# REDSPARC

# User Manual

**Version 1.0 (Alpha 2)**

**z5t1.com/redsparc**

## Contents

# Setting up the Redsparc

## *Downloading the Files*

Before we can use the Redsparc, we must set it up. **This is not done the same way you set up other custom maps.**

The Redsparc actually runs on a Bukkit server. It is, by design, dependent on custom Bukkit plugins to function. So to use the Redsparc, you must first setup a Bukkit server. Don't worry, this isn't as difficult as it sounds.

Head over to http://z5t1.com/redsparc/download to download the Redsparc. Once you've downloaded the zip file, extract it. The next step varies depending on what operating system you're using, so follow the directions in the appropriate section for your system.

## *Starting the Server*

### Windows

You've got it pretty easy:

1. Double click start.bat
2. When prompted, click run

I promise this is not a virus, but if you don't trust me, you can look at it for yourself. Right click on start.bat and click edit.

### Mac OS X

1. Open up a terminal
2. Navigate to the directory you unzipped the Redsparc files to using the cd command.
3. Type ./start.sh

**Note:** If typing ./start.sh doesn't work, you may first need to type chmod 775 start.sh

### Linux

If you use Ubuntu or any other distribution that uses Gnome, then you can do the following:

1. Double click start.sh
2. When prompted, click run in terminal

If the above doesn't work or your distribution doesn't use Gnome, do the following:

1. Open up a terminal

2. Navigate to the directory you unzipped the Redsparc files to using the cd command.
3. Type ./start.sh

**Note:** If typing ./start.sh doesn't work, you may first need to type chmod 775 start.sh

### *Connecting*

Now your server is up, running and waiting for you to connect. Launch Minecraft and connect to server 127.0.0.1 to start using the Redsparc.

## Running Programs

### *Running a Program*

Before you run any program, make sure to press the hardware reset button. Then, type the following command in game:

```
/load <filename>
```

Where <filename> is the File Name of the program you want to run. Once you have loaded a program, press the execute button and the Redsparc will begin executing the program. Remember to be patient if the program doesn't appear to be working at some point. Often, you will have to wait a few seconds before getting a response from the computer.

**Note:** Before you run any program, you should always make sure you have reset the computer first. To do this, press `power off` and then press `hardware reset`.

### *Using the Preinstalled Programs*

The Redsparc comes with some programs already installed for you to use. They are as follows:

| **Program** | **File Name** |
|---|---|
| Exponents – Displays the exponents of a number | exponents |
| Guess the Number – A simple guess the number game | gtn |
| Calculator – A four function calculator program | calculator |

### Exponents

This is a very simple program. To run it, type `/load exponents` and then press the execute button. The programs works as follows:

1. The ? symbol turns on indicating that the program is ready for you to enter input.

2. Input the starting number using the Numpad.
3. The numeric display will display $n^1$, then $n^2$ and so on until you press the power off button.

## Guess the Number

This is the classic guess the number game. To run it, type `/load gtn` and then press the execute button. The programs works as follows:

1. The program will generate a random number between 1 and 25 inclusive.
2. The ? symbol will turn on, indicating that the computer's ready for you to guess.
3. Enter a guess using the Numpad.
4. Your guess will be displayed on the numeric display.
5. The up arrow will turn on if you are too low, the down arrow will turn on if you're too high and the ! symbol will turn on if you've guessed the number.
6. If you didn't get the number, guess again.
7. If you got the number, how many guesses it took you will be displayed on the numeric display.

## Calculator

This is a four function calculator. It does addition, subtraction, multiplication and division. To run it, type `/load calculator` and then press the execute button. The programs works as follows:

1. You enter the first number on the Numpad.
2. You enter an operation on the Gamepad:
   - A for addition
   - B for subtraction
   - X for multiplication
   - Y for division
3. You enter the second number on the Numpad.
4. The numeric display will display the result.

## *Installing Programs*

You can also install additional programs for the Redsparc. To view a list of additional programs for the Redsparc, head over to http://z5t1.com/redsparc/programs.

