to RING_MOD enables the ring modulator. Voice control word 32 controls the modulator, and voices 0-31 are modulated, while voices 33-64 are not.

Symbol		Address		L	Register Contents and Function	Default
		(hex)	(dec)			
RING_MOD	RW	0x2814	10260	1	Modulator voice, affects all lower voices	0x40

Sample Registers

The Gameduino has the ability to play back stereo audio wave data using the SAMPLE_L and SAMPLE_R registers. Each register accepts a signed 16-bit integer reflecting the sample. The Gameduino updates audio samples every 64 cycles of it's 50MHz clock, corresponding to an update frequency of 781.25kHz, so values loaded into these registers are reflected on the audio output channels within 1.28μs.

Table 40: SAMPLE_L and SAMPLE_R Registers

Symbol		Address		Length		Byte Offset and Register Contents		Default
		(hex)	(dec)	(hex)	(dec)	+0	+1	
SAMPLE_L	RW	0x2810	10256	0x2	2	$L_7L_6L_5L_4L_3L_2L_1L_0$	$L_{15}L_{14}L_{13}L_{12}L_{11}L_{10}L_9L_8$	0x0000
SCROLL_R	RW	0x2812	10258	0x2	2	$R_7R_6R_5R_4R_3R_2R_1R_0$	$R_{15}R_{14}R_{13}R_{12}R_{11}R_{10}R_9R_8$	0x0000

L_0–L_{15}: Left audio channel sample value, 16-bit signed integer (L_{15} is the sign bit).
R_0–R_{15}: Right audio channel sample value, 16-bit signed integer (R_{15} is the sign bit).

J1 Coprocessor

The Gameduino includes a J1 Forth-based CPU as a coprocessor. The J1 is a minimalist but fully functional processor that has full access to the Gameduino's address space, and a few additional registers that are not accessible to the Arduino. The J1 CPU also executes much faster than the Arduino, and can move considerably more data than can fit through the SPI interface.

Coprocessor Interface

The Gameduino coprocessor interface consists of a control register, a microcode block, and a communications block that are shared between the Arduino and the coprocessor.

Coprocessor Control

The J1_RESET register allows the Arduino (or other host microcontroller) to stop and start the J1 coprocessor. Writing 0x01 to this register halts and resets the coprocessor, while 0x00 releases the coprocessor. Following a reset, the J1 begins executing code starting at J1_CODE, 0x2B00 (11008 decimal). The remaining seven bits of the register are unused and should be set to 0 for compatibility.

Table 41: J1_RESET Register

Symbol		Address		Length		Contents	Default
		(hex)	(dec)	(hex)	(dec)		
`J1_RESET`	`RW`	`0x2809`	`10249`	`0x1`	`1`	X X X X X X X R_0	`0x0000`

R_0: Coprocessor control: 1=run, 0=halt and reset.

Coprocessor Microcode

J1_CODE, a 256-byte block of Gameduino RAM from 0x2B00-0x2BFF (decimal 11008 through 11263) is set aside for J1 instructions. This is the only area of memory that the J1 can use to fetch instructions. J1 instructions are 16-bit words, so J1_CODE can contain only 128 instructions. However the coprocessor can read and write data from any location in the Gameduino address space, so clever programs can copy overlays from other areas of the J1 address space into J1_CODE. Unused sprite bitmap memory in the 16 kbyte RAM_SPRIMG block is a good place to locate J1 overlays.

Table 42: J1_CODE Memory Map

Symbol		Address		Length		Contents	Default
		(hex)	(dec)	(hex)	(dec)		
`J1_CODE`	`RW`	`0x2B00`	`11008`	`0x0100`	`256`	128 2-byte J1 instructions	`128 × 0x0000`

Coprocessor Communications Block

An otherwise-unused 48-byte block of RAM is set aside for use as a inter-processor communications area so that Arduino and J1 code can exchange data. The block is labeled COMM and runs from 0x2890 to 0x28BF (10384 to 10431 decimal). There is no standard structure to COMM – it is free to be defined by user programs, and may include data that is shared between the Arduino sketch and the J1 code, or private data structures manipulated by the J1. However, 48 bytes is not a lot of space for data structures. Unused sprite bitmap memory in the 16-kbyte RAM_SPRIMG block is a good place to locate large data structures that don't fit into COMM.

Table 43: COMM Memory Map

Symbol		Address		Length		Contents	Default
		(hex)	(dec)	(hex)	(dec)		
COMM	RW	0x2890	10384	0x30	48	48 bytes shared memory	0x00

Coprocessor-Only Registers

A few Gameduino registers are only available to J1 programs – they cannot be read or written by the host microcontroller via the SPI interface. Because they are so tightly tied to the J1 coprocessor, these registers can also be read and written via a 16-bit data path, so that load and store instructions write entire words (rather than writing just one byte, as is the case with loads or stores to ordinary Gameduino registers).

Current Raster Line: This is a read-only counter that increments as the screen display is generated. The register is incremented immediately after the last pixel of the line is composited, so changes in YLINE can be used to detect when it is safe to modify on-screen objects and control registers without causing visual glitches. J1 code has at least 45 cycles long, and up to 1677 cycles, before composition of the next raster line begins. The time taken depends on the number of sprites that must be composited onto the line: the minimum time of 45 cycles corresponds to the maximum of 96 sprites per line. Raster line 0 is the top-most visible line of the display, and line 299 is the bottom-most visible line. Waiting for YLINE>299 will detect the start of vertical blanking. However, YLINE is undefined during vertical blanking, and may take on unpredictable values, so YLINE data can be used to estimate the amount of time left in the vertical blanking interval.

Table 44: YLINE Register

Symbol		Address		Length		Register Contents
		(hex)	(dec)	(hex)	(dec)	
YLINE	R	0x8000	32768	0x2	2	X X X X X X X $Y_8Y_7Y_6Y_5Y_4Y_3Y_2Y_1Y_0$

Y_0–Y_8: Current raster line, 0x0000 to 0x14C (0 to 332).

FPGA ICAP Port: A set of 6 registers that allow J1 programs to directly access the internal configuration access port (ICAP) of the Xilinx field-programmable gate array (FPGA) used to implement the Gameduino. The ICAP port is not normally used for game programming, and use of these registers could alter the Gameduino's operation. Consult the Xilinx User Guide for details on how to use this port. The manual can be found on the Xilinx site at:
http://www.xilinx.com/support/documentation/user_guides/ug332.pdf

Table 45: FPGA ICAP Registers

Symbol		Address		Length		Register Contents
		(hex)	(dec)	(hex)	(dec)	
ICAP_O	R	0x8002	32770	0x2	2	X X X X X X X X $O_7O_6O_5O_4O_3O_2O_1O_0$
ICAP	W	0x8006	32774	0x2	2	X X X X X W_0 E_0 C_0 $I_7I_6I_5I_4I_3I_2I_1I_0$

O_0–O_7: ICAP output byte, 0x00-0xFF (0 to 255 decimal).
I_0–I_7: ICAP input byte, 0x00-0xFF (0 to 255 decimal).
W_0: ICAP write enable, 0=write and 1=read.
E_0: ICAP select (chip enable): 0=enable and 1=disable.
C_0: ICAP interface clock.

Programmable Frequency Generator: A programmable frequency generator is available to the J1 coprocessor using the FREQHZ and FREQTICK registers. The FREQHZ register sets the desired frequency, from 1Hz (0x0001) to 65,535Hz (0xFFFF); 8000Hz (0x1F40) is the default. The FREQTICK register increments by one at the specified frequency, allowing the J1 to count at a precise rate. There is no way to set or reset FREQTICK, it merely counts up at the specified rate. When FREQTICK reaches 0xFF, it rolls over to 0x00 the next time it increments. Setting FREQHZ to 0x0000 programs 0Hz and stops incrementing FREQTICK.

Table 46: FREQHZ and FREQTICK Registers

Symbol		Address		Length		Register Contents	Default
		(hex)	(dec)	(hex)	(dec)		
FREQHZ	W	0x800A	32778	0x2	2	Frequency setting, 1Hz to 65535Hz	0x1F40
FREQTICK	R	0x800C	32780	0x1	1	Counter increments at CLOCKHZ	0x00

Pin 2 Control: Two coprocessor-only registers, in combination with the shard IOMODE register, control pin 2 of the Gameduino's Arduino hardware interface. The shared IOMODE register is used to assign pin 2 to one of two functions, or to disable it entirely. When IOMODE is set to 0x4A (decimal 74 or ASCII "J"), the pin is controlled by the coprocessor-only P2_V and P2_DIR registers.

