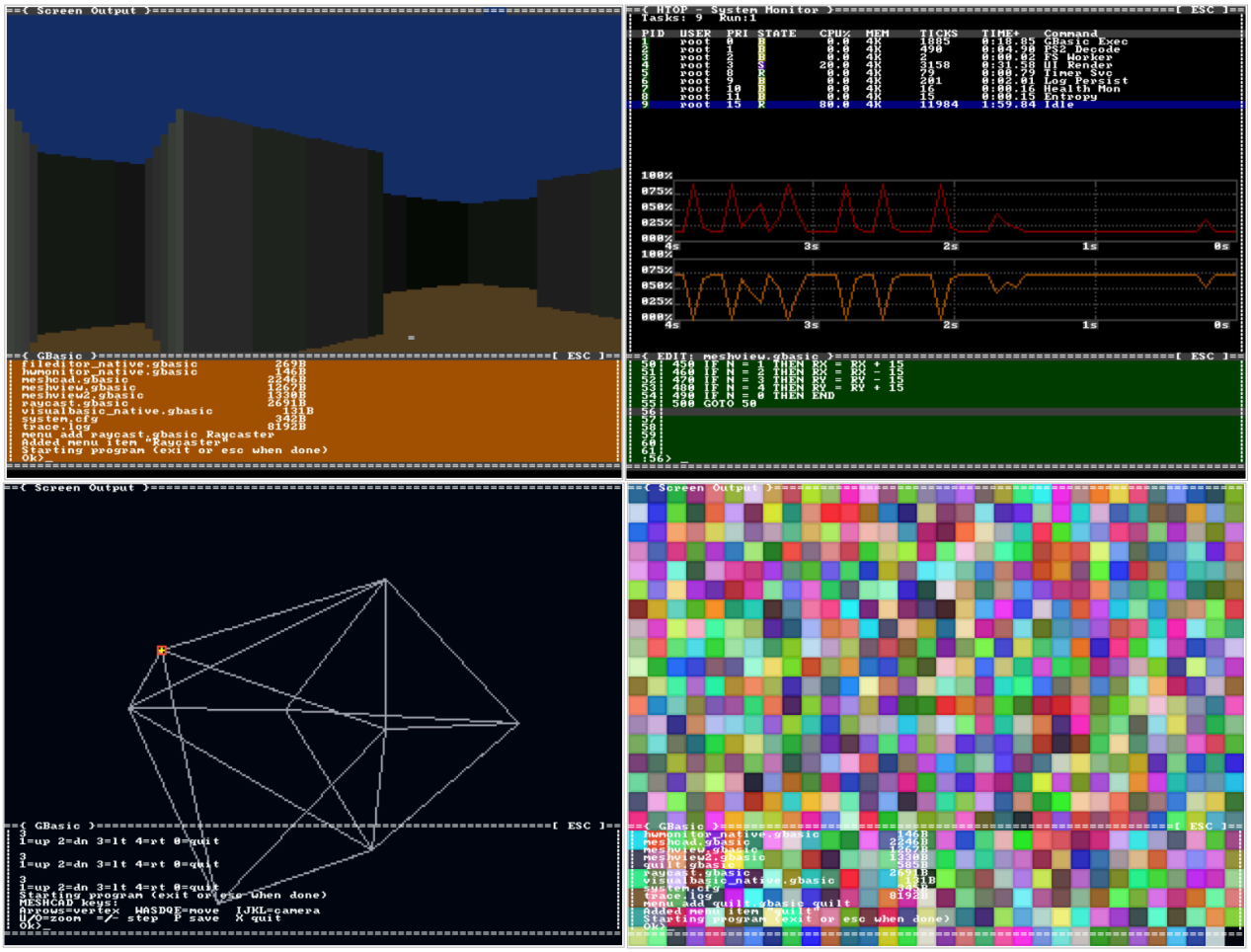
NiosV RTOS

ECE 243, Computer Organization



Ayan Ali
Avery Lor

Table of Contents

NiosV RTOS Overview

The NiosV RTOS is a real-time operating system that is able to be run on the NiosV soft-core CPU placed on the DE1-SoC. The operating system contains the GBasic programming language interpreter, short for GraphicalBASIC, a language that has a large subset of the language features of QBasic / GW-BASIC with special graphics and miscellaneous features.

The GBasic interpreter is able to interpret GBasic programs for running directly on top of the core RTOS, even some parts of the RTOS are also programmed in GBasic!

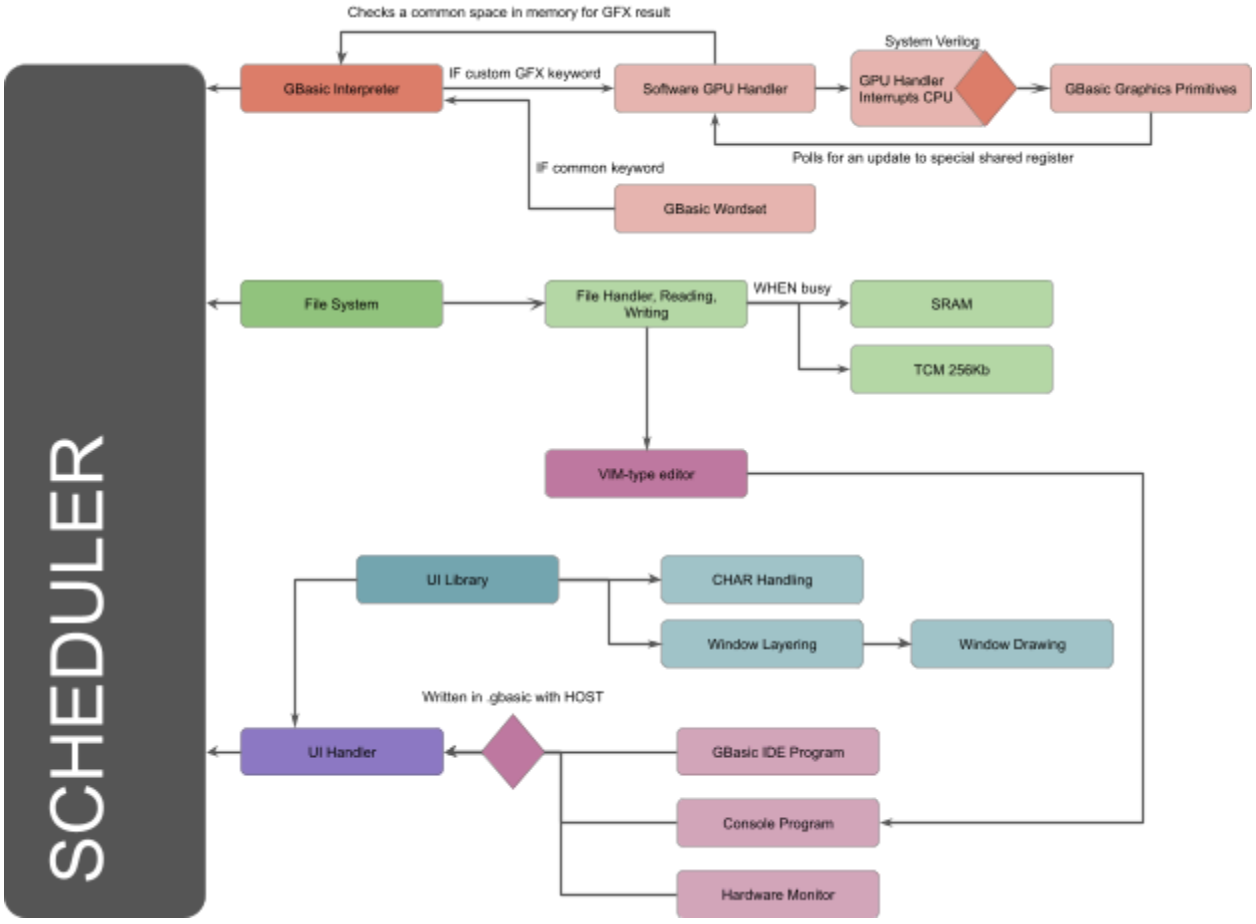


Figure 1. High-level Block diagram of the NiosV RTOS.

From a scheduler design perspective this means we prioritize determinism with multiple levels of task priority and interrupt priorities, this entire project will be synched to the RTOS rather than running asynchronously giving the true behaviour of an RTOS.

Technical Implementation of Specifics

This section describes the implementation specifics behind how the NiosV RTOS works. We have only selected a few technical implementations for explanation as we find them the most interesting and save us time in writing this report.

Scheduler

The scheduler for the Nios V RTOS is inspired by the FreeRTOS scheduler. The big difference between our scheduler is that we do not have a suspended task state which we did not implement mostly due to time. The aim of this scheduler is to prioritize determinism while minimizing tasks blocking the CPU. After all initial setup is done in the program, the program enters an infinite while loop which will continually check for the highest priority ready task to be run next.

The highest priority task has a priority number of 0. Every number that is greater than 0 is considered a lower priority task in order (so 1 is next highest priority, then 2 etc.).

It is important to note that our entire program is structured around this architecture. We wrote our own RTOS and the custom commands necessary to write our program surrounding this architecture. After all the initial set up the scheduler will continually jump between the function pointers of each task.

There are four states for a task.

