Alexander Valys
6.111 Final Project
11/1/2006

PROJECT PROPOSAL

(full block diagram on last page)

(full block diagram on last page)

SUMMARY

The system I will be building is a GPS data logger and visualization system. The data logging and visualization components will be implemented separately - the data logger on a Digilent Nexys board, and the visualization system on the 6.111 labkit.

The data logger will write a sequence of position fixes to a removable Flash ROM module. The visualization system will use that module as input, and produce a number of visualizations of the logged position information, such as:

- Altitude vs. time
- Velocity vs. time
- 2D Position plus velocity as color
- 2D Position plus altitude (3D)
- 2D Position plus velocity (3D)
- 2D Position plus altitude plus velocity as color

Additional visualizations will be added as time permits. All visualizations will support zooming, panning, and (for those rendered in 3D) 3-axis rotation. User interaction will be through a PS/2 wheelmouse.

The data logging component will use a SiRF StarIII GPS receiver chip to obtain position information. The StarIII provides position fixes once a second over a TTL-level RS232 interface. Passing through a number of parsing, encoding and control modules, these fixes are written to a 16 megabit serial Flash ROM. Each fix consists of 4 32-bit values (latitude, longitude, altitude, and x/y velocity), making for 128 bits per fix. At one fix per second, this amounts to approximately 35 hours of recording time, which is certainly ample.

The visualization component can be divided into three major parts. The first part is simply the reverse of the mobile data logger, reading position fixes from the Flash ROM and decoding them into a sequence of latitude/longitude/altitude/velocity values.

The second part is made up of the visualization modules. These are responsible for taking the sequence of position fixes and issuing commands to the rendering subsystem (see below) that draw a particular visualization of the data on a monitor. They have access to data packets from the PS/2 mouse, and are responsible for interpreting these packets as rotate/translate/zoom commands. Essentially, these are just FSMs, that move through a fixed procession of states: issuing the commands to set up the background, draw the axis, draw the labels, draw some summary information, and then draw the actual data plots.

The third part, the rendering subsystem, is really the core of the project. It is responsible for performing the coordinate transformations that map latitude, longitude, altitude, velocity and so forth into 2D coordinates on the screen, as well as the actual drawing of graphics on the screen.

The two ZBT RAMs on the labkit are used as video RAM. At any given point, one RAM is designated as 'active', and the image displayed on the monitor is read from that. All rendering operations draw to the inactive RAM. Once a frame is fully rendered, the RAMs switch positions during the vertical blanking interval.

Rendering is performed in a 3-stage pseudo-pipeline, with the stages termed ‘Transformation’, ‘Drawing’ and ‘Coloring’.
The transformation stage is responsible for coordinate transformation, taking latitudes, longitudes and altitudes (or velocities), and transforming them into screen coordinates based on the current view limits and orientation. This stage may be bypassed in order to draw directly to a given set of screen coordinates.