Table 47: P2 I/O Registers

Symbol		Address		Length		Register Contents
		(hex)	(dec)	(hex)	(dec)	
P2_V	RW	0x800E	32782	0x2	2	X X X X X X X X X X X X X X X V_0
P2_DIR	R	0x8010	32784	0x2	2	X X X X X X X X X X X X X X X D_0

V_0: Arduino interface pin 2 value, 0=low and 1=high.
D_0: Pin 2 direction, 0=output, 1=input.

When used as an output, J1 code running on the Gameduino can signal the host microcontroller on pin 2 of the interface. This is a digital pin on Arduino boards, and it can be used as an interrupt. J1 code on the Gameduino can use this interrupt to trigger Arduino code that must run synchronously with the Gamduino state. For example, the J1 can trigger an interrupt when vertical blanking is detected, and the Ardiuno's interrupt service routine can update on-screen objects without causing visual glitches in the video output.

When used as an input, pin 2 can either be connected directly to a digital signal that the J1 code monitors, or it can be used as an alternate signaling path between the host microcontroller and the Gameduino. The J1 processor doesn't support interrupts, so code that monitors pin 2 will need to poll the state of the pin to detect signals.

Random Number: The J1 can access a high-quality random number generator via the RANDOM register. The value in this read-only register is based on the Gameduino's built-in white noise generator. The hardware continuously updates this value so that every time it is read, the register produces a new random value.

Table 48: RANDOM Register

Symbol		Address		Length		Register Contents
		(hex)	(dec)	(hex)	(dec)	
`RANDOM`	R	`0x8012`	`32786`	`0x2`	2	16-bit random number

Clock: A clock cycle counter is available in the CLOCK register. This register counts the number of 50MHz J1 clock cycles since the Gameduino was reset. When the count reaches 0xFFFF (65535 decimal) it wraps around to 0x0000.

Table 49: CLOCK Register

Symbol		Address		Length		Register Contents
		(hex)	(dec)	(hex)	(dec)	
`CLOCK`	R	`0x8014`	`32788`	`0x2`	2	Clock cycle count, 0x0000 to 0xFFFF

SPI Flash Memory: When the shared IOMODE register is set to 0x46 (decimal 70 or ASCII "F"), the J1 CPU can use the FLASH registers to access the Gameduino's onboard flash memory. By directly manipulating a four-line bus, the J1 can initiate SPI transactions to read or write memory locations under program control.
This memory stores configuration data that is loaded into the Xilinx FPGA at boot-up time; changing data stored in this memory could permanently alter the functioning of the Gameduino.

Table 50: FLASH SPI Registers

Symbol		Address		Length		Register Contents
		(hex)	(dec)	(hex)	(dec)	
`FLASH_MISO`	R	`0x8016`	`32790`	`0x2`	2	X X X X X X X X X X X X X X X O_0
`FLASH_MOSI`	W	`0x8018`	`32792`	`0x2`	2	X X X X X X X X X X X X X X X I_0
`FLASH_SCK`	W	`0x801A`	`32794`	`0x2`	2	X X X X X X X X X X X X X X X C_0
`FLASH_SSEL`	W	`0x801C`	`32796`	`0x2`	2	X X X X X X X X X X X X X X X S_0

O_0: SPI MISO (master-in/slave-out) line input, 0=low and 1=high.
I_0: SPI MOSI (master-out/slave-in) signal to output, 0=low and 1=high.
C_0: SPI SCK (serial clock) signal to output, 0=low and 1=high.
S_0: SPI SSEL (slave select, also SS or SEL) signal to output, 0=low and 1=high.

J1 Architecture and Programming

The J1 is a stack-based microprocessor optimized to execute Forth code; in fact its instruction set includes 21 instructions that directly implement ANS Forth words.

The logical architecture of the processor is shown in **Error! Reference source not found.**. It is designed to execute Forth-like stack-based languages, and the most prominent features of the J1 are its two stacks: a data stack that is used by ALU operations, and a return stack that is primarily used to hold subroutine return addresses. Each stack can hold up to 32 entries, each entry containing a 16-bit word. A few stack entries have specific names, that are used to describe J1 operations: the top of the data stack is T (top of stack), while the next item on the data stack is N (next on stack); the top of the return stack is R (return address).

The basic stack operations are push and pop.

- **Push:** Pushing a value onto a stack saves it for later use. When words are pushed onto a stack, the new data becomes the top of the stack, and each element of the stack moves down. For example, when pushing a value onto the data stack, the value pushed becomes the new T, while the previous value of T becomes the new N, and so on.
- **Pop:** Popping an element from the stack is the exact reverse of a push. Popping a value removes it from the stack, and the rest of the elements move up. Values used in a computation are typically popped prior to computing the result.

The J1 stacks are finite, and can only store 32 elements, but the J1 does not check for overflow or underflow. If a stack already contains 32 elements, pushing a new element on the stack will not cause an exception; instead, the bottom-most element is discarded to make room. Popping a value from an empty stack will not cause an exception; the value of unused stack elements is undefined and may be random. Similarly, performing two-operand instructions on a data stack containing only one value will not result in an error, but the results of the computation are undefined, and will likely vary depending on the previous contents of the stack.

The J1 includes an ALU that can perform 16 logical operations, including addition, multiplication, comparison and bitwise logic operations. Inputs to the ALU are the values stored in T and N (for binary operators), and may be popped from the stack after use. Results are pushed back on the data stack. Instructions are also available to load and store the data stack from memory, and transfer values between the data stack and the return stack. All operations are performed on 16-bit words, for a computational range of 0 to 65535 (0x0000 to 0xFFFF). Signed comparison operators are available, allowing signed 16-bit integers in the range -32767 to 32767 to be represented. Boolean FALSE is represented as 0x0000, and while any non-zero value is TRUE, 0x0001 is used by convention.

Although the J1 architecture logically supports a 16-bit program counter (PC), the Gameduino implementation is limited to fetching instructions from the 256 bytes of RAM at J1_CODE. J1 instructions are always 2-byte words, so in effect the least-significant bit and the most significant byte of the J1 PC are hard coded to 0 and 0x2B (decimal 43) respectively. The J1 implements a conditional branch on T=0, an unconditional branch, as well as an unconditional subroutine call. Subroutine call instructions push the program counter onto the return stack. One unique feature of the J1 is that subroutine returns typically do not require an instruction, because any ALU instruction can be coded to also perform a subroutine return. Together with some of the J1's other features, this enables surprisingly compact code to be created.