Once you've found a program you want to install, download the **binary version** of it. You have to copy the file you downloaded to the proper location. Go to wherever you extracted the files for the Redsparc, click on `plugins` and then click on `Load`. You must copy the file you downloaded into this directory. You can then run it in game by pressing the hardware reset button, typing `/load <filename>` and pressing the execute button.

## Input & Output Devices

The Redsparc has three input devices: the Numpad, Gamepad and keyboard as well as three output devices: the numeric display, symbol display and GPU (or screen). This section documents how these devices work in general. For documentation on their use for a specific program, please see the documentation for that program.

### *Numpad*

The Numpad is used for inputting numeric values. It may look complicated, but it really isn't. There are ten columns and five rows. Each row represents a different digit of the number. The top row is the 10,000s place, row two is the 1,000s place and so on. Each column is a different value for that digit. Simply press the buttons as desired, and then press the enter button.

For example, to enter the number 62,574, do the following:

1. Press the 6 button on the top row
2. Press the 2 button on the second row
3. Press the 5 button on the third row
4. Press the 7 button on the fourth row
5. Press the 4 button on the bottom row
6. Press the enter button

### *Gamepad*

The Gamepad is the second of the input devices. It is used not only in games, but generally for getting any input that is not numeric or text. It has 12 buttons: back, start, ok, stop, up, down, left, right, A, B, X and Y. All you have to do with this device is press a button.

### *Keyboard*

The last of the input devices is the keyboard. It is usually used for getting text input, but is also sometimes used for other purposes. It has a layout very similar to the layout used on real keyboards. Simply step on the pressure plates for the key you wish to press.

### *Numeric Display*

The numeric display is capable of displaying numbers between 0 and 99,999 inclusive. Not much else to say here.

### *Symbol Display*

The symbol display consists of 6 different symbols: ^, v, <, >, ? and !. These symbols are used by programs to indicate their status or tell you some information other than a number.

### *GPU (Screen)*

The GPU, or display screen, is capable of displaying images, text and just about anything else.

---

# Variablez

### *A Note for the not so Tech Savvy*

This section contains information for people that want to know all the nitty gritty details of how the Redsparc works. If you are not one of those people, you need not (and probably should not) read this section.

### *An Introduction to Variablez*

As mentioned earlier, the Redsparc is, by design, heavily dependent on a custom Bukkit plugin: Variablez. This plugin does what its name implies; it allows you to store values in variables. It functions very similarly to how scoreboards do, but adds a couple of essential features.

While some may not appreciate this and some may even consider it cheating, there is a very valid reason for this. Making a computer in Minecraft using just vanilla content and no custom commands hasn't ever worked out too well. There have been computers made this way, but they are all extremely slow (not one of them has a clock speed faster than 0.5 Hz) and extremely limited with memory, not to mention that it isn't feasible to make a Von Neumann computer this way.

So if you can just put any predrawn conclusions aside, you may actually come to appreciate how much more powerful the Redsparc is because of this decision.

## Setting Values

All Variablez commands start with `/var`, followed by a subcommand. In its most basic form, it works as follows:

```
/var set <variable> <value>
```

Sets a variable equal to a value, where variable is the name of the variable and value is the value you want to set it to. The variable name may consist of uppercase and lowercase letters, numbers and underscores. The value must be an integer between -2147483648 and 2147483647 (or if you prefer, any signed 32 bit integer). So typing `/var set foo 42` would set variable foo equal to 42.

## Printing Values

Printing a value isn't that different from setting it. Simply type:

```
/var print <constant>
```

This prints the value specified as constant. So typing `/var print 42` would, in turn, print 42. To print a variable, we must use a constant expression.

## Constant Expressions

Many Variablez commands take a constant expression as one or more of the arguments. These are not as complicated as they may seem. A constant expression consists of a constant value. So you could use 42 as the constant, or any other value. It is also possible, however, to use variables in constant expressions. We simply put the variable name inside of [ ]. This tells Variablez that we want to use the **value** of that variable, not just the **name**.