State	**Description**
Scheduled	The task is currently running on the processor
Ready	The scheduler will take a look at this task to determine if it is of the highest priority to be run by the scheduler
Blocked	This task has no reason to be ready to be run by the scheduler yet (it has not received its semaphore to wake up)
Terminated	This task is no longer considered for either of the three states stated above, it will never run again, effectively being removed from memory

For tasks with the same priority that are both ready, the tasks with the earlier creation time (lower system tick) is chosen first. However, no high priority task can permanently block the CPU. A global time slice constant is defined which forces tasks to be rescheduled every 5 system ticks. As you might have noticed there is a chance that the CPU can block forever on the highest priority task; however this choice is by design and it is up to the programmer of the RTOS to effectively manage task priorities.

Every single task is defined by the following struct.

```
// task control block
typedef struct TaskControlBlock {
    uint32_t id;              // task id
    char name[32];            // task name
    TaskState state;          // task state
    TaskFunction function;    // pointer to entry function for task logic
    void* arg;                // argument passed to task function
    uint32_t stack[TASK_STACK_SIZE];
    uint32_t* stack_ptr;      // where the task is on the stack
    uint32_t cpu_time;        // ticks spent running
    uint32_t create_time;     // when created
    uint32_t priority;        // 0 = highest
    uint32_t last_start_tick; // tick when task was last scheduled running
} TaskControlBlock;
```

As you might expect in any other RTOS we have our own commands to create tasks, delay tasks, and kill tasks.

Tasks can only move to the ready stage when they receive their respective semaphore. A counting semaphore is used which increments based on if certain actions occur (often interrupt driven), for example a compiler flag is set when the GBASIC interpreter is running that will later allow for the necessary semaphore to be created for the task that actually executes the GBASIC to be run. The task instantly consumes its semaphore when it receives it and as a result will move to ready. However, after it has run without its semaphore it will move back to being blocked.

When a task is created it is automatically placed into the blocked state. The task cannot move from the blocked state to the ready state without its semaphore. Custom mutex implementations were also created to prevent race conditions during concurrent programming.

A list of all the tasks and their relevant information is shown below.

Task Name	Task Priority	Task Usage
GBasic Exec	0	Handles the running of the GBASIC interpreter, a flag will be set when the PS2 decodes that the program is executed to wake the task.
PS2 Decode	1	Decodes the PS2 input. The ISR for the PS2 places the input into a queue and this task drains that queue and responds with the appropriate actions, waking the UI render task to respond in an appropriate way.

FS Worker	2	Task that handles all file system actions. This includes navigating directories, making a directory, and editing a file.
UI Render	3	This task is responsible for rendering the UI. It is woken both by the PS2 decode and by a timer to update the systick count displayed on screen.
Timer SVC	8	This is a lower rate timer that triggers the log persist, health monitor and entropy tasks. These are considered background tasks.
Log Persist	9	Flushes queued trace messages to trace.log on the filesystem.
Health Mon	10	Checks for internal log warnings and if logs are dropping data.
Entropy	11	Attempt to create a random value by randomly XOR'ing system values such as systicks and the ticks of the current task. This is to replicate how real computers do encryption (on a really basic level).
Idle	15	This task is the only task that has no semaphore and is guaranteed to run if there are no higher priority ready tasks.

To better understand the scheduler, a diagram outlining the typical flow of the scheduler is shown below.

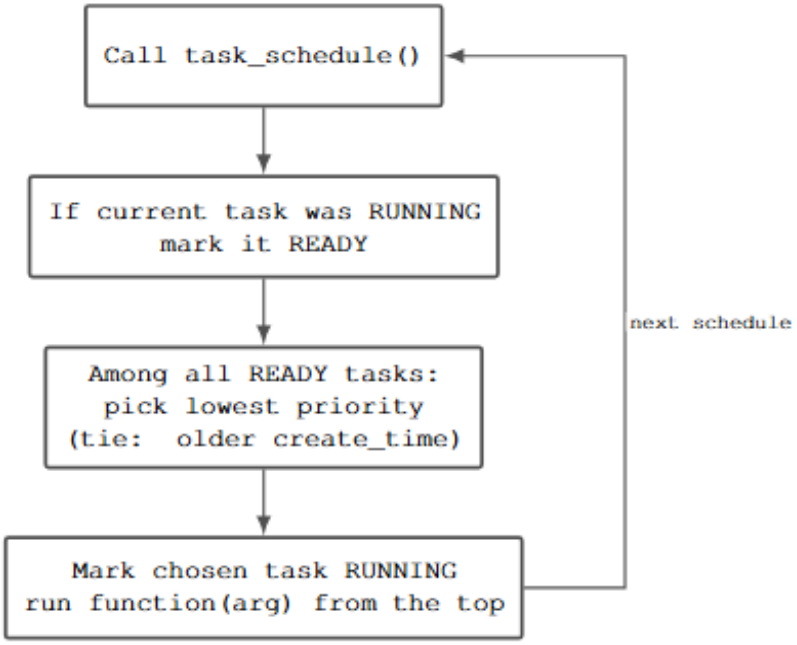


Figure 2. State diagram of how the task states change for the RTOS, driving by semaphores.

GBasic Data Model

At runtime GBasic maintains a set of fixed-size stores. The table below summarizes where the program state lives and which parts are shared pools.

Runtime Store	Role and Capacity
Program Lines	stores source lines for execution order (up to 128 lines, 80 chars each)
Variables	scalar working state (A–Z plus named variables, up to 64 total)
Control Stacks	loop/subroutine state for flow control (GOSUB depth 16, FOR depth 8)
Array Pool	shared integer array memory (2048 ints total)
Matrix Slots	transform matrices used by 3D math (8 slots of 4x4 fixed-point values)
Mesh Buffer	geometry working set (up to 256 vertices and 512 faces)
Interaction Model	program lines execute into variables, variables index arrays and feed matrix operations, and matrix/array data support mesh transforms and rendering

GBasic Execution Cycle

The interpreter follows a cooperative cycle, this means that the GBasic interpreter always hands control of the RTOS back to the scheduler at each program step rather than for a full program execution. Programs advance one step at a time, allowing UI and RTOS tasks to remain responsive.

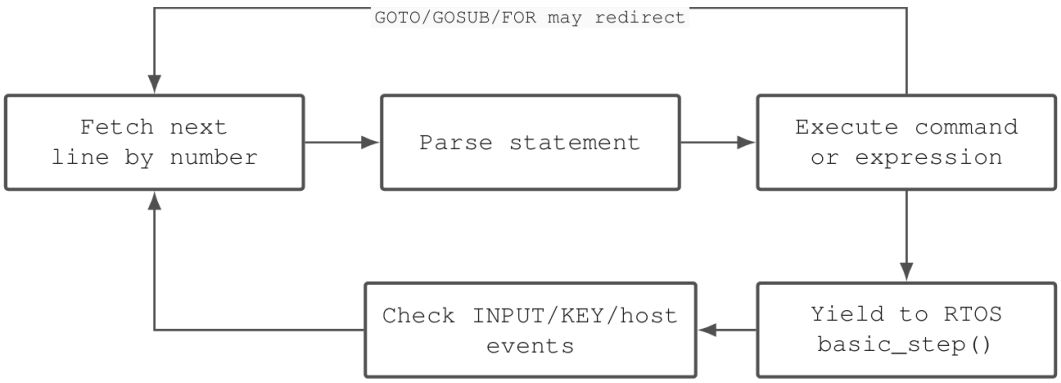


Figure 3. The interpreter loop is integrated with the RTOS scheduler.

With regards to the way the GBasic interpreter fetches by line number, it does not fetch by way of reading a file line by line and instead searching for a given GBASIC line number, essentially this means that you program as written a `.gbasic` need not be in order as long as the GBasic

line number is to your choosing. This is helpful if you need to insert a new line without having to edit all preceding lines in the code. See the below example:

```
10 INPUT "ENTER THE FIRST NUMBER"; A
30 LET C = A*B
40 PRINT C
50 GOTO 10

60 REM  I forgot to ask the user to enter B! No worries.
11 INPUT "ENTER THE SECOND NUMBER", B
```

GBasic Graphics Pipeline

Graphics operations are organized in layers so programs can scale from single-pixel demos to matrix-driven mesh rendering. For best throughput, prefer higher-layer hardware-backed commands when available. In the actual source code of the RTOS, higher layer functions use lower level code as helpers to complete their tasks if they are on the same tier.

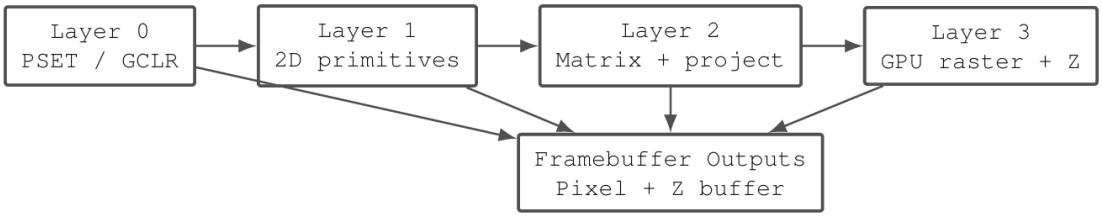


Figure 4. Layered graphics path from basic raster operations to GPU-accelerated depth writes.