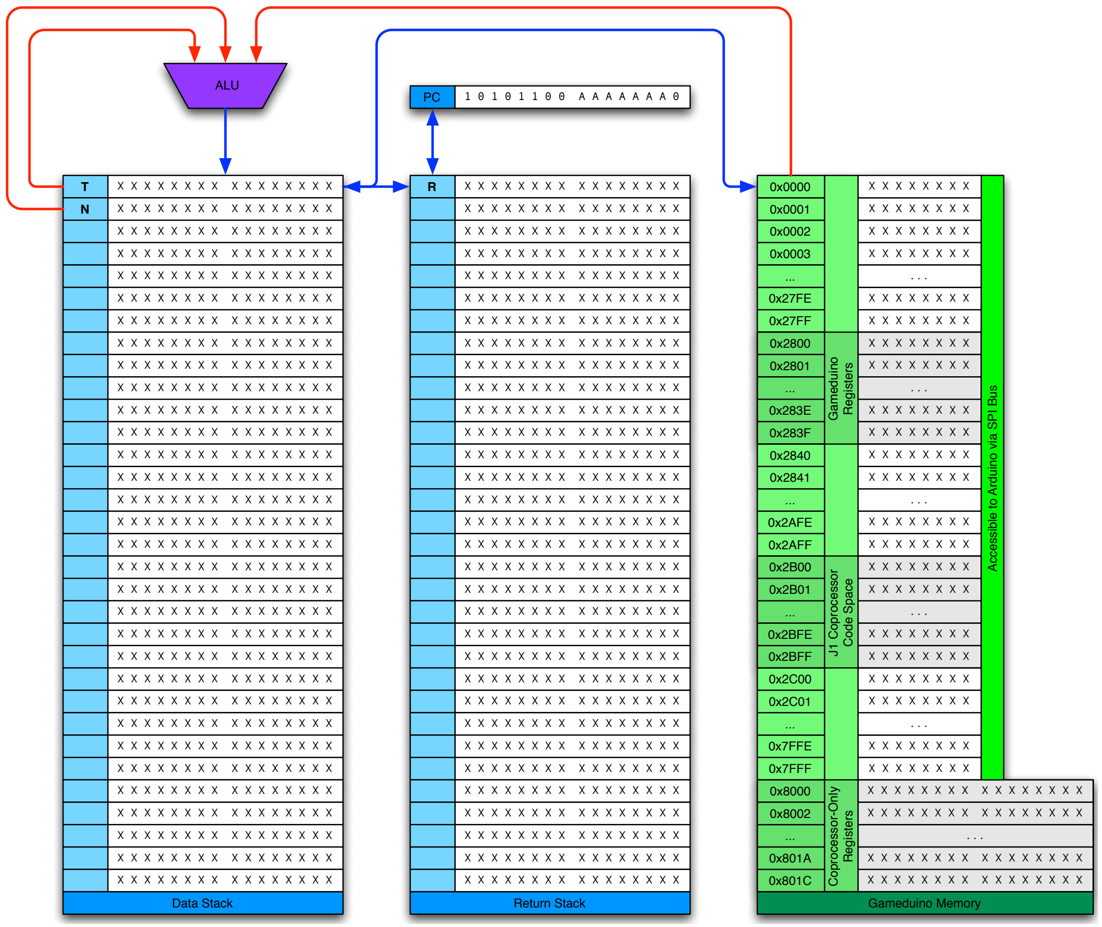

Figure 4: J1 Logical Architecture

Because the Gameduino's address space is byte-oriented, load and store operations only transfer one byte of data, despite the fact that the J1 is a 16-bit CPU. Values loaded from the Gameduino memory occupy the low 8 bits of a J1 word while the upper bits are set to zero. Writes from the J1 to Gameduino address space ignore the high byte, and store only the low byte. The only exception to this is when accessing the coprocessor-only registers, which are 16 bits wide; load and store operations to these registers transfer an entire word.

Sketch 5: J1 Assembly Program Template

```
start-microcode example
/ subroutine definitions go here
: main
  / main program code goes here
  begin again
;
end-microcode
```

Forth-Based Assembly Language

The standard way of programming the J1 is to use its Forth-based assembly language. An assembler is provided as part of the

Gameduino coprocessor software development kit. A valid J1 assembly program must contain a subroutine named "main", which will be the primary entry point of the program. This routine must not return to the caller – it should either loop indefinitely itself, or end with an empty infinite loop ("begin again"). Optionally, the program may contain additional subroutines before main. The shell of a J1 assembly language program is presented as Sketch 5: J1 Assembly Program Template. A summary of the J1 assembly language is provided as Table 51: J1 Assembly Language Summary.

Table 51: J1 Assembly Language Summary

Instruction	Pops	Pushes	Function
Assembler Directives			
`start-microprogram` *symbol*	---	---	
`end-microprogram`	---	---	
nnnnn `constant` *symbol*	---	---	Defines symbol
Literal Instructions			
`d#` *nnnnn*	---	nnnnn	Pushes constant nnnn
`h#` *hhhh*	---	hhhh	Pushes constant hhhh
`[char]`*c*	---	c	Pushes constant c
symbol	---	value	Pushes value of symbol
ANS Forth Instructions			
`+`	N, T	N+T	Add
`1-`	T	T-1	Decrement
`=`	N, T	N=T	Bitwise equality
`<`	N, T	N<T	Unsigned less-than
`u<`	N, T	N<T	Signed less-than
`xor`	N, T	N⊕T	Bitwise exclusive or
`and`	N, T	N∧T	Bitwise and
`or`	N, T	N∨T	Bitwise or
`invert`	T	~T	Bitwise negation
`swap`	N, T	T, N	Swaps T and N
`dup`	---	T	Duplicates T
`drop`	T	---	Discards T
`over`	---	N	Pushes a copy of N
`nip`	N	---	Discards N (preserving T)
`>r`	T	R	Moves T to R
`r>`	R	T	Moves R to T
`r@`	---	R	Pushes a copy of R
`rshift`	N, T	N>>T	Logical right shift
`*`	N, T	N × T	Multiply
Additional Instructions			
`swab`	T	T'	Swap bytes within T
`noop`	---	---	No operation
Multi-Word ANS Forth Instructions			
`c@`	T	[T]	Load byte at memory address T
`c!`	N, T	---	Store N to memory address T

Instruction	Pops	Pushes	Function
Merged Instructions			
`2dup+`	---	N+T	over over +
`over+`	T	N+T	over +
`2dup=`	---	N=T	over over =
`over=`	T	N=T	over =
`2dup<`	---	N<T	over over <
`over>`	T	N>T	over >
`2dupu<`	---	N<T	over over u<
`overu>`	T	N>T	over u>
`2dupxor`	---	N⊕T	over over xor
`overxor`	T	N⊕T	over xor
`2dupand`	---	N∧T	over over and
`overand`	T	N∧T	over and
`2dupor`	---	N∨T	over over or
`overor`	T	N∨T	over or
`dup>r`	---	R	dup r>
`2duprshift`	---	N>>T	over over rshift
`2dup*`	---	N × T	over over *
`over*`	T	N × T	over *
`dupswab`	---	T’	dup swab
`dupc@`	---	[T]	dup c@
Flow Control			
`:` *`symbol`*	---	---	Define subroutine
`;`	R	---	Return, pop R to PC
`;fallthru`	---	---	End subroutine without return
`symbol`		R	Call subroutine, push PC to R
`if`	T	---	Conditional branch
`else`	---	---	Conditional branch separator
`then`	---	---	Conditional branch terminator
`begin`	---	--	Begin loop
`again`	---	---	Return to “begin”, infinite loop
`until`	T	---	Bottom-exit, loop when false
`while`	T	---	Middle-exit, loop when true
`repeat`	---	---	Return to “begin”

Each subroutine begins with a definition, includes multiple instructions, and ends with a return. A subroutine definition is a colon, followed by the name of the subroutine being defined. Forth words within each subroutine are assembled into J1 instructions, and are executed in the order that they are encountered. Due to the J1’s stack-oriented architecture, arithmetic is executed in postfix fashion: programs must first push the appropriate values onto the data stack prior to coding arithmetic or logic instructions that will operate on them Finally, the subroutine ends with a semicolon.

Using the J1 Assembler

The J1 assembler is not integrated into the Arduino IDE. Instead, a separate J1 assembler, written in the gforth programming language, is used to assemble J1 code into binary format. One of the outputs of the J1 assembler is a header file that can be included into Arduino sketches. The header includes the assembled J1 code as an array of bytes stored in program memory.

<<<to-do: insert directions for assembling programs>>>

J1 Assembly Language Reference

J1 instructions include literals, which push values onto the data stack, ANS Forth instructions and additional instructions that perform arithmetic, logical and data operations, and flow-control statements. These instructions are described in the tables below.

Assembler Directives: The assembler supports three directives: start- and end-directives for marking the beginning and end of J1 assembly code, plus an assembler constant facility. The directives are summarized in Table 52: J1 Assembler Directives, below.

Table 52: J1 Assembler Directives

Instruction	Explanation
`start-microprogram` *`symbol`*	**<u>Start Program</u>** **Function:** Marks the start of a J1 microprogram for the assembler. **Syntax:** The start-microprogram directive <u>must</u> be followed by a valid alphanumeric symbol. All microprograms should contain a "main" subroutine containing one or more instructions, and close with the "end-microprogram" directive. Microprograms may optionally contain additional subroutine definitions. **Example:** `start-microprogram example1`
`end-microprogram`	**<u>End Program</u>** **Function:** Marks the end of a J1 microprogram to the assembler. **Syntax:** None. **Example:** `end-microprogram`

Instruction	Explanation
`nnnnn constant symbol`	**<u>Define Constant</u>** **Function:** Defines an assembler constant named *symbol* with the value *nnnnn*. **Syntax:** The "constant" directive <u>must</u> be preceded by a valid decimal integer, *nnnnn*, in the range 0 to 65535, or a preprocesor definition that evaluates to a decimal integer. It <u>must</u> be followed by a valid symbol, the name of the constant. **Example:** `10386 constant fillchar` Defines an assembler constant named "fillchar" with the value 10386 (0x2892).