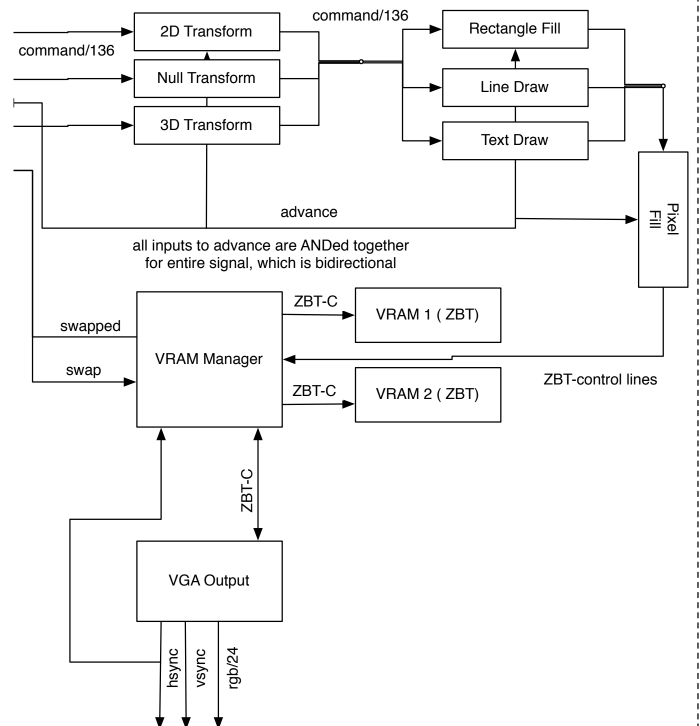


Figure 1: rendering pipeline / subsystem block diagram

The drawing stage provides abstractions that handle specific tasks of actually drawing to the screen: rectangle filling, line drawing, and text drawing (more may be added). As input, it takes the screen coordinates produced by the first stage. In general, a single command given to the drawing stage will be transformed into multiple commands for the subsequent (coloring) stage - drawing a 100-pixel line will require coloring 100 pixels.

The third and final stage, coloring, is responsible for actually interacting with the inactive ZBT RAM, and writing color values to specific pixels. It supports two methods of drawing: standard overwriting, in which a pixel completely overwrites whatever was in its place before, and alpha transparency, in which the color of the new pixel is blended with color of the existing pixel at its location to produce the final value.

The pipeline is controlled using a serial, undirectional bus that carries ‘rendering commands’. These are 128-bit values (the length may be changed) specifying an operation for each stage to perform, and a set of data to operate on. The model is similar to processor bytecode. The exact form of the commands has not been established, but one can imagine having three bits controlling the output of each stage, with the remaining 119 bits for data. For the first stage, 000 might indicate a NOOP or bypass (performing no transformation), 001 might indicate a 2D transformation for four values, 010 might indicate a 3D transformation for six values, 100 might indicate a scaling of the 2D view dimensions, and so forth. For the second stage, 001 might indicate text drawing, 010 rectangle filling, etc.

Commands move through the pipeline sequentially, with the output of each stage being the input of the following stage. Stages may modify the data in commands (i.e. transforming coordinates), or change a command into multiple subcommands (for instance, a line drawing command will be transformed into a number of pixel-coloring commands).

While most commands will result in data being written to the VRAM, some will not. For instance, the 2D and 3D transforms accept commands that set the parameters of the view being rendered. These commands are ignored by the modules in the drawing and coloring stages.

The *advance* signal ties together every component in the pipeline. The *advance* output of every component is AND’d together, and returned to the components as an input. The pipeline only progresses when *advance* is high - if any component brings it low, it stops. This is to allow for the fact that the amount of time the various pipeline stages will take is indeterminate: drawing a 1000-pixel line will take less time than a 10-pixel one. Thus, the drawing stage (for example) can temporarily halt the rest of the pipeline while it issues the appropriate coloring commands.

Note that the pipeline is not constantly running: the active visualization module is responsible for determining whether the currently-rendered image stored in the active VRAM should be replaced. If the user does not touch the mouse or otherwise adjust the view parameters, the pipeline will be idle (that is, executing NOOPS).

The visualization modules and rendering pipeline are coordinated through the rendering manager module. This is essentially a switch, that determines which visualization module is active, and also manages the behavior of the video RAM and position-fix-reading modules on behalf of the visualizations.

The rendering pipeline will be the target of the majority of any additional improvements that are added to the system once the basic functionality is completed: support for 3D surfaces, textures, improved line-drawing, anti-aliasing, and so forth.

MOBILE DATA LOGGER MODULE DESCRIPTIONS

SiRF StarIII GPS Receiver

This is a self-contained GPS receiver chip. Once it has established a satellite lock, it provides position and status information every second over an RS232 interface. The information provided includes latitude, longitude, altitude, velocity, heading, the current time, the number of satellites in view, and so forth.

Communication with the StarIII over RS232 runs at 9600 baud.

StarIII Initializer

The StarIII's behavior is mostly automatic, however it does require some initial configuration (setting the update period, communication mode, and so forth). The initializer module performs this configuration over the RS232 interface in the cycles following a reset.