So for example, to print the value of the foo variable, we would type `/var print [foo]`. This would cause `42` to be printed.

You can also prepend a constant to a variable as follows: `/var print foo=[foo]`. This would cause `foo=42` to be printed instead of just `42` as in the previous example.

Finally, you can nest braces. This allows you, in a sense, to create arrays of values. Say that we wanted to get the RAM value at index foo. We would just simply type `/var print [ram[foo]]`. This would print RAM address 42. This is essential to the Redsparc, as without this feature you would not be able to use registers as RAM indices.

**Note:** You should only use a constant expression when an argument specifically says it takes a constant expression. If an argument just takes a plain variable name, you do not surround the variable name in braces. You can, however, use braces for retrieving an index from an array, so `/var set ram[foo] 66` would be perfectly valid.

## Incrementing and Decrementing

Variablez allows for simple incrementing and decrementing of variables. Simply use `/var inc <variable>` and `/var dec <variable>` respectively. Variable should **not** be a constant expression.

## Mathematical and Bitwise Operations

Variablez also lets you perform mathematical and bitwise operations. The syntax is as follows:

```
/var <operation> <a> <b>
```

Where operation is a valid operation (see table below), a is a variable name (**not** a constant expression) and b is a constant expression. This performs the operation on a and b, storing the result in variable a.

**Note:** For operations where the order of operands matters (subtraction, division and modulus), a should be the first operand and b should be the second.

**Valid Operations**

| Operation | Subcommand |
|---|---|
| Addition | /var add |
| Subtraction | /var sub |
| Multiplication | /var mul |
| Division | /var div |
| Modulus | /var mod |
| Bitwise AND | /var and |
| Bitwise OR | /var or |
| Bitwise XOR | /var xor |

## If Statements

Finally, Variablez allows for comparison via the if subcommand. It's a bit complicated and works as follows:

```
/var if <a> <?> <b> <world> <x> <y> <z>
```

First, a must be a variable name (**not** a constant expression), b must be a constant expression and ? must be a comparison operator (see table below). If the comparison is true, then the block at x, y, z in world will be set to a redstone block (ID 152). Otherwise, it will be set to air (ID 0). This allows for conditional branching, another feature essential to the Redsparc.

For example, `/var if foo = [bar] world 1 2 3` would set the block at 1, 2, 3 in world to a redstone block if and only if the value of foo equaled the value of bar. Otherwise, it sets it to air.

**Comparison Operators**

| Operator | Function |
|---|---|
| = | Equal to |
| != | Not equal to |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |

### *Redsparc Variable Assignment*

The Redsparc uses three different arrays of variables:

- The **reg** array – used for storing the register values (AX, BX and so on).
- The **ram** array – used for storing, you guessed it, the RAM values
- The **io** array – used for for storing the values of the serial io ports.

## The Registers Array – reg

The reg array stores the values of all the registers used by the Redsparc. All registers in the Redsparc are 32 bits in size. Each index of this array corresponds to a different register. For a complete list of which registers correspond to which indices, see the Register Address table under Reference Tables section.

Some of these registers are accessed directly when programming. These registers are all the general purpose registers plus the SP and BP registers. The general purpose registers serve no special purpose and can be used for anything. The stack pointer (SP) register points to the bottom of the stack and is used during all stack related operations (push, pop, call, ret). The base pointer (BP) register currently serves no special purpose, but this behavior should **not** be relied upon when writing code, as it is likely to change in future versions of the Redsparc.

The other registers should **not ever** be accessed directly when programming. These registers are by the processor to store internal variables, such as the address of the next operation and the comparison flags. Accessing these registers directly **will** cause unexpected behavior and system instability.

## The RAM Array – ram

The ram array stores the the RAM values. This one is pretty simple. It uses linear addressing, with the first RAM value being `ram[0]`, the second one being `ram[1]` and so on. Each RAM value is 32 bits in size (to correspond with the register size). When the execute button is pressed, the processor will start executing the instruction at `ram[0]`, using `ram[1]` as `x` and `ram[2]` as `y`. For details on how `x` and `y` are used, see the Op Codes table under the

Reference Tables section.