Literal Instructions: As shown in Table 53: J1 Literal Instructions, a variety of literal values can be pushed onto the J1's stack. Literals can include 16-bit values, and are assembled into one or two instructions. A single instruction is used for values of 0 through 32767 (0x000 through 0x7FFF), while two instructions are needed for values of 32768 through 65535 (0x8000 through 0xFFFF).

Table 53: J1 Literal Instructions

Instruction	Explanation
`d# nnnnn`	**<u>Decimal Number:</u> nnnnn → T** **Function:** Pushes a decimal literal onto the data stack. **Syntax:** The "d#" directive <u>must</u> be followed by a valid decimal integer, *nnnnn*, in the range 0 to 65535. **Note:** Assembles to a single instruction for values in the range 0 to 32767; two instructions and two J1 CPU cycles are required for values in the range 32768 to 65535. **Example:** `d# 128` 0x0080 → T
`h# hhhh`	**<u>Hexadecimal Number:</u> 0xhhhh → T** **Function:** Pushes a hexadecimal literal onto the data stack. **Syntax:** The "h#" directive <u>must</u> be followed by a valid hexadecimal integer, *hhhh*, in the range 0x0000 to 0xFFFF. **Note:** Assembles to a single instruction for values in the range 0x0000 to 0x7FFF; two instructions and two J1 CPU cycles are required for values in the range 0x8000 to 0xFFFF. **Example:** `h# 2B80` 0x2B80 → T
`[char]c`	**<u>Character:</u> c → T** **Function:** Pushes a one-byte character literal, c, onto the data stack. **Syntax:** The "[char]" directive <u>must</u> be followed by a single one-byte character, *c*. **Example:** `[char] Z` 0x005A → T

Instruction	Explanation
`symbol`	**<u>Insert Constant</u>** **Function:** Inserts the named assembler constant as a literal; functionally equivalent to the "d#" directive above. **Syntax:** "*symbol"* is the name of any defined assembler constant. An assembler error is returned if the name is invalid or undefined. **Example:** `fillchar` 0x2892 → T

ANS Forth Instructions: The following Forth primitives are assembled as single instructions. For the examples, note that literal instructions are used to load the data stack, so T=0x0029 and if needed N=0x0003 (decimal 41 and 3 respectively). Assume that Gameduino memory locations in RAM_PIC (including both 0x0029 and 0x0003) contain an ASCII blanks (0x20, decimal 32).

Table 54: J1 ANS Forth Instructions

Instruction	Explanation
`+`	**<u>Plus:</u> N + T → T** **Function:** Pops two values (N and T) from the data stack and pushes the sum N+T onto the data stack, reducing the net stack depth by one. **Syntax:** The "+" operator requires two operands on the data stack. Arithmetic overflow does not cause an exception; instead, the carry bit (most significant bit) of the result is discarded. **Example:** `d# 3 h# 0029 +` N=0x0003 + T=0x0029 = 0x002C → T
`1-`	**<u>Decrement:</u> --T → T** **Function:** Subtracts one from T; does not change the stack depth. **Syntax:** The "1-" operator requires one operand on the data stack. Arithmetic underflow does not cause an exception; decrementing 0x0000 results in 0xFFFF. **Example:** `h# 0029 1-` T=0x0029 – 1 = 0x0028 → T
`=`	**<u>Equals:</u> N == T → T** **Function:** Pops two values (N and T) from the data stack and pushes a Boolean flag onto the stack, reducing the net stack depth by one. The flag is true (0x0001) if T and N are bit-for-bit identical, or false (0x0000) otherwise. **Syntax:** The "=" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 =` N=0x0003 == T=0x0029 = 0x0000 → T
`<`	**<u>Less Than:</u> N < T → T** **Function:** Pops two values (N and T) from the data stack and pushes a Boolean flag onto the stack, reducing the net stack depth by one. The flag is true (0x0001) if N < T when N and T are treated as <u>signed two's complement</u> 16-bit integers, or false (0x0000) otherwise. **Syntax:** The "<" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 <` N=0x0003 < T=0x0029 = 0x0001 → T

Instruction	Explanation
`u<`	**Unsigned Less Than: N < T → T** **Function:** Pops two values (N and T) from the data stack and pushes a Boolean flag onto the stack, reducing the net stack depth by one. The flag is true (0x0001) if N < T when N and T are treated as unsigned 16-bit integers, or false (0x0000) otherwise. **Syntax:** The "u<" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 u<` N=0x0003 < T=0x0029 = 0x0001 → T
`xor`	**Exclusive Or: N ⊕ T → T** **Function:** Pops two values (N and T) from the data stack and pushes the result of a bit-by-bit exclusive-or between N and T onto the stack, reducing the net stack depth by one. **Syntax:** The "xor" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 xor` N=0x0003 ⊕ T=0x0029 = 0x002A → T
`and`	**And: N ∧ T → T** **Function:** Pops two values (N and T) from the data stack and pushes the result of a bit-by-bit and between N and T onto the stack, reducing the net stack depth by one. **Syntax:** The "and" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 and` N=0x0003 ∧ T=0x0029 = 0x0001 → T
`or`	**Or: N ∨ T → T** **Function:** Pops two values (N and T) from the data stack and pushes the result of a bit-by-bit inclusive-or between N and T onto the stack, reducing the net stack depth by one. **Syntax:** The "or" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 or` N=0x0003 ∨ T=0x0029 = 0x002B → T
`invert`	**Invert: ~T → T** **Function:** Performs a bitwise inversion of T; all one bits become zero and vice-versa. Does not change the stack depth. **Syntax:** The "invert" operator requires one operand on the data stack. **Example:** `h# 0029 invert` ~ T=0x0029 = 0xFFD6 → T
`swap`	**Swap: N ⇔ T** **Function:** Exchanges the values of N and T on the data stack. Does not change the stack depth. **Syntax:** The "swap" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 swap` N=0x0003 T=0x0029 → N=0x0029 T=0x0003
`dup`	**Duplicate: T → [++dstack]** **Function:** Pushes the value of T onto the stack, effectively duplicating T; increases the stack depth by one. **Syntax:** The "dup" operator requires one operand on the data stack. **Example:** `h# 0029 dup` T=0x0029 → T

Instruction	Explanation
`drop`	**<u>Drop</u>** **Function:** Discards the top element of the data stack, decreasing the stack depth by one and promoting N to the new top of stack. **Syntax:** The "drop" operator requires one operand on the data stack. **Example:** `drop`
`over`	**<u>Over:</u> N → T** **Function:** Pushes a copy of N onto the data stack, increasing stack depth by one. **Syntax:** The "over" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 over` N=0x0003 → T
`nip`	**<u>Nip</u>** **Function:** Removes N from the data stack, preserving T and decreasing stack depth by one. **Syntax:** The "nip" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 nip`
`>r`	**<u>Put-rStack:</u> T → R** **Function:** Pops T from the data stack and pushes it onto the return stack, decreasing data stack depth by one and increasing return stack depth by one. **Syntax:** The ">r" operator requires one operand on the data stack. Since the return stack is also used to store subroutine return addresses, any items pushed onto the return stack must be removed before attempting to return via ";". **Example:** `h# 0029 >r` T=0x0029 → R
`r>`	**<u>rStack-Get:</u> R → T** **Function:** Pops R from the return stack and pushes it onto the data stack, decreasing return stack depth by one and increasing data stack depth by one. **Syntax:** The "r>" operator requires one operand on the return stack. User programs must take care not to disturb existing return addresses when manipulating the return stack. **Example:** `r>` R=0x0029 → T
`r@`	**<u>rStack-Fetch:</u> R → [T** **Function:** Copies R from the return stack and pushes it onto the data stack, increasing data stack depth by one. Return stack depth is not affected. **Syntax:** The "r@" operator requires one operand on the return stack. **Example:** `r@` R=0x0029 → T