This module will be tested live against the StarIII chip directly, in combination with the RS232 output module.

RS232 Input

The RS232 input module handles the low-level details of receiving data over RS232. At every clock cycle, it produces a nine-bit output. The first bit indicates whether there is a valid byte of data available, and the remaining eight bits comprise that byte.

This module will be tested live, against the StarIII chip.

RS232 Output

The RS232 output module handles the low-level details of sending data over RS232. It takes a nine-bit input: the first bit indicates that the remaining eight bits contain a byte that should be sent. The *ready* output indicates that the module is capable of receiving bytes to send (i.e. it is not in the process of sending a byte already).

This module will be tested live, against the StarIII chip, in combination with the initializer.

SiRF Message Parser

The message parser module receives messages from the SiRF chip byte-by-byte through the RS232 module, identifies the messages containing position information, and parses them. When it has a message available of the desired type, it raises *available* high and places the message's longitude, latitude, altitude, timestamp, and velocity values on the *fix* output. Note that the contents of *fix* are subject to change.

This module will be tested in simulation, in isolation.

Data Log Encoder
The data log encoder module takes position fixes (arriving once a second) from the message parser, and converts them into log messages to store in the Flash ROM. It is also responsible for erasing the ROM (through the ROM controller) at initialization, and writing a termination record to the ROM before shutdown.

The data log format has not been determined, but it will likely consist of an initialization record (containing a magic number and initial timestamp), plus an indefinite number of data records, plus a termination record (containing another magic number).

The data records will store latitude, longitude, altitude, and velocity. Each value is 4 bytes long, making for a total data record length of 16 bytes, or 128 bits. Using a 16 megabit Flash ROM, this leaves room for 125,000 samples, or (at one sample per second) approximately 35 hours of logged data.

This module will be tested in simulation, in isolation.

Flash ROM Output Controller
The ROM controller is responsible for writing data to the Flash ROM module. The ROM is a serial device, with two inputs (clock and serial-in), and one output (serial-out).

The ROM controller raises *ready* high when it is able to receive commands (i.e. when it is not performing some other action). If *erase* is high at the rising edge of the clock, it will erase the ROM. If *write* is high, it will write the bye on the *data* input to the next available address (starting at 0 and incrementing automatically).

To save energy, the controller may cache write requests and perform them in blocks, leaving the ROM in power-save mode when it is not being written to.

This module will be tested live against the Flash ROM module, using a custom circuit, alongside the Flash ROM Input Controller (see below).

Serial Flash ROM
The serial flash module is a Micron M25P16. It has a 16 megabit capacity. All communication is performed over a serial (SPI) interface, at up to 50 MHz. The interface consists of three signals: clock, data in, and data out.

DATA VISUALIZATION SYSTEM MODULE DESCRIPTIONS

Serial Flash ROM
This is the same module described above (Micron P25M16).

Flash ROM Input Controller
The ROM input controller module is responsible for reading data from the serial Flash ROM. It presents data one byte at a time, raising *available* high when there is a byte available, and placing the byte's value on the *data* line. When the *next* signal is driven high, it reads the byte at the next address. The *restart* signal forces it to begin reading at address zero.

This module will be tested live, against the Flash ROM chip, in combination with the ROM Output controller (see above).

Data Log Decoder
The data log decoder is responsible for parsing the bytes it receives from the ROM input controller into position fixes. Each fix is a 129-bit value, consisting of the following values (listed MSB first):

1 bit: EOF indicator. If 1, indicates that the *previous* fix was the last recorded. All other values in the fix are undefined if this bit is 1.
32 bits: Latitude
32 bits: Longitude
32 bits: Altitude
32 bits: Velocity

The *available* output indicates whether a fix is available. The *next* input indicates whether to read the next fix. The *restart* input indicates that reading is restarting from the beginning of the ROM.

The decoder will be tested in simulation, in isolation.