Functions written in GBasic are implemented in various different ways to ensure that they are as fast as possible whilst minimizing the development time for us. Figure 5 shows off the various ways we have implemented our GBasic function to balance this time vs. speed.

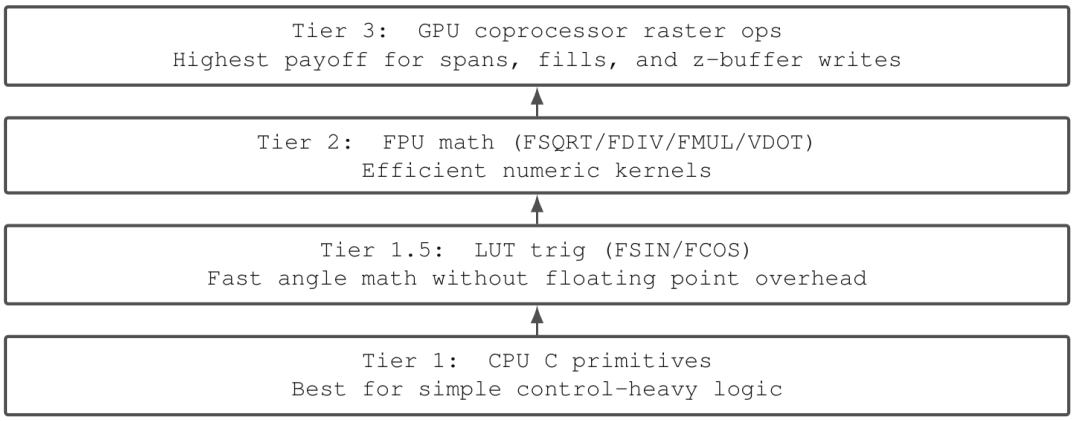


Figure 5. Decision table for selecting functions based on internal implementation.

Tier 1 Functions are directly implemented in C and run on the NiosV ALU, these functions were chosen to run on the ALU as there was no real gain with running on custom hardware due to the chance of the custom hardware being placed a large distance from the memory, slowing down computation.

Tier 1.5 was implemented quite late in the development timeline as we noticed SINF and COSF were quite slow and called 4 GBasic instructions in the interpreter, to speed this up we implemented a LUT.

It is important to note that Tier 2 FPU acceleration has not been completely validated as we ran out of time due to the vast number of features required. In this case, we have a fallback on a C-based primitive and so all programs should run as normal.

It is important to note that Tier 3 GPU HW acceleration is non-functional at the moment as we ran out of time to implement it. In this case, since the custom hardware is not being detected there is a C-based primitive fallback (that was used to validate the hardware during development) that is being called in the code so all programs should run as normal.

How to Operate the NiosV RTOS

This section contains two parts, the first part contains the setup for the NiosV RTOS, and the second part contains the demonstration of all features.

Setup for CPULator and Compilation on DE1-SoC

This section explains how to set up the NiosV RTOS for simulation, running on the DE1-SoC, and adding new files such that they are persistent across system runs.

Setting up the NiosV RTOS

To setup the NiosV RTOS for simulation or compilation for the DE1-SoC, follow the process:

1. Run the `combineFiles.py` and the combined file will be in the `combined\mainCombined.c` for simulation or for compilation uses
2. Then:
    a. Place the `mainCombined.c` into CPUlator or into the relevant gdb folder filled with the provided `makefile` as part of this project as it contains special `math.h` headers

NiosV Filesystem and adding new persistent files

Before discussing adding persistent files, we will discuss the file system. The file system on the NiosV supports the ability to add directories, create files, remove files, and edit files among other commands that can be found under the `help` command in the console. All files in the NiosV RTOS are stored as plain-text files with the exception of the .mesh file which contains encoded vertex information generated by a `objToMesh.py` script.

When the RTOS is running you are able to create new directories and files that correspond to the `.gbasic`, `.mesh`, `.cfg`, and files names with no exact extension. The same can be done in the windows file system where you build the combined `combined\mainCombined.c` file for use in building using gdb or for simulation in CPUlator

To add a new file, follow the process:

1. Start by creating the file as it would be in the NiosV RTOS file system, i.e. `testProgram.gbasic`
2. Place this file in the `src\program` directory
3. Run the `combineFiles.py` and the combined file will be in the `combined\mainCombined.c` for simulation or for compilation uses

Operating the NiosV RTOS

This section discusses the various features of the NiosV RTOS and how to navigate them.

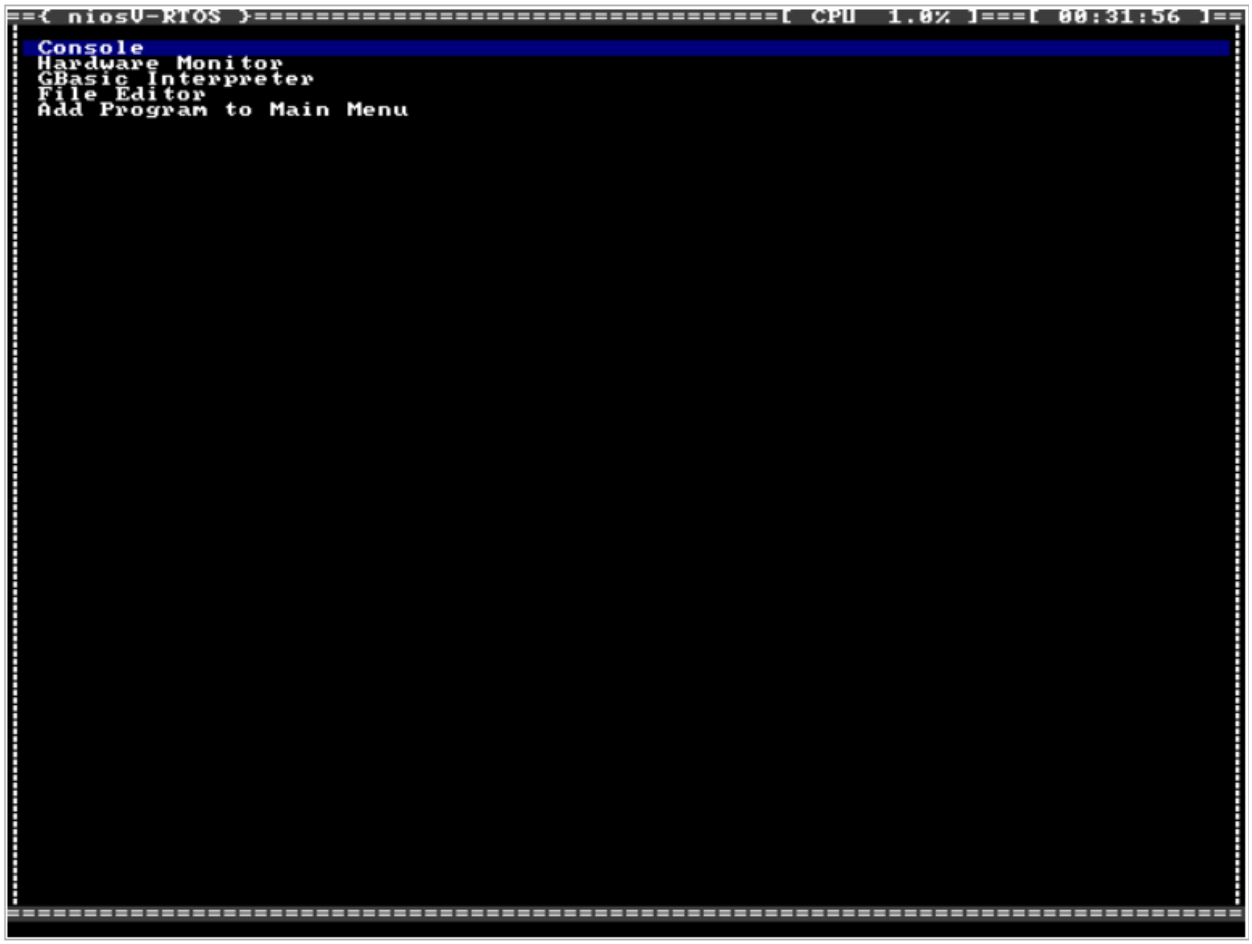


Figure 6. Shows the NiosV RTOS at initial start. The current program (i.e. Hardware Monitor, etc.) are all `.gbasic` scripts and are stored in the file system as `<name>_native.gbasic`.