Instruction	Explanation
`rshift`	**Right Shift: N >> T → T** **Function:** Pops two values (N and T) from the data stack, logically shifts N to the right by T bit places, inserting zeros into the most significant places vacated by the shift. The result is pushed onto the data stack; "rshift" reduces the net stack depth by one. **Syntax:** The "rshift" operator requires two operands on the data stack. The result of the operation is undefined if T >= 16. **Example:** `h# 1A3C h# 0003 rshift` N=0x1A3C >> T=0x0003 = 0x0347 → T
`*`	**Multiply: N × T → T** **Function:** Pops two values (N and T) from the data stack and pushes the product T×N onto the data stack, reducing the net stack depth by one. **Syntax:** The "*" operator requires two operands on the data stack. Arithmetic overflow does not cause an exception; instead, the carry bit (most significant bit) of the result is discarded. **Example:** `d# 3 h# 0029 *` N=0x0003 × T=0x0029 = 0x007B → T

Additional Instructions: In addition to the ANS Forth instructions, the J1 coprocessor implements two instructions that are not present in the standard Forth language, as described in Table 55: J1 Additional Instructions below. These operations are also implemented as single instructions on the Gameduino's J1 CPU:

Table 55: J1 Additional Instructions

Instruction	Explanation
`swab`	**Swap Bytes: T' → T** **Function:** Swaps the high and low bytes of T. Does not change the stack depth. **Syntax:** The "swab" operator requires one operand on the data stack. **Example:** `h# 0029 swab` T=0x0029 swab = 0x2900 → T
`noop`	**No Operation** **Function:** Performs no operation, does not change any stack. **Syntax:** None. **Example:** `noop`

Multi-word Instructions: In addition to some literal instructions, a few additional ANS Forth base words require multiple J1 instructions for implementation, as described in Table 56: J1 Multi-Word Instructions. In particular, memory access timing requires that memory access take two J1 machine cycles to complete, so Forth words that require memory access are assembled into two machine instructions to provide sufficient time.

Table 56: J1 Multi-Word Instructions

Instruction	Explanation

Instruction	Explanation
`c@`	<u>**Character Fetch:**</u> **[T] → T** **Function:** Fetches an 8-bit byte or a 16-bit word from memory, depending on context: • If T is between 0x0000 and 0x7FFF (0 to 32767 decimal), reads the byte at memory address T and pads the upper 8 bits with zeros. • If T is an even number between 0x8000 and 0x801C (32768 to 32796 decimal), reads the 16-bit word at memory address. In either case, the top of the data stack is replaced with the value read; net data stack depth is not affected. **Syntax:** The "c@" operator requires one operand on the data stack. The operation of "c@" is undefined when T contains an odd number between 0x8001 and 0x801D, or when T contains a value greater than 0x801C. **Note:** Assembled as "noop c@" to ensure correct memory access timing. **Example:** `h# 0x0029 c@` [T=0x0029]=0x0020 → T
`c!`	<u>**Character Store:**</u> **N → [T]** **Function:** Writes an 8-bit or a 16-bit value to memory, depending on context: • If T is between 0x0000 and 0x7FFF (0 to 32767 decimal), writes the low (least-significant) byte of N to the memory address T. • If T is an even number between 0x8000 and 0x801C (32768 to 32796 decimal), writes all 16 bits of N to the word at memory address T and T+1. In either case, "c!" pops two values (N and T) from the data stack, reducing data stack depth by two. **Syntax:** The "c!" operator requires two operands on the data stack. The operation of "c!" is undefined when T contains an odd number between 0x8001 and 0x801D. Data is discarded without warning if the operation attempts to write data to non-existent or read-only memory addresses. **Note:** Assembles as two instructions (one that writes to memory but only removes N from the stack, followed by drop) to ensure correct memory access timing. **Example:** `d# 3 h# 0x0029 c!` N=0x0003 → [T=0x0029]

Merged Instructions: The J1 CPU can execute a number of single instructions that perform the equivalent of two ANS Forth primitives in sequence. In each case, the merged instruction is named for its components: for example, executing "overand" is the same as executing the code fragment "over and". Merged instructions represent a significant opportunity to save code size and execution time by taking advantage of features unique to the J1 CPU architecture. Table 57: J1 Merged Instructions describes these instructions.

Table 57: J1 Merged Instructions

Instruction	Explanation

Instruction	Explanation
`2dup+`	<u>**2dup Plus:**</u> **N + T → T** **Function:** Pushes the sum N+T onto the data stack, increasing the net stack depth by one. **Syntax:** The "2dup+" operator requires two operands on the data stack. Arithmetic overflow does not cause an exception; instead, the carry bit (most significant bit) of the result is discarded. **Example:** `d# 3 h# 0029 2dup+` N=0x0003 + T=0x0029 = 0x002C → T
`over+`	<u>**2dup Plus:**</u> **N + T → T** **Function:** Pops one value (T) from the data stack, and pushes the sum N+T onto the data stack, leaving the net stack depth unchanged. **Syntax:** The "over+" operator requires two operands on the data stack. Arithmetic overflow does not cause an exception; instead, the carry bit (most significant bit) of the result is discarded. **Example:** `d# 3 h# 0029 over+` N=0x0003 + T=0x0029 = 0x002C → T
`2dup=`	<u>**2Dup Equals:**</u> **N == T → T** **Function:** Pushes a Boolean flag onto the data stack, increasing the net stack depth by one. The flag is true (0x0001) if T and N are bit-for-bit identical, or false (0x0000) otherwise. **Syntax:** The "2dup=" operator requires two operands on the data stack. Stack underflow does not raise an exception; results of a computation involving an underflow are undefined. **Example:** `d# 3 h# 0x0029 2dup=` N=0x0003 == T=0x0029 = 0x0000 → T
`over=`	<u>**Over Equals:**</u> **N == T → T** **Function:** Pops one value (T) from the data stack, and pushes a Boolean flag onto the data stack, leaving the net stack depth unchanged. The flag is true (0x0001) if T and N are bit-for-bit identical, or false (0x0000) otherwise. **Syntax:** The "over=" operator requires two operands on the data stack. Stack underflow does not raise an exception; results of a computation involving an underflow are undefined. **Example:** `d# 3 h# 0x0029 over=` N=0x0003 == T=0x0029 = 0x0000 → T
`2dup<`	<u>**2dup Less Than:**</u> **N < T → T** **Function:** Pushes a Boolean flag onto the data stack, increasing the net stack depth by one. The flag is true (0x0001) if N < T when N and T are treated as <u>signed two's complement</u> 16-bit integers, or false (0x0000) otherwise. **Syntax:** The "2dup<" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 2dup<` N=0x0003 < T=0x0029 = 0x0001 → T

Instruction	Explanation
`over>`	**<u>2dup Greater Than:</u> N > T → T** **Function:** Pops one value (T) from the data stack and pushes a Boolean flag onto the stack, leaving the net stack depth unchanged. The flag is true (0x0001) if N > T when N and T are treated as <u>signed two's complement</u> 16-bit integers, or false (0x0000) otherwise. **Syntax:** The "over>" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 over>` N=0x0003 > T=0x0029 = 0x0000 → T
`2dupu<`	**<u>2dup Unsigned Less Than:</u> N < T → T** **Function:** Pushes a Boolean flag onto the data stack, increasing the net stack depth by one. The flag is true (0x0001) if N < T when N and T are treated as <u>unsigned</u> 16-bit integers, or false (0x0000) otherwise. **Syntax:** The "2dupu<" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 2dupu<` N=0x0003 < T=0x0029 = 0x0001 → T
`overu>`	**<u>Over Unsigned Greater Than:</u> N > T → T** **Function:** Pops one value (T) from the data stack, and pushes a Boolean flag onto the stack, leaving the net stack depth by unchanged. The flag is true (0x0001) if N > T when N and T are treated as <u>unsigned</u> 16-bit integers, or false (0x0000) otherwise. **Syntax:** The "overu>" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 overu>` N=0x0003 > T=0x0029 = 0x0000 → T
`2dupxor`	**<u>2Dup Exclusive Or:</u> N ⊕ T → T** **Function:** Pushes the result of a bit-by-bit exclusive-or between N and T onto the data stack, increasing the net stack depth by one. **Syntax:** The "2dupxor" operator requires two operands on the data stack. Stack underflow does not raise an exception; results of a computation involving an underflow are undefined. **Example:** `d# 3 h# 0x0029 2dupxor` N=0x0003 ⊕ T=0x0029 = 0x002A → T
`overxor`	**<u>Over Exclusive Or:</u> N ⊕ T → T** **Function:** Pops one value (T) from the data stack, and pushes the result of a bit-by-bit exclusive-or between N and T onto the data stack, leaving the net stack depth unchanged. **Syntax:** The "overxor" operator requires two operands on the data stack. Stack underflow does not raise an exception; results of a computation involving an underflow are undefined. **Example:** `d# 3 h# 0x0029 overxor` N=0x0003 ⊕ T=0x0029 = 0x002A → T
`2dupand`	**<u>2dup And:</u> N ∧ T → T** **Function:** Pushes the result of a bit-by-bit and between N and T onto the stack, increasing the net stack depth by one. **Syntax:** The "2dupand" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 2dupand` N=0x0003 ∧ T=0x0029 = 0x0001 → T