Fix Queue
The fix queue maintains a FIFO buffer of position fixes, in order to maintain a constant supply of them for the rendering modules. The optimal size of the queue will be determined experimentally, but a length 128 fixes will be used initially, making for a total queue size of 16 kilobits, or just under one BRAM on the Virtex-II.

The *empty* output indicates that the queue is empty, and the *read* input requests that a new fix be popped off the queue. Fix values are placed on the *fix* output.

The queue will be tested in simulation, in isolation.

IMPS/2 Decoder
This module is responsible for communicating with an attached mouse via the Intellimouse PS/2 protocol, and providing data packets indicating mouse movement to the rest of the system. The *available* output indicates that a packet is available, and each packet consists of *lmr,* a 3-bit signal indicating whether the left, middle or right mouse buttons are pressed, as well as 8-bit *x, y,* and *z* signals indicating the movement of mouse and mouse wheel.

The module will be tested live, with an actual mouse.

Visualization Modules
These modules will be responsible for issuing commands to the rendering pipeline that draw their respective visualization to the screen. Very little computation takes places in these, as that is handled by the transform stage of the pipeline.

The 4-bit *active* signal indicates which module is active - each module recognizes only one *active* value, and remains idle if it is not seen.

As input, the modules take logged position fixes from the rendering manager, as well as mouse data packets from the IMPS/2 decoder. The modules are responsible for notifying the rendering manager via pipeline commands when they wish to begin rendering a scene (because the user has changed the view parameters), and when they have finished.

Optimally, the modules will output one command per clock cycle. Commands that are always issued (i.e. to set up axes and labeling) can be stored in a ROM.

These modules will be tested live, integrated into the whole system.

Rendering Manager
The rendering manager ties the rendering subsystem together, as you might expect. It can be considered part of the pipeline itself, because it responds to commands.

Essentially, it's just a switch, routing signals from the active visualization module to the pipeline and the fix queue. It is also responsible for selecting the active visualization module, which at the moment will be performed based on mouse movement.

The two commands it responds to are 'Re-render', which is issued by a visualization module when it wants to re-render a scene, and 'Render complete', which is issued by a visualization module when it has finished rendering. Upon receiving a re-render signal, the rendering manager raises *restart* high in order to begin restart reading fixes from the Flash ROM. Upon receiving a render complete signal, it raises *swap* high to tell the VRAM manager to switch the active and rendering memories.

It will be tested in simulation, in isolation.

2D and 3D Transform
The transform modules transform real-world coordinates (latitude and longitude, plus altitude or velocity) into screen (pixel) coordinates. This will involve a fair amount of numerical computation (multiplication, division, matrix multiplication, etc.). They will need to support transforming four values at once, to allow for the rectangle fill operation (x, y, height, and width). Their performance is not critical, but the faster they can operate, the faster the screen can be redrawn.

The dimensions that the modules are transforming to (i.e. the range of latitudes and longitudes that the screen covers) can be set by issuing a command.

They will be tested in simulation, in isolation.

Null Transform
The null transform module performs no transformation between coordinates: it simply relays the values it receives to the next stage in the pipeline. This allows the visualization modules to draw to absolute pixel locations directly.

It will be tested in simulation, in isolation.

Rectangle Fill
The rectangle fill module fills a specified rectangle on the screen with a given color, by issuing pixel fill commands.

It will be tested in simulation, in isolation.

Line Draw
The line draw module draws a line on the screen, of a specified color, between two pixel coordinates. The rasterization algorithm that will be used has not been determined.

It will be tested in simulation, in isolation.

Text Draw
The text draw module draws a specified string of text to the screen, using an internal ROM to lookup character bitmaps.

It will be tested in simulation, in isolation.

Pixel Fill
The pixel fill module fills specified pixels on the screen with a specified color, and may optionally use alpha-blending to produce a transparency effect. The pixel fill module interacts directly with the rendering ZBT RAM.