Main Menu Navigation

The following scheme is mimicked across all scrollable or selectable menus in the RTOS:

1. Navigate menus via `upArrow` and `downArrow`
2. Enter a program using `enter`

The Console

You should start of by doing the following:

1. Type `help` to be able to see the full list of commands and their uses
2. Type `help 2` for the 2nd page of commands, there are currently 4 page
3. Commands have aliases, i.e. `file ls` has the alias `ls`

Navigating the Console, the file editor is managed in a similar way:

1. Scroll up and down the console via `upArrow` and `downArrow`
2. Edit a line of text using `leftArrow` and `rightArrow`
3. Send a command using `enter`
4. Enter a previous command using `shift+upArrow`
5. Scroll through previous commands using `shift+downArrow` and `shift+upArrow`

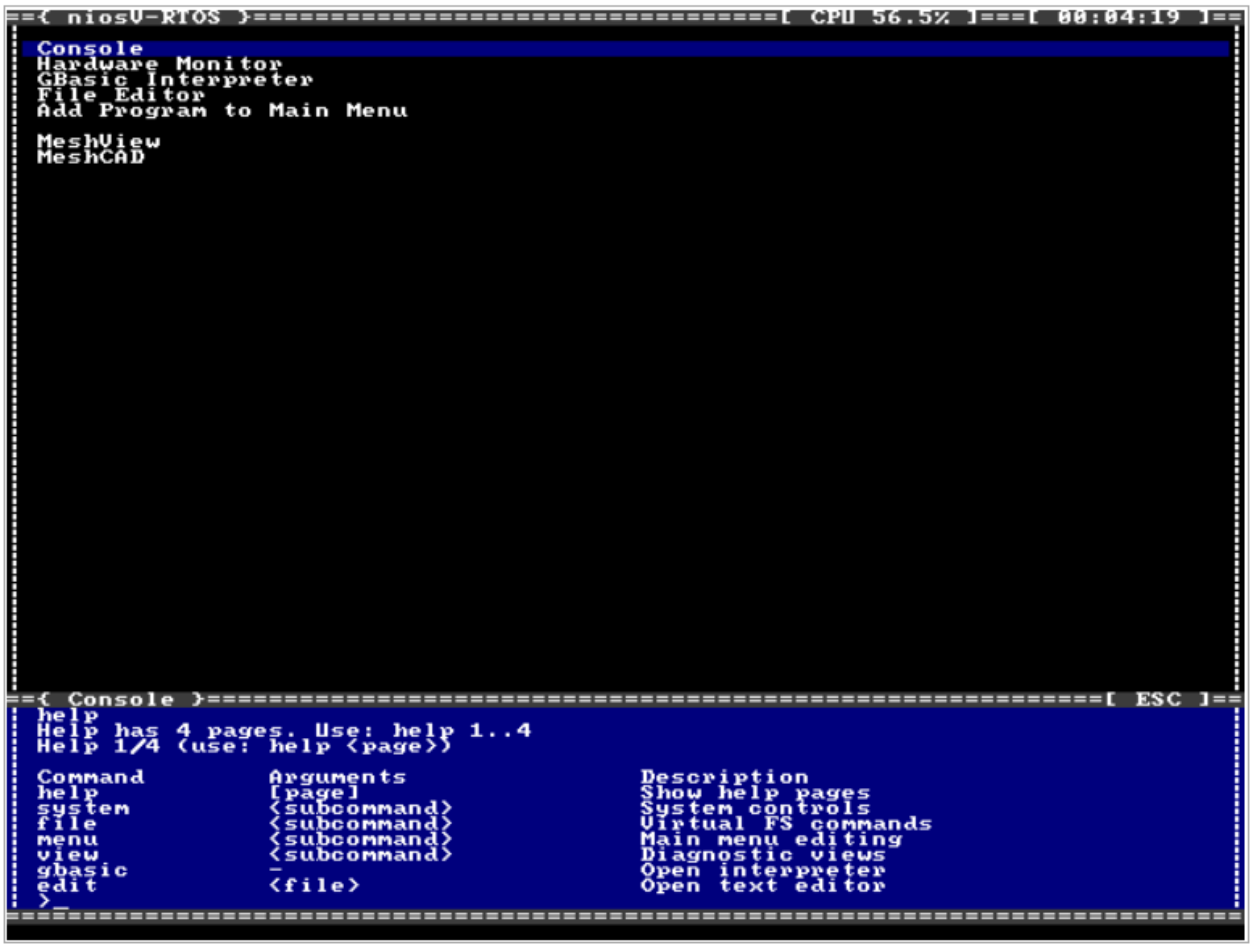


Figure 7. The console current shows the output of typing in the help command.

The File Editor

Using the file editor works as follows

1. You can edit files by using the `edit` command or by using the File Editor macro in the main menu
2. Type `:wq` on an empty line to close and save the file
3. Type `:q` on an empty line to close without saving
4. Press `enter` to save a line after editing
5. It is highly recommended when writing `.gbasic` programs that you increment your GBasic line counter by 10 in the case you want to insert lines in the program without having to edit all preceding lines after the line.

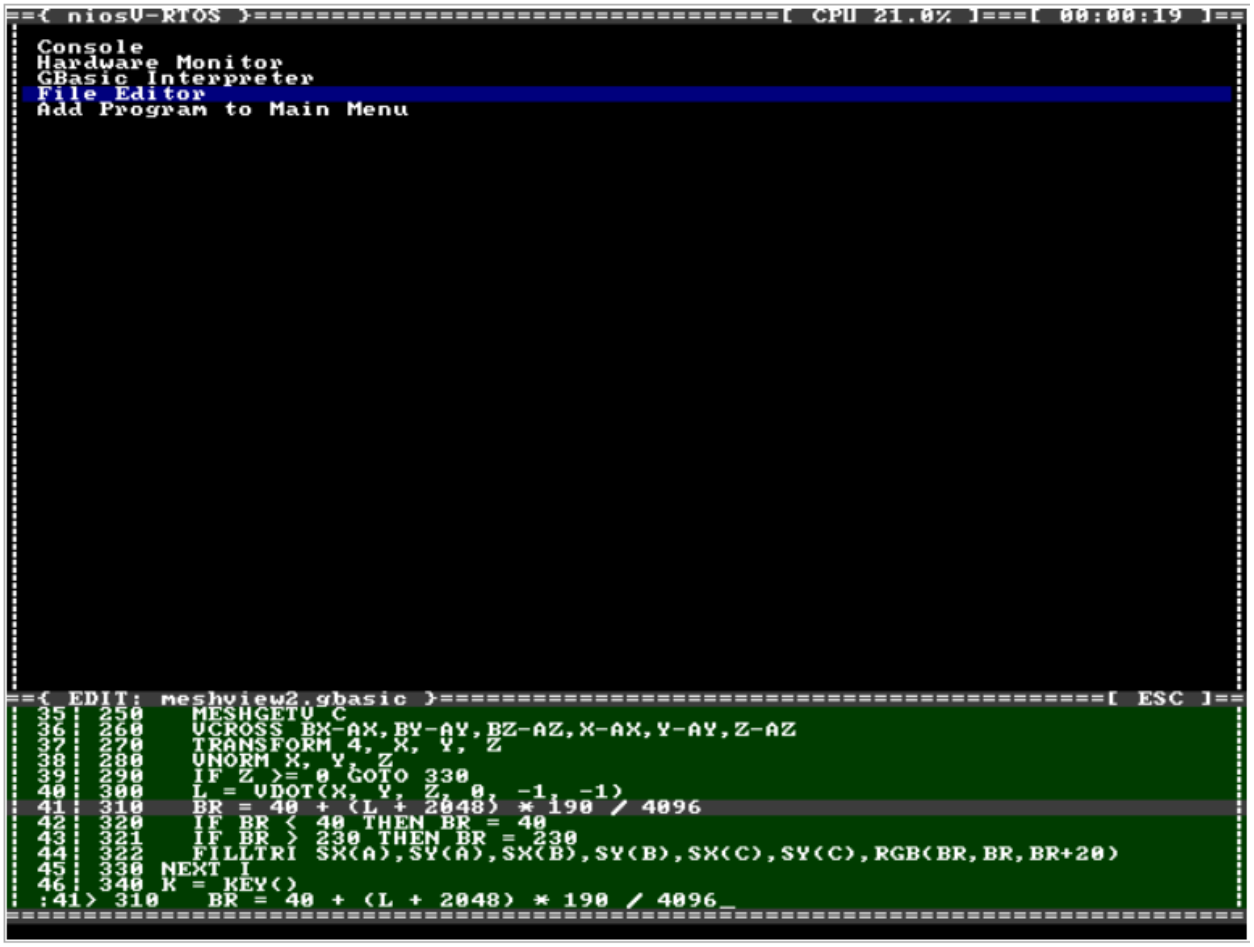


Figure 8. Console view when editing GBASIC program in file editor.

The Hardware Monitor

To access the hardware monitor

1. move the selection bar over “Hardware Monitor”
2. Press enter
3. Exit the monitor and press the escape key.

Several useful commands are listed below:

1. Tasks can be terminated by entering the command into the terminal shown in the HTOP `kill <task_id>` or `htop kill <task_id>`
2. It is useful to know that all tasks can be moved back from terminated by doing `ctrl + R` on the keyboard.
3. Up arrow and down arrow will navigate through tasks shown on the HTOP, the graph closest to the bottom will show the CPU% that task uses relative to the idle task.
4. Hold the shift key while using the arrow key to use normal arrow actions in the terminal.
5. The graph closest to the graph shows total CPU% by calculating the % of each task run relative to the idle task.