Instruction	Explanation
`overand`	**Over And: N ∧ T → T** **Function:** Pops one value (T) from the data stack and pushes the result of a bit-by-bit and between N and T onto the data stack, leaving the net stack depth unchanged. **Syntax:** The "overand" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 overand` N=0x0003 ∧ T=0x0029 = 0x0001 → T
`2dupor`	**2dup Or: N ∨ T → T** **Function:** Pushes the result of a bit-by-bit inclusive-or between N and T onto the data stack, increasing the net stack depth by one. **Syntax:** The "2dupor" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 2dupor` N=0x0003 ∨ T=0x0029 = 0x002B → T
`overor`	**Over Or: N ∨ T → T** **Function:** Pops one values (T) from the data stack and pushes the result of a bit-by-bit inclusive-or between N and T onto the stack, leaving the net stack depth unchanged. **Syntax:** The "overor" operator requires two operands on the data stack. **Example:** `d# 3 h# 0029 overor` N=0x0003 ∨ T=0x0029 = 0x002B → T
`dup>r`	**Dup Put-rStack: T → R** **Function:** Copies T from the data stack to the return stack, increasing return stack depth by one but leaving the data stack depth unchanged. **Syntax:** The "dup>r" operator requires one operand on the data stack. Since the return stack is also used to store subroutine return addresses, any items pushed onto the return stack must be removed before attempting to return via ";". **Example:** `h# 0x0029 >r` T=0x0029 → R
`2duprshift`	**2dup Right Shift: N >> T → T** **Function:** Logically shifts N to the right by T bit places, inserting zeros into the most significant places vacated by the shift. The result is pushed onto the data stack; "2duprshift" increases the net stack depth by one. **Syntax:** The "2duprshift" operator requires two operands on the data stack. The result of the operation is undefined if T >= 16. **Example:** `h# 1A3C h# 0003 2duprshift` N=0x1A3C >> T=0x0003 = 0x0347 → T
`2dup*`	**2dup Multiply: N × T → T** **Function:** Pushes the product T×N onto the data stack, increasing the net stack depth by one. **Syntax:** The "2dup*" operator requires two operands on the data stack. Arithmetic overflow does not cause an exception; instead, the carry bit (most significant bit) of the result is discarded. **Example:** `d# 3 h# 0029 2dup*` N=0x0003 × T=0x0029 = 0x007B → T

Instruction	Explanation
`over*`	**<u>Over Multiply:</u> N × T → T** **Function:** Pops one value (T) from the data stack and pushes the product T×N onto the data stack, leaving the net stack depth unchanged. **Syntax:** The "over*" operator requires two operands on the data stack. Arithmetic overflow does not cause an exception; instead, the carry bit (most significant bit) of the result is discarded. **Example:** `d# 3 h# 0029 over*` N=0x0003 × T=0x0029 = 0x007B → T
`dupswab`	**<u>Dup Swap Bytes:</u> T' → T** **Function:** Swaps the high and low bytes of T and pushes the result onto the data stack, increasing the stack depth by one. **Syntax:** The "swab" operator requires one operand on the data stack. **Example:** `h# 0029 dupswab` T=0x0029 swab = 0x2900 → T
`dupc@`	**<u>Dup Character Fetch:</u> [T] → T** **Function:** Fetches an 8-bit byte or a 16-bit word from memory, depending on context, and pushes it onto the data stack: • If T is between 0x0000 and 0x7FFF (0 to 32767 decimal), reads the byte at memory address T and pads the upper 8 bits with zeros. • If T is an even number between 0x8000 and 0x801C (32768 to 32796 decimal), reads the 16-bit word at memory address T. In either case, the resulting value is pushed onto the data stack, increasing the net stack depth by one. **Syntax:** The "dupc@" operator requires one operand on the data stack. The operation of "dupc@" is undefined when T contains an odd number between 0x8001 and 0x801D, or when T contains a value greater than 0x801C. **Note:** Assembled as "dup c@" to ensure correct memory access timing. **Example:** `h# 0x0029 dupc@` [T=0x0029]=0x0020 → T

Flow Control: The assembler also recognizes a number of that used to determine how preceding and subsequent instructions should be assembled, or that implement flow control such as branching and subroutines. Most of the directives will assemble into one J1 instruction, typically a branch or call. Some directives, including ":", "then", and "begin" mark branch or call addresses instead. Subroutine returns via ";" do not require a separate instruction in most cases.

Table 58: J1 Assembler Flow Control

Instruction	Explanation

Instruction	Explanation
`: symbol`	**<u>Define Subroutine</u>** **Function:** Defines a subroutine entry point named "symbol". Code following the subroutine name up to the next semicolon will be assembled into a subroutine and called when the symbol is invoked. **Syntax:** The ":" directive must be followed by a valid symbol (the name of the subroutine) and zero or more instructions. Subroutine definitions may not be nested: every subroutine must be closed with ";" or ";fallthru" directive before another subroutine can be defined. **Example:** `: > swap < ;` Defines a subroutine entry point named ">" that will execute "swap <" whenever it is called.
`;`	**<u>Return from Subroutine</u>: R → PC** **Function:** Completes the definition of a subroutine and assembles a return instruction: the top-most value of the return stack is removed and loaded into the program counter. Reduces the return stack depth by one; the data stack depth is unaffected. **Syntax:** The ";" directive must be preceded by a valid subroutine definition consisting of the ":" directive, a symbol, and zero or more instructions. **Note:** The J1 architecture often allows a "free" return: ALU instructions can be coded to also return from a subroutine, making subroutines particularly efficient at reducing the overall code size of a program. **Example:** `;` R → PC
`;fallthru`	**<u>End Subroutine without Return</u>** **Function:** Completes the definition of a subroutine, but <u>does not</u> assemble a return instruction. This allows subroutines with multiple entry points. Execution continues with the next instruction following the ";fallthru" directive. **Syntax:** The ";fallthru" directive <u>must</u> be preceded by a valid subroutine definition beginning with the ":" directive, a symbol, and zero or more executable instructions. It <u>must</u> be followed by another subroutine definition. **Example:** `: 0> d# 0 ;fallthru : > swap < ;` Defines a subroutine with two entry points, "0>" and ">". When "0>" is called, the instruction stream "d# 0 swap <" is executed; when ">" is called, only "swap <" is executed.
`symbol`	**<u>Call Subroutine</u>: PC → R; *symbol* → *PC*** **Function:** Pushes the location of the following instruction onto the return stack and calls the named subroutine. Return stack depth increases by one. **Syntax:** The name of any defined subroutine, separated by white space. An assembler error is returned if the name is invalid or if a subroutine with the given name can't be found. **Example:** `0>` Calls the subroutine defined as an example above; the instruction stream "d# 0 swap <" is executed.