### The Input / Output array – io

The io array is used for storing serial port values. It is this array that is used to interface with the hardware. The first serial port is `io[0]`, the second `io[1]` and so on. Each port controls a different piece of hardware. For information on which ports control what, see the Serial Port Assignment table under the Reference Tables section.

## Interfacing with the Built in Hardware

### *A Note for the not so Tech Savvy*

This section contains information for people that want to know all the nitty gritty details of how the Redsparc works. If you are not one of those people, you need not (and probably should not) read this section.

### *Basic Hardware Input / Output*

All hardware on the Redsparc is controlled by serial ports (in the io array). Interfacing with most hardware is as simple writing or reading values to or from serial ports. This section will cover in detail how each IO device works.

To read values from a serial port, use the `in` instruction (Op code 3). To write values to a serial port, use one of the `out` instructions (Op codes 1 & 2).

### *Input Devices*

### Numpad

The numpad uses the following serial port: `0`

When the user presses enter on the numpad, serial port 0 is set to the value the user entered, which will be between 0 and 99,999 inclusive. If there is no input, serial port 0 will be set to 100,000.

### Gamepad

The gamepad uses the following serial port: `9`

When the user presses a button on the gamepad, serial port 9 is set to the corresponding value. When no button has been pressed, serial port 9 is set to 0. For a list of which values correspond to which button, see the Gamepad Key Values table under the Reference Tables section.

### Keyboard

The keyboard uses the following serial port: `10`

When the user presses a key on the keyboard, serial port 10 is set to the corresponding character value. When no key has been pressed, serial port 10 is set to 0. For a list of which values correspond to which characters, see the Character Map table under the Reference Tables section.

## *Output Devices*

### Numeric Display

The numeric display uses the following serial port: `1`

The numeric display simply displays whatever value is in serial port 1; you need only change the value to change the display. Note however, that if the number is greater than 99,999 only the last 5 digits will be displayed.

### Symbol Display

The symbol display uses the following serial ports: `2 3 4 5 6 7`

Each of the symbol display serial ports controls a different symbol. To turn a symbol on, set the port to 1. To turn a symbol off, set the port to 0.

| **Port** | **Name** | **Symbol** |
|---|---|---|
| 2 | Up Arrow | ^ |
| 3 | Down Arrow | v |
| 4 | Left Arrow | < |
| 5 | Right Arrow | > |
| 6 | Question Mark | ? |
| 7 | Exclamation Point | ! |

## *GPU*

The GPU uses the following serial ports: `16 17 18 19 20 21 22 23 24 25`

The GPU is by far the most complex of all the input / output devices and probably deserves a manual of its own (which clearly isn't happening). The GPU also uses 10 serial ports, which is more than every other output device combined. Each of these ports serves a special purpose which is explained quite nicely by the following table:

| Serial Port | Name | Description |
|---|---|---|
| 16 | Execute | You set this to 1 when the GPU should execute a command. |
| 17 | Command | The op code for the command the GPU should execute (see GPU Op Codes under Reference Tables for a complete list of commands). |
| 18 | X1 | Stores the X1 coordinate. |
| 19 | Y1 | Stores the Y1 coordinate. |
| 20 | X2 | Stores the X2 coordinate. |
| 21 | Y2 | Stores the Y2 coordinate. |
| 22 | Color | Stores the current working color. |
| 23 | Buffer | Stores the current working buffer. |
| 24 | Register | Used as an extra argument for certain operations. |
| 25 | Status | Set to 1 if the GPU is busy, otherwise set to 0. |

## Colors

The Redsparc GPU supports all 16 Minecraft wool colors. For all operations where the GPU is drawing something, it uses the color specified by serial port 22. To change the color, simply write the appropriate color code to serial port 22.