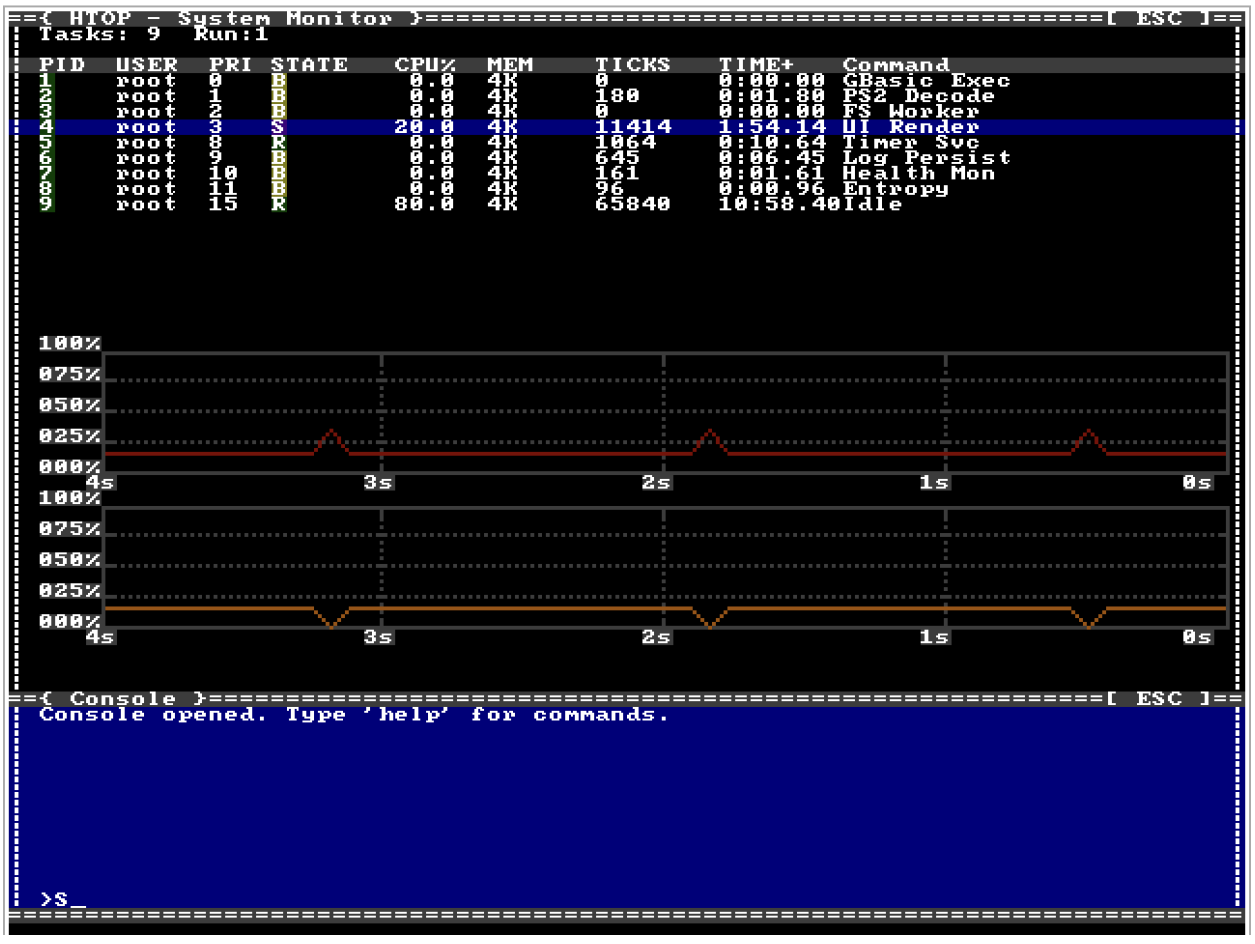


Figure 9. Hardware monitor / HTOP displaying task, CPU% and console to interact with.

The GBasic Interpreter

To run GBasic programs do the following:

1. Type `gbasic` into the terminal
2. Enter `load <filename>` to load a program, from there the program can be ran
3. Enter `run` to actually run the program

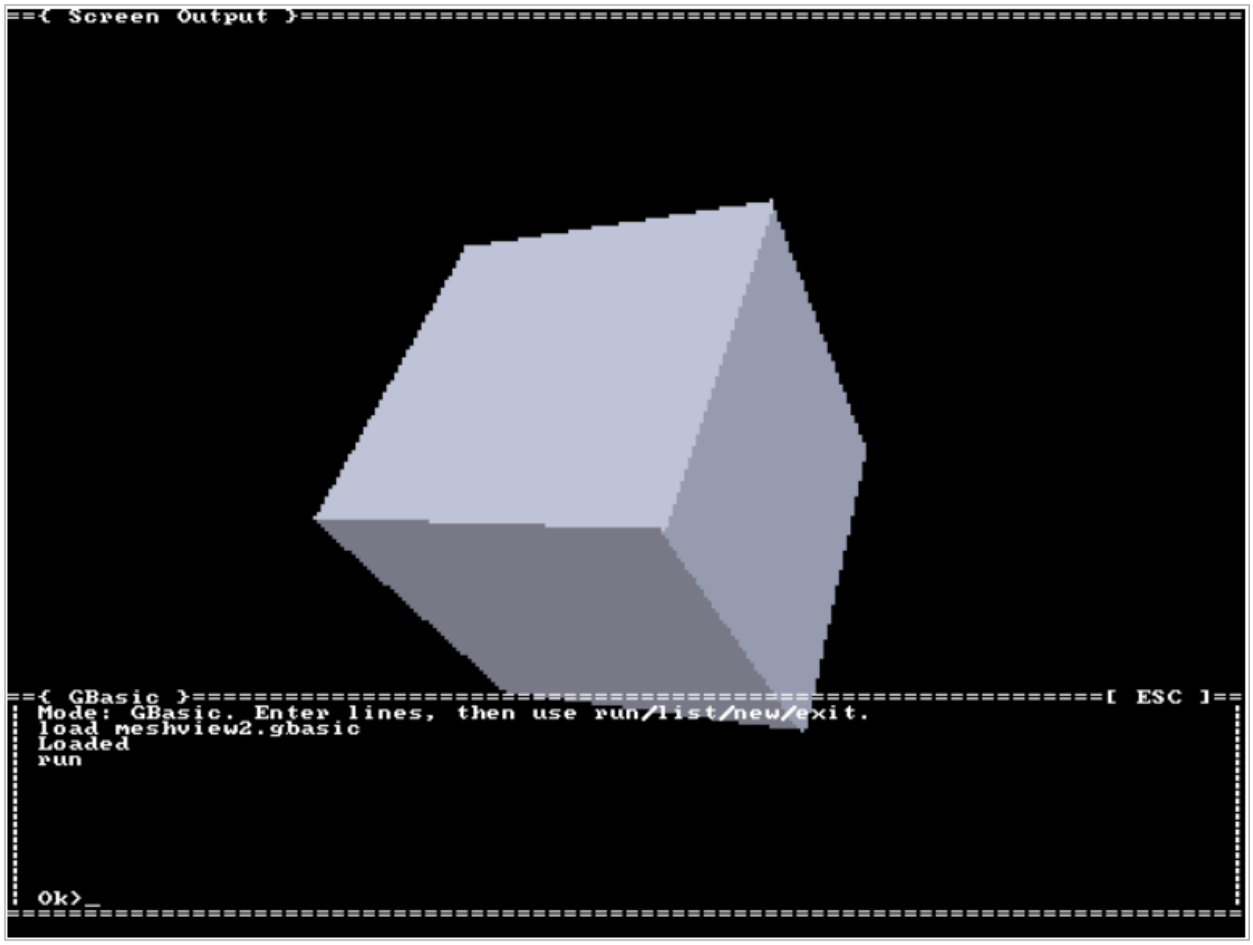


Figure 10. terminal view of entering into a GBasic program and running it.

Adding Programs to the Main Menu

To add a program to main menu do the following:

1. On the main screen select “Add Program to Main Menu”
    a. The terminal will fill out the command `menu add`
2. Enter your desired program to add, followed by the label you desire for it to be shown on the main console.
    a. Full example: `menu add cube.gbasic cube`

Figure 11. Adding cube program through the “Add Program to Main Menu” option

Editing <name>_native.gbasic Files

Files like `visualbasic_native.gbasic` are launcher scripts.

Figure 12. Contents of a typical `<name>_native.gbasic` file.

So what this means is that you can execute console commands through `.gbasic` files.

As a result kill commands can be run through these files essentially allowing for attackers to easily kill all tasks on the RTOS.

Editing the system.cfg File

The system.cfg file is the global settings file for the runtime. To use it:

1. Enter `edit system.cfg`
    a. Global variables will be defined, in which you can edit the values of
2. First, for a basic example you can edit the colors of the RTOS

3. Each color corresponds to all references of that color throughout the UI so changing the HEX value for the `color_black` variable changes the background to the new HEX value.

Figure 13. Editing the `system.cfg` color_black variable which controls all components that use the black color.

As you can see by entering and saving the change we made in Figure 14. We can get the RTOS's background to change to pink.

Figure 14. RTOS background color black changed to pink by editing the `system.cfg` file.

One more thing that is interesting is that you can configure how many ticks at most the basic executor can receive.