Instruction	Explanation
`if`	**<u>Conditional Branch</u>** **Function:** Pops T from the data stack and treats it as a Boolean flag. When the flag is false (0x0000), the code between "else" and "then" is executed. Otherwise, the code between "if" and "else" is executed. In either case, execution resumes with the instruction following "then". "If" reduces the data stack depth by one. **Syntax:** A conditional branch begins with "if", may optionally contain "else", and must close with "then"; each is separated by zero or more words. The execution path is undefined if the stack is empty. Conditional branches may be nested to an unspecified depth. **Example:** `2dup < if nip else drop then` Compares T and N, and removes the smaller value from the stack preserving the larger as the new top of stack: max(N,T) → T.
`else`	**<u>Conditional Branch Separator</u>** **Function:** Separates the two alternatives code paths in a conditional branch. If there is no code between "else" and "then", the "else" may also be omitted. See "if" above. **Syntax:** A conditional branch begins with "if", may optionally contain "else", and must close with "then"; each is separated by zero or more words. Conditional branches may be nested to an unspecified depth. **Example:** `2dup < if nip else drop then` Compares T and N, and removes the smaller value from the stack preserving the larger as the new top of stack: max(N,T) → T.
`then`	**<u>Conditional Branch Terminator</u>** **Function:** Ends a conditional branch; normal execution resumes with the instruction following "then". See if above. **Syntax:** A conditional branch begins with "if", may optionally contain "else", and must close with "then"; each is separated by zero or more executable instructions. Conditional branches may be nested to an unspecified depth. **Example:** `2dup < if nip else drop then` Compares T and N, and removes the smaller value from the stack preserving the larger as the new top of stack: max(N,T) → T.
`begin`	**<u>Begin Loop</u>** **Function:** Starts a looping construct; after reaching the bottom of the loop, execution will resume with the instruction following "begin". Does not change data or return stack depth. **Syntax:** A loop may take one of three forms: • An infinite loop consisting of "begin" followed by zero or more words and completed by "again", • A bottom-exit loop consisting of "begin" followed by zero or more words ending with "until", or • A middle-exit loop consisting of "begin", "while", and "repeat", with zero or more words between each. Loops may be nested to an unspecified depth. **Example:** `begin d# 1 + again` Loops forever, adding 1 to the top of the stack.

Instruction	Explanation
`again`	**<u>Infinite Loop</u>** **Function:** Returns execution to the instruction following the preceding "begin". Any code following "again" is never executed. Does not affect data or return stack depth. **Syntax:** An infinite loop consists of "begin" followed by zero or more words and terminated by "again". Loops may be nested to an unspecified depth: a "begin" … "again" loop may itself contain any number and types of loop construct. **Example:** `begin d# 1 + again` Loops forever, adding 1 to the top of the stack.
`until`	**<u>Bottom-Exit Loop</u>** **Function:** Completes a bottom-exit loop: pops T from the data stack and treats it as a Boolean flag. If false (0x0000), returns execution to the instruction following the preceding "begin". Otherwise, continues execution with the instruction following "until". Reduces data stack depth by one. **Syntax:** A bottom-exit loop consists of "begin" followed by zero or more words, ending with "until". Loops may be nested to an unspecified depth: a "begin" … "until" loop may itself contain any number and types of loop constructs. **Example:** `d# 45 d# 1 begin d# 1 + 2dup= until` Count up from 1 to 45 by adding one to the top of stack until the resulting value equals 45.
`while`	**<u>Middle-Exit Loop Exit</u>** **Function:** Pops T from the data stack and treats it as a Boolean flag. If false (0x0000), exits the loop by returning execution to the instruction following the subsequent "repeat". Otherwise, execution continues with the instruction after "while". Reduces data stack depth by one. **Syntax:** A middle-exit loop consists of "begin", "while", and "repeat", with zero or more executable instructions between each. Loops may be nested to an unspecified depth: a "begin" … "while" … "repeat" loop may contain any number and types of loop constructs. **Example:** `d# 0 d# 4 begin 1- 2dup= while foo repeat` Counts down from 4 to 0, executing the "foo" subroutine four times.
`repeat`	**<u>Middle-Exit Loop Boundary</u>** **Function:** Returns execution to the instruction following the matching "begin". **Syntax:** A middle-exit loop consists of "begin", "while", and "repeat", with zero or more executable instructions between each. Loops may be nested to an unspecified depth: a "begin" … "while" … "repeat" loop may contain any number and types of loop constructs. **Example**: `d# 0 d# 4 begin 1- 2dup= while foo repeat` Counts down from 4 to 0, executing the "foo" subroutine four times.

Useful Subroutines: The following subroutines are useful in a wide variety of programs, either because they define additional ANS Forth words, or perform frequently-used code sequences.

<<<to-do: insert a table of useful subroutines>>>

Code Optimization

Although the J1 coprocessor only has 256 bytes (128 instructions) of program memory, it also has a number of features that a clever programmer can use to write efficient code that occupies as little space as possible. These techniques include:

1. **Factor common code into subroutines:** A J1 subroutine call is a single instruction, and the return is almost always free in terms of code space and execution time, since a return from subroutine can be coded into any ALU instruction. This means that even short sequences of instructions can be made into subroutines to save space. Factor out:
   - Sequences of 2 instructions that are used 3 or more times in your code, and
   - Sequences of 3 or more instructions that are used at least twice in your code.

   For example, if the subroutine in Sketch 6: Example Subroutine replaces 5 instances where the three-instruction sequence "d# 2 * 2dup=" would be used in your code, the net savings is 7 instructions (14 bytes).

   Sketch 6: Example Subroutine

   ```
   : 2*2dup= d# 2 * 2dup= ;
   ```

2. **Use multiple entry subroutines:** The J1 assembler allows subroutines to have multiple entry points. This allows the clever programmer to factor more code into subroutines. After factoring repeated instruction sequences into subroutines, look for instances where the same instruction or sequence of instructions precedes a subroutine call multiple times in your code. Factor these out into an additional subroutine entry point preceding the original subroutine using ;fallthru.

   For example, after factoring your code into subroutines as described above, you find 3 instances where your code reads "+ 2*2dup=", and two other instances that are unique. You can then define a multiple-entry subroutine using ;fallthrou as in Sketch 7: Example Multiple-Entry Subroutine to save an additional 3 instructions (6 bytes) – a total of 10 instructions (20 bytes) savings.

   Sketch 7: Example Multiple-Entry Subroutine

   ```
   : +2*2dup= + ;fallthru
   : 2*2dup= d# 2 * 2dup= ;
   ```

3. **Use merged instructions:** The J1 Forth instruction set includes two merged instructions, 2dup= and 2dupxor, that merge the function of the Forth 2dup operation with quality or logical exclusive-or (respectively). In the J1, these merged instructions implement common sequences of operations into a single instruction. These two instructions are particularly useful for programming loops, because they leave both of their arguments on the stack after execution, leaving the stack properly set up for another iteration of the loop.

If you are programming directly in J1 microcode rather than assembler, many additional merged instructions are possible by manipulating the J1 stack pointers: any ALU instruction that removes values from either stack can be coded to retain the operands on the stack instead. This effectively merges the instruction with dup or 2dup (for one-operand and two-operand instructions, respectively), saving instruction space. Possible merged instructions include 2dup+, dup1-, 2dup<, dupu<, 2dupand, 2dupor, dupinvert, 2duprshift, and 2dup*.

4. **Code to facilitate free returns:** Subroutines should end with ALU instructions. ALU instructions are those in Table 54: J1 ANS Forth Instructions or Table 55: J1 Additional Instructions. This allows the return to be coded directly into the last instruction in the subroutine. If this isn't possible, ending a subroutine with a call to a different subroutine will allow the assembler to replace the call with a jump, also resulting in a free return. Avoid ending a subroutine with an instruction from Table 53: J1 Literal Instructions, since these can't be converted into free returns.

5. **Use one-instruction literals:** When possible, avoid literal values greater than 0x7FFF (32,767 decimal). Literals between 0x0000 and 0x7FFFF can be assembled into a single load instruction. Literals 0x8000 (decimal 32,768) must be assembled into two instructions, a literal followed by invert.

6. **Count down instead of up**: The J1 instruction set includes a single instruction decrement (1-), but does not include an increment instruction. Incrementing a counter typically takes two instructions (d# 1 +). It often saves an instruction to decrement through a loop working from a higher memory address to a lower one or to decrement a loop counter towards zero, rather than incrementing from lower addresses to higher ones or incrementing a counter towards a limit. For example, Sketch 8: Incrementing Loop and Sketch 9: Decrementing Loop both perform the same function (a logical "and" of sprite image memory with 0xCF to clear one of the sets of 4-color sprite images). However, the decrement version is four bytes (two instructions) shorter: the use of the single-instruction decrement (1-) instead of the two instruction literal and add (d# 1 +) saves one, and the elimination of the two-instruction literal (0x8000) in favor of a one-instruction value (0x7FFF) saves another.