**Wool Colors**

| Color Code | Color |
|---|---|
| 0 | White |
| 1 | Orange |
| 2 | Magenta |
| 3 | Light Blue |
| 4 | Yellow |
| 5 | Lime |
| 6 | Pink |
| 7 | Gray |

| Color Code | Color |
|---|---|
| 8 | Light Gray |
| 9 | Cyan |
| 10 | Purple |
| 11 | Blue |
| 12 | Brown |
| 13 | Green |
| 14 | Red |
| 15 | Black |

## Buffers

The Redsparc GPU also has multiple buffers. Traditionally, games use doubled buffer mode when rendering images. The Redsparc takes that a step further, having 17 buffers, numbered 0 - 16. The reason it has so many buffers is to allow software to create all the screens it needs at startup and then just swap them during execution (since drawing can be a bit slow), making the application seemingly run faster.

You are free to use buffers 0 - 15 for whatever your heart desires, however it is imperative that you do **not** modify buffer 16. Buffer 16 is used as the blank buffer and the GPU relies on it containing nothing other than black wool, as it is used as part of the reset process. However, since buffer 16 is guaranteed to be blank, you can use buffer 16 as the source buffer in the copy buffer command if you wish to clear another buffer.

Buffer 0 is the display buffer that the end user sees. Use the copy command (op code 1) to copy one buffer to another. The equivalent of doubled buffer mode would be using buffer 1 as your working buffer, and then copying buffer 1 to buffer 0 to “push” the image to the user.

## Executing Commands

GPU command execution works as follows: the op code is written to serial port 17 and the rest of the arguments are written to their corresponding serial ports. Once all of the arguments have been written, set serial port 16 to 1. This will make the GPU actually execute the command.

It's also worth mentioning that all serial ports (aside from port 25) retain their values across multiple operations. This means that if the color is the same across 2 or more operations, you need to write it to port 22 only once.

## GPU Error & Range Checking

It is important to take note of the fact that the GPU has no error checking or range checking on the values passed to it. This means that if you pass the GPU an X coordinate or a buffer that is outside of its bounds, the GPU will actually proceed to modify blocks in other parts of the map. This could potentially destroy the CPU or some other vital component of the computer, so don't do that.

# Troubleshooting

## The GPU (screen) / numeric display / symbol display isn't working

This can sometimes happen when you join the game or perform a hardware reset. Check the clock status lights (at the far left). If either of them are not blinking, press the clock reset button. If this is not the problem, try flying closer to the display.

## I pressed execute but nothing's happening

This can be caused by a wide variety of issues. Do the following steps **in order**:

1. Wait a few seconds. Often programs take a few seconds to start.
2. Make sure you've pressed the execute button.
3. Check the clock status lights. Ensure that they are blinking. If any one is not, press the clock reset button.
4. Press the hardware reset button, load the program again and press execute.
5. If all else fails, there could be a bug in the program you're trying to run.

# Reference Tables

## *Op Codes*

All op codes for the Redsparc have two arguments, `x` and `y`. For commands that don't use `x` and/or `y`, they should be set to `0`.