- Setting a small value will lead to better UI performance, slower program execution
- Setting a larger value will lead to better basic execution, worse UI performance

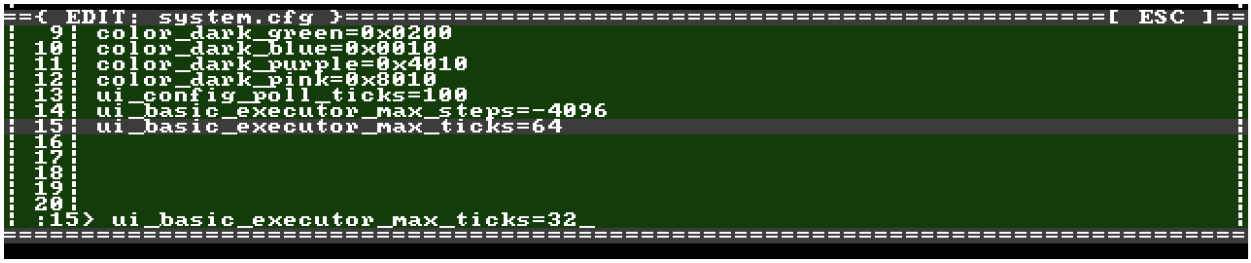


Figure 15. Setting the `ui_basic_executor_max_ticks` to a lower tick value will slow down execution, but improve UI rendering.

GBasic Language Reference

This section contains a full reference on the functions and limitations that are available for use. Example programs are available under the `program\` folder in the codebase.

Overview

Data Types

GBasic includes the following data types, it should be noted that DIM assigns arrays to a shared memory area and has no dynamic resizing. This means that you should take care of the total usage.

Data Type	Description
integers only	all variables store 32-bit signed integers
fixed-point graphics	world coordinates use FP_SHIFT=10 (multiply by 1024)
arrays	1D integer arrays via DIM, shared pool of 2048 elements (8 KB)

Variables

Variables also have limits on them, taking a further note of the limits of GBasic as it stands should prove useful.

Property	Description
single-letter	A–Z (26 variables, case-insensitive)
multi-character	up to 64 total with names like PX, MY, SPEED
automatic creation	variables are created on first use, initialized to 0
reserved outputs	`COLOR r,g,b` → C, `TRANSFORM` → X,Y,Z,W, `PROJECT` → X,Y,Z, `ZPSET` → Z

Program Structure

The method through which execution occurs is discussed in the GBasic Execution Cycle section. Taking a further note of the limits of GBasic as it stands should prove useful.

Rule	Description
line numbers	required for all program lines (e.g. `10 PRINT "HELLO"`)
execution order	runs from lowest to highest line number
program capacity	up to 128 lines, 80 characters per line

Operators

Arithmetic		Comparison & Logical	
`+`	addition	`=`	equal
`-`	subtraction	`< >`	less / greater than
`*`	multiplication	`<= >=`	less/greater or equal
`/`	integer division	`<>`	not equal
`%`	modulo (remainder)	`AND`	logical and
`()`	grouping	`OR`	logical or

Core Commands & Host Bridge

This section contains the main structural commands of GBasic and one special command for running strings in the console and opening the console named `HOST`. This is most useful for things like the `<name>_native.gbasic` files.

Command	Syntax	Notes
REM	`REM text`	comment line, ignored by interpreter
PRINT	`PRINT expr [; expr ...]`	`;` suppresses space between items; `,` tabs to next 8-column position
LET	`[LET] var = expr`	LET keyword is optional
IF...THEN	`IF cond THEN stmt`	supports inline statements or GOTO targets; comparison: `= < > <= >= <>`; logical: `AND OR`
GOTO	`GOTO linenum`	unconditional jump to line number
GOSUB	`GOSUB linenum`	subroutine call; up to 16 nested levels
RETURN	`RETURN`	return from most recent GOSUB
FOR...NEXT	`FOR v=a TO b [STEP s] ... NEXT v`	counted loop; STEP defaults to 1; up to 8 nested
INPUT	`INPUT ["prompt";] var`	optional prompt with semicolon; stores integer or ASCII value of first character
CLS	`CLS`	clear console output
END	`END`	stop program execution
HOST	`HOST "action", "payload"`	host shell/UI bridge; result in Z (0=success); actions: exec, prefill, mode, close

Arrays

Command	Syntax	Notes
DIM	`DIM name(size) [, ...]`	1D arrays from shared pool (2048 ints, 8 KB); up to 16 arrays; elements init to 0
array access	`name(index) = expr`	-indexed; out-of-bounds reads return 0, writes silently ignored
ERASE	`ERASE`	reset all arrays and free pool memory

Built-In Functions

General Functions

Function	Description
`KEY()`	non-blocking key poll; returns 0 when no key pending; printable keys return ASCII (letters normalized to uppercase); arrows: Up=1001, Down=1002, Left=1003, Right=1004
`LEN("text")`	length of a string literal
`ASC("text" [, index])`	ASCII code of character at index (default 0); out-of-range returns 0
`VAL("text")`	parse signed integer from a string literal
`POINT(x, y)`	read pixel color from framebuffer
`RGB(r, g, b)`	convert 8-bit RGB to RGB565 color value
`MGET(slot, row, col)`	read one matrix element from a slot
`RND(n)`	pseudo-random integer from 0 to n−1

Mesh Query Functions

Function	Description
`MESHVERTS()`	returns loaded mesh vertex count
`MESHFACES()`	returns loaded mesh face count

Tier 1.5 and Tier 2 Math Functions

These functions provide faster math paths through LUT and FPU-backed operations.

Function	Description
`FSIN(angle)`	LUT-backed sine; returns fixed-point
`FCOS(angle)`	LUT-backed cosine; returns fixed-point
`FSQRT(x)`	hardware square root using niosV FPU
`FDIV(a, b)`	hardware floating-point divide
`FMUL(a, b)`	hardware floating-point multiply

`VDOT(x0,y0,z0, x1,y1,z1)`	vector dot product using FPU

Tier 2 FPU Commands

Command	Notes
`VCROSS x0,y0,z0, x1,y1,z1`	vector cross product; stores result in X, Y, Z
`VNORM x, y, z`	normalize vector; stores result in X, Y, Z as fixed-point

Graphics Commands

Layer 0: Pixel Access

Command	Syntax	Notes
`PSET`	`PSET x, y, color`	set a single pixel
`GCLR`	`GCLR [color]`	clear entire pixel buffer
`GCLROLD`	`GCLROLD [color]`	clear pixel buffer using legacy C primitive path

Layer 1: 2D Primitives

Command	Syntax	Notes
`LINE`	`LINE x0, y0, x1, y1, color`	draw line (Bresenham)
`RECT`	`RECT x0, y0, x1, y1, color`	draw rectangle outline
`CIRCLE`	`CIRCLE cx, cy, radius, color`	draw circle outline
`FILLRECT`	`FILLRECT x0, y0, x1, y1, color`	filled rectangle
`FILLRECTOLD`	`FILLRECTOLD x0, y0, x1, y1, color`	filled rectangle using legacy C path
`FILLCIRCLE`	`FILLCIRCLE cx, cy, radius, color`	filled circle
`FILLTRI`	`FILLTRI x0,y0,x1,y1,x2,y2,color`	filled triangle
`COLOR`	`COLOR r, g, b`	compute RGB565 and store in variable C

Layer 2: 3D Matrix Operations

Command	Syntax	Notes
MIDENT	MIDENT slot	set matrix to identity
MSET	MSET slot, row, col, value	set matrix element (fixed-point value)
MROTX	MROTX slot, angle	rotation around X axis (integer degrees)
MROTY	MROTY slot, angle	rotation around Y axis
MROTZ	MROTZ slot, angle	rotation around Z axis
MSCALE	MSCALE slot, sx, sy, sz	scale matrix (fixed-point values)
MTRANS	MTRANS slot, tx, ty, tz	translation matrix (fixed-point values)
MMUL	MMUL slotA, slotB, slotOut	matrix multiply (tier 3 hardware)
TRANSFORM	TRANSFORM slot, x, y, z	transform vertex by matrix; stores clip-space result in X, Y, Z, W; input coordinates are fixed-point (world × 1024)
PROJECT	PROJECT slot, x, y, z	transform and project to screen; stores screen coords in X, Y; Z = visibility (1=visible, 0=behind camera)

Mesh Commands

Command	Syntax	Notes
MESHLOAD	MESHLOAD "name"	load mesh from filesystem into internal buffer
MESHGETV	MESHGETV index	load vertex into X, Y, Z variables
MESHGETF	MESHGETF index	load face vertex indices into A, B, C variables
MESHSETV	MESHSETV index, x, y, z	update a mesh vertex using world coordinates
MESHSAVE	MESHSAVE "name"	serialize current mesh and save to filesystem; stores bytes written in Z
MESHDATA	MESHDATA "name", arrayname	read mesh file as raw int16 values into array

Layer 3: GPU Raster Operations

Primary commands below use the hardware-backed path (with internal fallback as needed). Commands ending in OLD force the legacy C primitive path.