Sketch 9: Decrementing Loop

```
h# cf >r
h# 8000 h# 4000
begin
  dup c@ r@ and
  over c!
  d# 1 + 2dup=
until
```

Sketch 8: Incrementing Loop

```
h# cf >r
h# 3FFF h# 7FFF
begin
  dup c@ r@ and
  over c!
  1- 2dup=
until
```

J1 Microcode

The J1 machine language is powerful, and contains a large number of instructions that are not directly implemented in the assembler language. The hardware architecture of the J1 differs somewhat from the logical architecture described above.

<<<to-do: insert diagram of J1 hardware architecture>>>

J1 microcode is bit-field coded, as described in the following tables. There are 5 overall types of instructions: literals, conditional and unconditional jumps, a subroutine call, and ALU instructions:

- **Literal**: Loads a literal value V into T, increments the data stack pointer, and copies the previous value of T to this location as the new value of N: V → T, T → [++dstack].
- **Unconditional Jump**: Loads a new value N into the program counter. The old value is discarded, and neither stack is modified. Because J1 instructions are 16-bit words, N must be an even number.
- **Conditional Jump**: Removes T from the stack by decrementing dstack. If T=0, jumps to N by loading it into the program counter. The old value PC value is discarded, and the return stack is not modified. Because J1 instructions are 16-bit words, N must be an even number
- **Subroutine Call**: Loads a new value N into the program counter. The old value is pushed onto the return stack, increasing the return stack depth by one. The data stack is not modified. Because J1 instructions are 16-bit words, N must be an even number.
- **ALU Operation**: ALU operations are multi-purpose instructions that can include a return from a subroutine, store the result of a computation onto the data stack, copy T into N or R, write to memory, and adjust the data and return stack pointers – or potentially all of these operations in a single instruction.

Coding of J1 microinstructions is described in Table 59: J1 Microcode Instructions. The bit fields for ALU operations are further explained in the additional tables below.

Table 59: J1 Microcode Instructions

Instruction	Format
Literal	1 $V_{14}V_{13}V_{12}V_{11}V_{10}V_{09}V_{08}V_{07}V_{06}V_{05}V_{04}V_{03}V_{02}V_{01}V_{00}$
Unconditional Jump	0 0 0 0 1 1 0 0 N_7 N_6 N_5 N_4 N_3 N_2 N_1 N_0
Jump on T=0	0 0 1 0 1 1 0 0 N_7 N_6 N_5 N_4 N_3 N_2 N_1 N_0
Subroutine Call	0 1 0 0 1 1 0 0 N_7 N_6 N_5 N_4 N_3 N_2 N_1 N_0
ALU Operation	0 1 1 P O_3 O_2 O_1 O_0 S A W X D_1 D_0 R_1 R_0

V_{00}–V_{14}: Immediate value to load, 0x0000 to 0x7FFF (0 to 32767 decimal).
X: Unused bit, set to 0 for compatibility
N_0–N_7: Target address for jump or call, 0x00 to 0xFE (0 to 254 decimal); must be even.
P: Pop program counter from return stack, [rstack--]→PC; 0=no and 1=yes.

O_0–O_3: ALU operation to execute, see Table 60: ALU Operation Codes.
S: Copy T→N , 0=no and 1=yes.
A: Copy T→R, 0=no and 1=yes.
W: Write N→[T], 0=no and 1=yes.
D_0–D_1: Data stack pointer (dstack) adjustment, see Table 61: Data Stack Adjustment.
R_0–R_1: Return stack pointer (rstack) adjustment, see Table 62: Return Stack Adjustment.

Table 60: ALU Operation Codes

ALU Output	O_0-O_3
T	`0x0`
N	`0x1`
N + T	`0x2`
N ∧ T	`0x3`
N ∨ T	`0x4`
N ⊕ T	`0x5`
~T	`0x6`
N==T	`0x7`
N < T	`0x8`
N >> T	`0x9`
T – 1	`0xA`
R	`0xB`
[T]	`0xC`
N × T	`0xD`
swab T	`0xE`
N < T	`0xF`

Table 61: Data Stack Adjustment

dstack	D_0-D_1
No adjustment	`0 0`
dstack + 1	`0 1`
Undefined	`1 0`
dstack – 1	`1 1`

Table 62: Return Stack Adjustment

rstack	D_0-D_1
No adjustment	`0 0`
rstack + 1	`0 1`
rstack – 2	`1 0`
rstack – 1	`1 1`

<<<to-do: better description of J1 microprogramming!>>>

Screenshot Feature

The Gameduino hardware composes each raster line in a memory buffer before outputting it to the screen. The screenshot feature makes this buffer available at the SCREENSHOT memory locations, 0x2C00-0x2F1F (11264-12063 decimal) so that the host microcontroller can read it via the SPI interface.

Screenshot Line Select Register

To capture a row of video data, load SCREENSHOT_Y with a value between 0x8000 and 0x812B (decimal 32768 and 33067), corresponding to the flag bit F=1 and the S_0-S_8 equal to the line number 0x000-0x12B (0 to 299 decimal) to be captured. Writing a 0 disables the screenshot feature. The low bits of the SCREENSHOT_Y register always reflect the current raster line being generated. When the flag bit of the register reads 1, the selected line of data is available in memory locations 0x2C00-0x2F1F (11264-12063 decimal).

Table 63: SCREENSHOT_Y Register

Symbol		Address		Length		Byte Offset and Register Contents		Default
		(hex)	(dec)	(hex)	(dec)	`+0`	`+1`	

SCREENSHOT_Y	RW	0x2406	10246	0x2	2	$S_7S_6S_5S_4S_3S_2S_1S_0$	F X X X X X X S_8	0x0000

X: Unused bit position, ignored; should be 0 for compatibility reasons.
S_0–S_8: Raster line to capture, 0x000-0x12B (0-299 decimal).
F: Screenshot flag, 0=disabled or not ready, or 1=enabled or data ready.

Screenshot Line Buffer

The screenshot line buffer, SCREENSHOT, contains color information for 400 pixels and occupies 800 bytes beginning at address 0x2C00 (11264 decimal). Every pixel is composed of a 2-byte Gameduino color value. Because pixels output to the video hardware cannot be transparent, the alpha channel data is always zero.

Table 64: SCREENSHOT Memory Map

Pixel		Address		Length		Byte Offset and Register Contents	
		(hex)	(dec)	(hex)	(dec)	+0	+1
000	R	0x2C00	11264	0x2	2	$G_2G_1G_0B_4B_3B_2B_1B_0$	X $R_4R_3R_2R_1R_0G_4G_3$
001	R	0x2C01	11265	0x2	2	$G_2G_1G_0B_4B_3B_2B_1B_0$	X $R_4R_3R_2R_1R_0G_4G_3$
398	R	0x2F1E	12062	0x2	2	$G_2G_1G_0B_4B_3B_2B_1B_0$	X $R_4R_3R_2R_1R_0G_4G_3$
399	R	0x2F1F	12063	0x2	2	$G_2G_1G_0B_4B_3B_2B_1B_0$	X $R_4R_3R_2R_1R_0G_4G_3$

X: Alpha channel color information, always 0.
R_4–R_0: Red channel color information, 0-32.
B_4–B_0: Green channel color information, 0-32.
B_4–B_0: Blue channel color information, 0-32.

Glossary

Acronyms

MISO: Master In/Slave Out, the SPI line used to send data from the Gameduino to the host microcontroller. The Gameduino uses Arduino pin 11 for MISO. See SPI.

MOSI: Master Out/Slave In, the SPI line used to send data from the host microcontroller to the Gameduino. Arduino pin 12 is MOSI on the Gameduino board. See SPI.

SCK: Serial Clock, the SPI line used to send clock pulses that synchronize data transfers. The Gameduino uses Arduino pin 13 for SCK. See SPI.

SEL: Select, the SPI pin used to select the target device for SPI communication. By default, the Gameduino uses Arduino digital pin 9 for SEL. See SPI.

SPI: Serial Peripheral Interface, a synchronous serial data link protocol. The SPI interface uses 4 digital lines: SEL, MISO, MOSI, and SCK. See MISO, MOSI, SCK, and SEL.

SS: Slave select, an alternate term for SEL. See SEL.

Time

ms, millisecond: One thousandth (1×10^{-3}) of a second.
µs, microsecond: One millionth (1×10^{-6}) of a second.
ns, nanosecond: One billionth (1×10^{-6}) of a second.
1ms = 1000µs = 1,000,000ns