| Op Code | Hex | Syntax | Assembly | Explanation | Implemented |
|---|---|---|---|---|---|
| 0 | 0 | hlt | hlt | Halts computer operation | Redsparc |
| 1 | 1 | out x, y | out 4, 1 ;uses port 1 | Writes x to serial port y | Redsparc |
| 2 | 2 | out reg[x], y | out ax, 1 | Writes the value at register x to serial port y | Redsparc |
| 3 | 3 | in reg[x], y | in ax, 1 | Reads the value from serial port y into register x | Redsparc |
| | | | | | Redsparc |
| 8 | 8 | push reg[x] | push ax | Pushes register x onto the stack | Redsparc |
| 9 | 9 | pop reg[x] | pop ax | Pops register x from the stack | Redsparc |
| 10 | A | call x | call label | Calls x, saving the current instruction to the stack | Redsparc |
| 11 | B | ret | ret | Returns from a call, restoring the original instruction pointer | Redsparc |
| 12 | C | jmp x | jmp label | Jumps to location x | Redsparc |
| 16 | 10 | mov reg[x], y | mov ax, 4 | copies a constant into register x | Redsparc |
| 17 | 11 | mov reg[x], reg[y] | mov ax, bx | copies the contents of register y into register x | Redsparc |
| 18 | 12 | mov reg[x], ram[y] | mov ax, [32] | copies the contents of RAM location y into register x | Redsparc |
| 19 | 13 | mov reg[x], ram[reg[y]] | mov ax, [bx] | copies the contents of the RAM location pointed to by register y to register x | Redsparc |
| 20 | 14 | mov ram[x], y | mov [32], 4 | copies a constant into RAM location x | Redsparc |
| 21 | 15 | mov ram[x], reg[y] | mov [32], ax | copies the contents of register y into RAM location x | Redsparc |
| 22 | 16 | mov ram[x], ram[y] | mov [32], [33] | copies the contents of RAM location y into RAM location x | Redsparc |
| 23 | 17 | mov ram[reg[x]], reg[y] | mov [ax], bx | copies the contents of register y to the RAM location pointed to by register x | Redsparc |
| 32 | 20 | add reg[x], reg[y] | add ax, bx | adds the contents of registers x and y, saving the result to register x | Redsparc |
| 33 | 21 | sub reg[x], reg[y] | sub ax, bx | subtracts the contents of registers x and y, saving the result to register x | Redsparc |
| 34 | 22 | mul reg[x], reg[y] | mul ax, bx | multiplies the contents of registers x and y, saving the result to register x | Redsparc |
| 35 | 23 | div reg[x], reg[y] | div ax, bx | divides the contents of registers x and y, saving the result to register x | Redsparc |
| 36 | 24 | mod reg[x], reg[y] | mod ax, bx | modulus divides the contents of registers x and y, saving the result to register x | Redsparc |
| 37 | 25 | and reg[x], reg[y] | and ax, bx | ands the contents of registers x and y, saving the result to register x | Redsparc |
| 38 | 26 | or reg[x], reg[y] | or ax, bx | ors the contents of registers x and y, saving the result to register x | Redsparc |
| 39 | 27 | xor reg[x], reg[y] | xor ax, bx | xors the contents of registers x and y, saving the result to register x | Redsparc |
| 40 | 28 | inc reg[x] | inc ax | increments register x | Redsparc |
| 41 | 29 | dec reg[x] | dec ax | decrements register x | Redsparc |
| 48 | 30 | cmp reg[x], reg[y] | cmp ax, bx | Compares the contents or registers x and y, setting all comparison flags | Redsparc |
| 49 | 31 | je x | je label | Jumps to location x if the equal flag is set | Redsparc |
| 50 | 32 | jne x | jne label | Jumps to location x if the not equal flag is set | Redsparc |
| 51 | 33 | jg x | jg label | Jumps to location x if the greater than flag is set | Redsparc |
| 52 | 34 | jge x | jge label | Jumps to location x if the greater than or equal flag is set | Redsparc |
| 53 | 35 | jl x | jl label | Jumps to location x if the less than flag is set | Redsparc |
| 54 | 36 | jle x | jle label | Jumps to location x if the less than or equal flag is set | Redsparc |
| 55 | 37 | cmp reg[x], y | cmp ax, 4 | Compares the contents or register x and constant y, setting all comparison flags | Redsparc |
| 64 | 40 | add reg[x], y | add ax, 4 | adds the contents of register x and constant y, saving the result to register x | Redsparc |
| 65 | 41 | sub reg[x], y | sub ax, 4 | subtracts the contents of register x and constant y, saving the result to register x | Redsparc |
| 66 | 42 | mul reg[x], y | mul ax, 4 | multiplies the contents of register x and constant y, saving the result to register x | Redsparc |
| 67 | 43 | div reg[x], y | div ax, 4 | divides the contents of register x and constant y, saving the result to register x | Redsparc |
| 68 | 44 | mod reg[x], y | mod ax, 4 | modulos the contents of register x and constant y, saving the result to register x | Redsparc |
| 69 | 45 | and reg[x], y | and ax, 4 | ands the contents of register x and constant y, saving the result to register x | Redsparc |
| 70 | 46 | or reg[x], y | or ax, 4 | ors the contents of register x and constant y, saving the result to register x | Redsparc |
| 71 | 47 | xor reg[x], y | xor ax, 4 | xors the contents of register x and constant y, saving the result to register x | Redsparc |