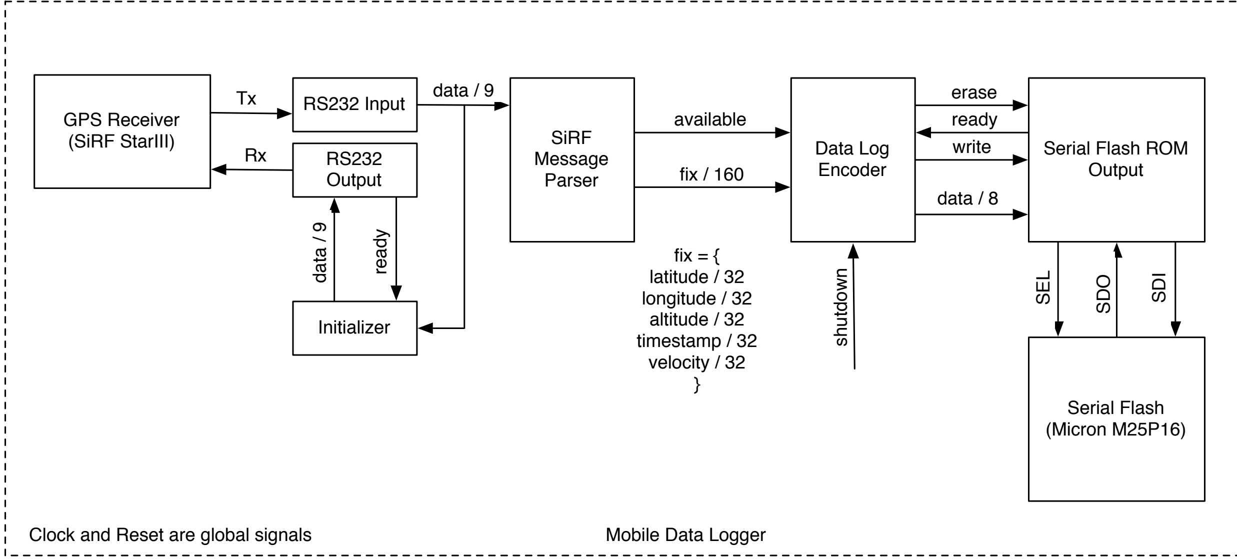
It will be tested in simulation, in isolation.

VRAM Manager
The VRAM manager is the gateway between the ZBT RAMs, the rendering pipeline, and the VGA output module. At any given point, one of the RAMs is designated as ‘active’, providing data for VGA output, and the other is designated as ‘inactive’, being written to by the graphics pipeline.

The pipeline and VGA module communicate with the ZBT RAMs through this module. When the *swap* input goes high, the VRAM manager waits until *vsync* goes low (indicating that we have entered the blanking interval between screen refreshes), and swaps the roles of the RAMS, making the active RAM the inactive RAM, and vice-versa. Once this happens, it raises the *swapped* output high for one clock cycle.

The VRAM manager will be tested in simulation, in isolation.

VGA Output
The VGA output module simply reads pixels from the active VRAM, and sends the appropriate signals through the VGA connector to display them on a monitor.



Serial Flash (Micron M25P16)

SEL SDI SDO

Serial Flash ROM Input

available

data / 8

next

Data Log Decoder

next

available

fix / 129

Fix Queue

fix is as above, but w/o timestamp, an with extra bit indicating EOF

restart

fix / 129 empty read

Altitude vs. time

Velocity vs. time

2D Position

2D Position + Altitude (3D)

2D Position + Velocity (3D)

active / 4

command/128

advance

fix / 129

available

Rendering Manager

command/136

2D Transform

Null Transform

3D Transform

command/136

Rectangle Fill

Line Draw

Text Draw

Pixel Fill

advance

all inputs to advance are ANDed together for entire signal, which is bidirectional

command is a tristate bus

available lmr/3 x/8 y/8 z/8

IMPS/2 Decoder

ps/2 data ps/2 clk

swapped

swap

VRAM Manager

ZBT-C

VRAM 1 (ZBT)

ZBT-C

VRAM 2 (ZBT)

ZBT-control lines

ZBT-C

VGA Output

hsync vsync rgb/24

Clock and Reset are global signals

Data Display