Command	Syntax	Notes
HLINE	`HLINE x0, x1, y, color`	horizontal span (tier 3 hardware)
HLINEOLD	`HLINEOLD x0, x1, y, color`	horizontal span using legacy C path
VLINE	`VLINE x, y0, y1, color`	vertical column (tier 3 hardware)
VLINEOLD	`VLINEOLD x, y0, y1, color`	vertical column using legacy C path
ZBUFCLR	`ZBUFCLR [value]`	clear z-buffer (tier 3 hardware); defaults to 0xFFFF (max depth)
ZBUFCLROLD	`ZBUFCLROLD [value]`	clear z-buffer using legacy C path; defaults to 0xFFFF
ZPSET	`ZPSET x, y, depth, color`	z-buffered pixel write (tier 3 hardware); Z=1 if written, Z=0 if rejected by z-test
ZPSETOLD	`ZPSETOLD x, y, depth, color`	z-buffered pixel write using legacy C path; Z=1 if written, Z=0 if rejected

Limits & Implementation

Current Constraints

Constraint	Details
1D arrays only	no multi-dimensional arrays (use offset math: `arr(y * W + x)`)
no string variables	string literals are supported in PRINT and string literal functions
no floating-point	all math is integer (use fixed-point scaling)
no user-defined functions	only built-in functions; use GOSUB for subroutines
no line editing	programs loaded from filesystem or entered line-by-line
cooperative execution	programs yield via `basic_step()` for RTOS integration

Fixed-Point Conventions

Convention	Details
world coordinates	multiply by 1024 (FP_ONE) for sub-pixel precision
matrix elements	stored as fixed-point integers (FP_SHIFT=10)
angles	specified in integer degrees (0–359)
colors	RGB565 format (5 bits red, 6 bits green, 5 bits blue)

Memory Constraints

Resource	Limit
program storage	lines × 80 chars = 10 KB max
variables	× 4 bytes = 256 bytes
arrays	shared pool of 2048 ints (8 KB), up to 16 arrays
stack depth	GOSUB levels, 8 FOR loops
matrix slots	slots × 4×4 × 4 bytes = 512 bytes
mesh buffer	vertices, 512 faces max

Development Procedure

This section discusses the techniques we used to aid our development time, how we validated our custom hardware files, and how we

The Power of Scripting

We used a variety of scripts to help aid our development cycle and to reduce the time spent fiddling with very long files. We have developed a `combineFiles.py` script that is able to take all the files in the corresponding `src\`directory and place them all into a single combined file.

We have also developed a `objToMesh.py` that takes in `.obj` files from a CAD tool of choice, in this case we used Blender 5.1 and convert it to the encoded .mesh file for insertion into the persistent file folder `src\program\`

Hardware Verification Procedure

Given that lengthy synthesis is cumbersome and error-prone to allow bugs to pass through. With this, to ensure that the system verilog that was written for the graphics co-processor was functional we wrote the hardware implementation in `verilatorTest.cpp` that could be orchestrated in a test run via a makefile to compare a verilated version of the `.sv` against the `.cpp`. Once we verified the error was <1% we moved onto synthesis.

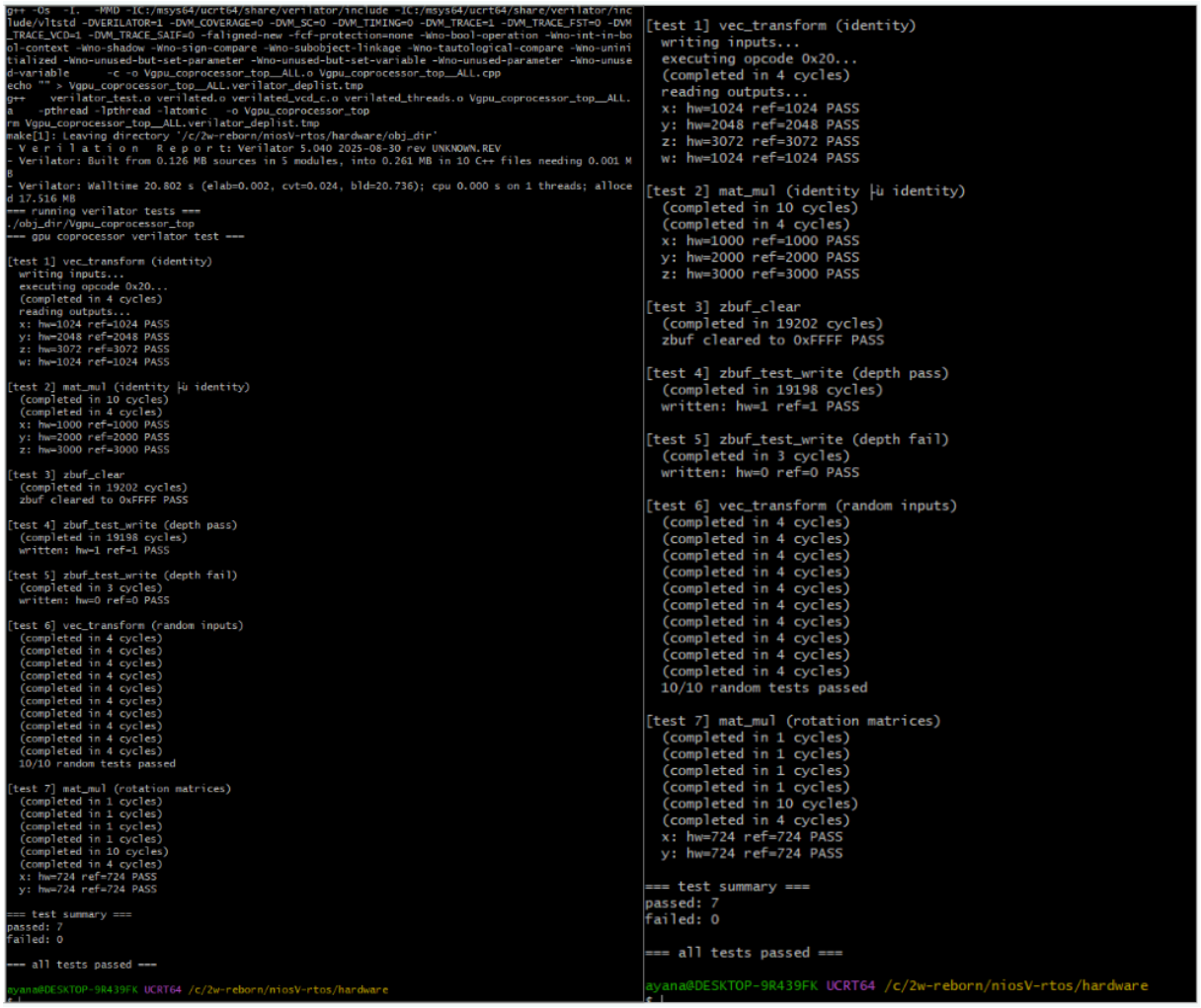


Figure 16. Shows the debug output of the makefile running the automated tests in an MSYS2 UCRT64 window.

Integrating the Hardware with the NiosV Core

When integrating our hardware we had to first obtain the platform design .qsys file. This was a major ordeal and involved an unfortunately poorly documented process. Once we obtained the NiosV Computer System for the DE1-SoC .qsys file, we removed IPs like the audio manager, LED drivers, etc. to reduce our synthesis time and wired the corresponding Avalon MM Bus Master and Slave connections to the NiosV core.