## GPU Op Codes

| Op Code | Command | Description |
|---|---|---|
| 0 | reset | Resets the GPU |
| 1 | copy_buffer | Copies the working buffer to the buffer specified by GPU register (IO port 24) |
| 2 | set_pixel | Sets the pixel at X1, Y1 to the working color |
| 3 | fill | Fills the area from X1, Y1 to X2, Y2 with the working color |
| 4 | border | Sets the border for the area from X1, Y1 to X2, Y2 to the working color |

## Register Addresses

| Register | Value | Description |
|---|---|---|
| AX | 0x00 | General Purpose |
| BX | 0x01 | General Purpose |
| CX | 0x02 | General Purpose |
| DX | 0x03 | General Purpose |
| SI | 0x04 | General Purpose |
| DI | 0x05 | General Purpose |
| SP | 0x06 | Stack Pointer |
| BP | 0x07 | Base Pointer |
| IP | 0x10 | Instruction Pointer |
| I1 | 0x11 | Instruction Word 1 |
| I2 | 0x12 | Instruction Word 2 |
| I3 | 0x13 | Instruction Word 3 |
| MA | 0x14 | Memory Address |
| FEQ | 0x15 | Equal Flag |
| FNEQ | 0x16 | Not Equal Flag |
| FG | 0x17 | Greater Than Flag |
| FGE | 0x18 | Greater Than or Equal Flag |
| FL | 0x19 | Less Than Flag |
| FLE | 0x1A | Less Than or Equal Flag |
| I1P | 0x20 | Instruction 1 Pointer |
| I2P | 0x21 | Instruction 2 Pointer |
| I3P | 0x22 | Instruction 3 Pointer |

## GPU Register Addresses

| Register | Serial Port | Purpose |
|---|---|---|
|  | 16 | Set to 1 when the GPU should execute a command |
| GPU_CMD | 17 | Stores the current command |
| GPU_X1 | 18 | X1 coordinate |
| GPU_Y1 | 19 | Y1 coordinate |
| GPU_X2 | 20 | X2 coordinate |
| GPU_Y2 | 21 | Y2 coordinate |
| GPU_CLR | 22 | Current working color |
| GPU_BUF | 23 | Current working buffer |
| GPU_REG | 24 | Extra register for user interface |
|  | 25 | Stores the GPU status; 0 = free, 1 = busy |
| GPU_1 |  | GPU Internal Register 1 |

## ***Serial Port Assignment***

Key:

- Blue – ports already in use for built in hardware.
- Yellow – ports already in use for the GPU.
- Green – ports designated for use when developing your own hardware.
- Grey – ports reserved for future hardware.

Ports greater than 63 should not be used. They have not been allocated yet and may be used for built in hardware in future versions.