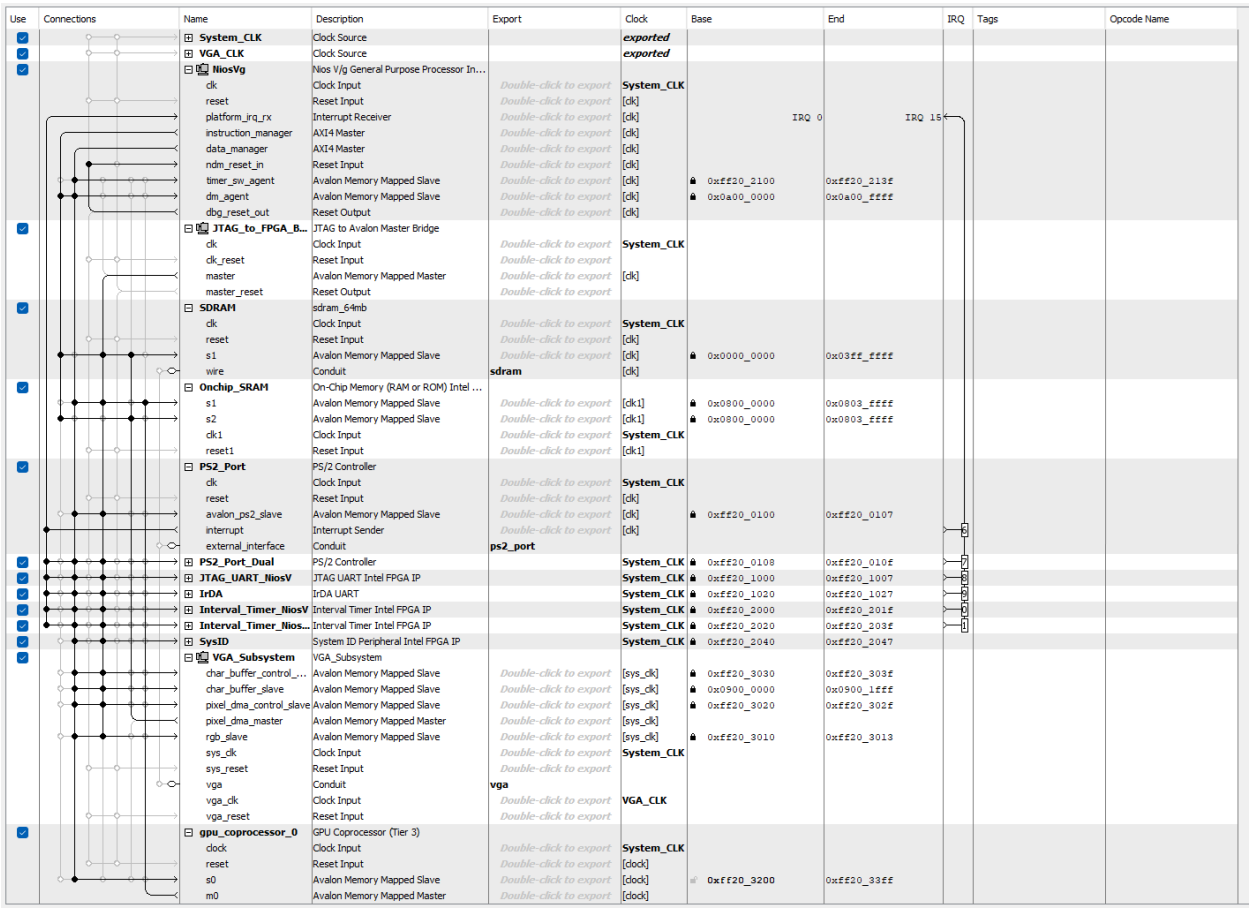


Figure 17. Here we are wiring the corresponding Avalon MM Master and Slave signals to the GPU coprocessor alongside a sharded clock and reset signal.

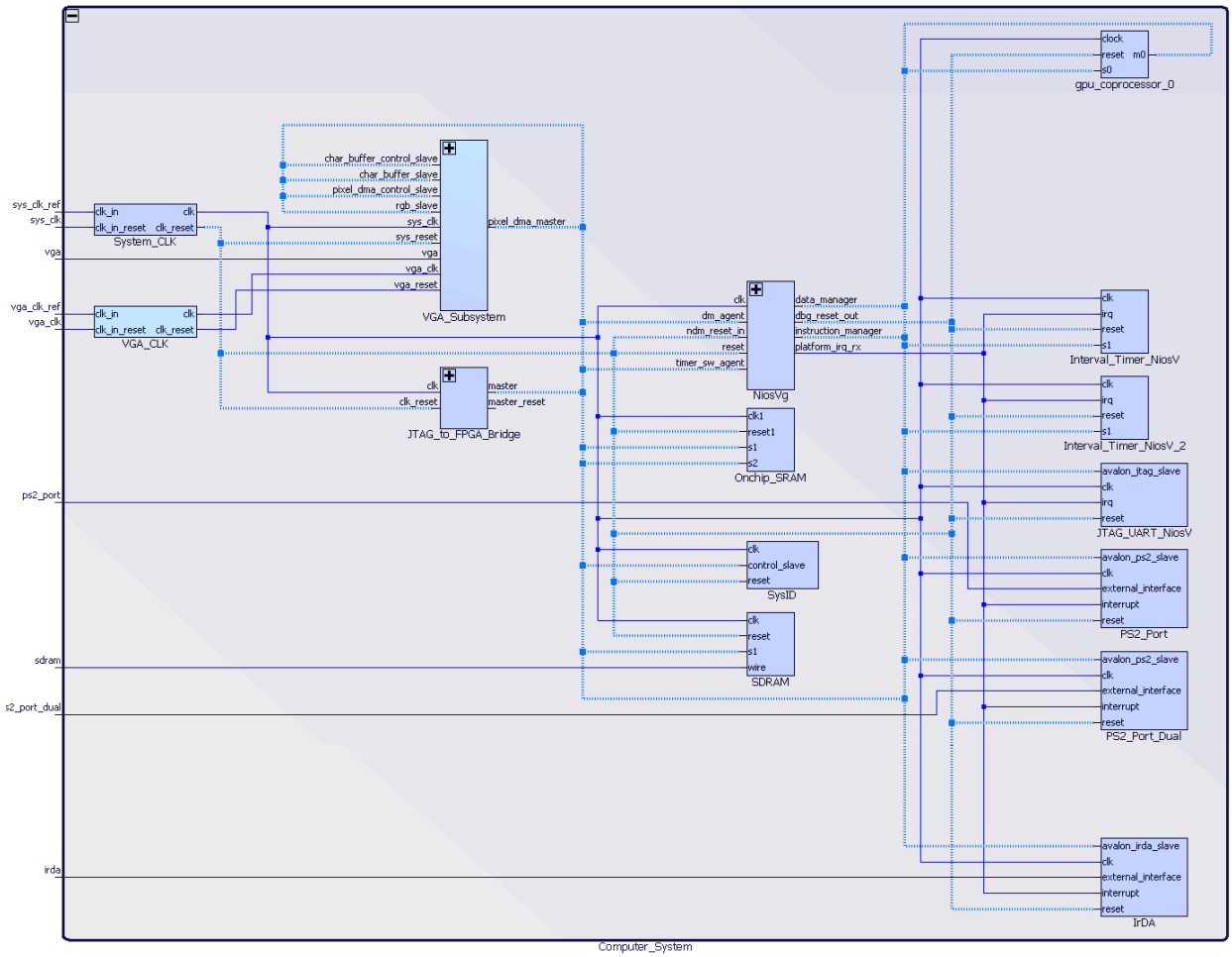


Figure 18. This figure shows the final schematic of our designed platform for synthesis into a .qif file.

Once we had completed our platform design, we obtained a .qif file that we used to synthesize into a .sof file that we could use in our gdb system makefile to upload our new NiosV platform to the DE1-SoC.

Unfortunately we were unable to synthesize in time before our project demo as the synthesis time was on the order of 3-4 hours.

Future Development

We can say that this was a ***fun*** project, we learned quite a bit. We would have done things slightly ***differently***, such as allocating more time for getting the hardware to synthesize and learn platform design. We believe that our direction of the project was ***appropriate***, in the sense that we explored a large range of featuresets and was highly visual which is always interesting.

With this in mind we aim to continue to develop this RTOS into something more usable, fun, or interesting even if it ends up being a toy.

GBasic Feature Extensions and Hardware Integration

As discussed previously, we did not manage to get our custom hardware co-processor to be able to synthesize in time. Getting the custom hardware would allow us to bring this project to the next level for the case of making our graphics programs (i.e. rasterization) run as fast as possible.

Perhaps even adding the ability to program the graphics card using a very simple shader library would be interesting as we can develop compute shaders designed to accelerate almost any task on the single core NiosV.

Missing OpenGL1.0-Adjacent Features

To make our GBasic instruction set more useful to the graphics programmer we will add the following features, these features will lend themselves very well to programs like Quake-adjacent games.

Command	Syntax	Description
`BLIT`	`BLIT sx, sy, w, h, dx, dy`	block image transfer
`TEXVLINE`	`TEXVLINE x, y0, y1, u, h, ptr`	textured vertical span
`TEXHLINE`	`TEXHLINE x0, x1, y, v, w, ptr`	textured horizontal span
`COLOR_EXPAND`	`COLOR_EXPAND src, dst, n, pal`	palette expansion
`ROP`	`ROP op, src, dst, count`	raster operations (e.g. XOR)

String Support

Adding string support would allow us to write to files more cleanly, and expand the range of features available for the programmer.

Feature	Description
string variables	suffix with $ (e.g. `NAME$ = "PLAYER"`)
string functions	LEFT$, RIGHT$, MID$, LEN, CHR$, ASC

Expanding the Filesystem and GBasic Limitations

We believe in expanding the limitations of the GBasic interpreter by way of placing the filesystem into an SDcard via GPIO pins on the DE1-SoC and reserving the fast SRAM for GBasic and other RTOS tasks.

Processes-based OS

Due to time constraints we chose to design much of our architecture surrounding an RTOS. However, in the future, given the direction we went with the project, having an incredibly deterministic scheduler is not as important as we thought originally.

I believe that given more time the RTOS could have been generalized more to a standard OS where instead of tasks we would have processes.

The main difference is that processes are much more advanced. Each process could have several tasks running within it and processes have much more ownership over their own resources in comparison to threads.

Overall delving more into further developing the robustness of our scheduler and changing our architecture from an RTOS to a general OS would have been quite interesting.

Work Attributions

The below two tables contain the work split by short description and work split by task.

Ayan	Worked on the GBasic interpreter, custom graphics hardware, scripting for development cycle, file system centralization, and GBasic programming set.
Avery	Worked on RTOS scheduler, syncing all programs to the RTOS, CPU% statistics and miscellaneous file system / UI tasks

The below tables contain a more granulated task assignment.

Assignee	Task
Avery	Scheduler Design and Implementation
Avery	RTOS Syncing
Avery	Semaphore and Mutex Creation
Avery	CPU Statistics
Avery	CPU Graph
Ayan	GBasic Interpreter
Ayan	GBasic Typeset
Ayan Avery	UI
Ayan Avery	Console UI
Ayan	PS2 Decode and UI
Ayan	Console Responsiveness (arrow keys, help directions)
Ayan	File System Initial Implementation
Avery	File System Removing Files, Directories, etc.
Ayan	Writing GBasic programs