| Port | Direction | Use | Port | Direction | Use |
|---|---|---|---|---|---|
| 0 | In | Numpad Input | 32 | N/A | Available for User Developed Hardware |
| 1 | Out | Numeric Display Output | 33 | N/A | Available for User Developed Hardware |
| 2 | Out | Symbol Display Up | 34 | N/A | Available for User Developed Hardware |
| 3 | Out | Symbol Display Down | 35 | N/A | Available for User Developed Hardware |
| 4 | Out | Symbol Display Left | 36 | N/A | Available for User Developed Hardware |
| 5 | Out | Symbol Display Right | 37 | N/A | Available for User Developed Hardware |
| 6 | Out | Symbol Display ? | 38 | N/A | Available for User Developed Hardware |
| 7 | Out | Symbol Display ! | 39 | N/A | Available for User Developed Hardware |
| 8 | In | Random Number Generator | 40 | N/A | Available for User Developed Hardware |
| 9 | In | Gamepad | 41 | N/A | Available for User Developed Hardware |
| 10 | In | Keyboard | 42 | N/A | Available for User Developed Hardware |
| 11 | N/A | Reserved for Future Use | 43 | N/A | Available for User Developed Hardware |
| 12 | N/A | Reserved for Future Use | 44 | N/A | Available for User Developed Hardware |
| 13 | N/A | Reserved for Future Use | 45 | N/A | Available for User Developed Hardware |
| 14 | N/A | Reserved for Future Use | 46 | N/A | Available for User Developed Hardware |
| 15 | N/A | Reserved for Future Use | 47 | N/A | Available for User Developed Hardware |
| 16 | Out | GPU Execute | 48 | N/A | Available for User Developed Hardware |
| 17 | Out | GPU Command | 49 | N/A | Available for User Developed Hardware |
| 18 | Out | GPU X1 | 50 | N/A | Available for User Developed Hardware |
| 19 | Out | GPU Y1 | 51 | N/A | Available for User Developed Hardware |
| 20 | Out | GPU X2 | 52 | N/A | Available for User Developed Hardware |
| 21 | Out | GPU Y2 | 53 | N/A | Available for User Developed Hardware |
| 22 | Out | GPU Color | 54 | N/A | Available for User Developed Hardware |
| 23 | Out | GPU Working Buffer | 55 | N/A | Available for User Developed Hardware |
| 24 | Both | GPU Register | 56 | N/A | Available for User Developed Hardware |
| 25 | In | GPU Status | 57 | N/A | Available for User Developed Hardware |
| 26 | N/A | Reserved for Future Use | 58 | N/A | Available for User Developed Hardware |
| 27 | N/A | Reserved for Future Use | 59 | N/A | Available for User Developed Hardware |
| 28 | N/A | Reserved for Future Use | 60 | N/A | Available for User Developed Hardware |
| 29 | N/A | Reserved for Future Use | 61 | N/A | Available for User Developed Hardware |
| 30 | N/A | Reserved for Future Use | 62 | N/A | Available for User Developed Hardware |
| 31 | N/A | Reserved for Future Use | 63 | N/A | Available for User Developed Hardware |

## *Character Map*

| Value | Character | Value | Character | Value | Character | Value | Character |
|---|---|---|---|---|---|---|---|
| 0 | NULL | 19 | I | 38 | = | 57 | Space |
| 1 | 0 | 20 | J | 39 | \ | 58 | Carriage Return |
| 2 | 1 | 21 | K | 40 | ( | 59 | Tab |
| 3 | 2 | 22 | L | 41 | ) | 60 | Backspace |
| 4 | 3 | 23 | M | 42 | - | 61 | Escape |
| 5 | 4 | 24 | N | 43 | + | 62 | Control |
| 6 | 5 | 25 | O | 44 | ; | 63 | Alt |
| 7 | 6 | 26 | P | 45 | : | 64 | F1 |
| 8 | 7 | 27 | Q | 46 | ' | 65 | F2 |
| 9 | 8 | 28 | R | 47 | “ | 66 | F3 |
| 10 | 9 | 29 | S | 48 | , | 67 | F4 |
| 11 | A | 30 | T | 49 | . | 68 | F5 |
| 12 | B | 31 | U | 50 | < | | |
| 13 | C | 32 | V | 51 | > | | |
| 14 | D | 33 | W | 52 | ? | | |
| 15 | E | 34 | X | 53 | ! | | |
| 16 | F | 35 | Y | 54 | / | | |
| 17 | G | 36 | Z | 55 | * | | |
| 18 | H | 37 | _ | 56 | \| | | |

## *Gamepad Key Values*

| Value | Key |
|---|---|
| 0 | NULL |
| 1 | Back |
| 2 | Start |
| 3 | Ok |
| 4 | Stop |
| 5 | Up |
| 6 | Down |
| 7 | Left |
| 8 | Right |
| 9 | A |
| 10 | B |
| 11 | X |
| 12 | Y |