# Two Enhancements for Native Inferno

*Brian L. Stuart*
*University of Memphis*
*blstuart@bellsouth.net*

*ABSTRACT*

Here I report on recent work done in the process of moving native Inferno to a tablet PC. One part of the work was the porting of Francisco J. Ballesteros's new Plan9 USB support to Inferno. The other was a VGA driver that operates with the 640x480x4 mode common to all VGA controllers.

## Introduction

Recently, I worked on getting native Inferno to run on a tablet PC on loan from the vendor. This ruggedized unit had a camera/2D barcode reader, a touch panel, Ethernet, and 802.11. What it did not have was PS/2 keyboard or mouse interfaces. Unable to get the legacy support working and knowing that USB support would probably be needed anyway, I ported to Inferno the new Plan9 USB support from Franciso J. Ballesteros[1]. After getting it running in a text-only mode, it was natural to expect some form of graphics, particularly when the touchpad was being used. After a little searching, it became clear that it was possible to write a driver that handled the VGA features that are common to all controllers with only a moderate amount of effort. Here are the results of those efforts.

## USB

There are four main components of the Plan9 USB support: the [uoe]hci drivers, a USB support library, the usbd daemon, and the drivers for individual USB devices.

### USB HCI Drivers

Only minimal changes were required to port the files usbohci.c, usbuhci.c, and usbehci.c to Inferno. One change was in the calls to *kproc*(10). Inferno's version takes one more argument than Plan9's. The EHCI driver uses clink as a structure member name. However clink is a macro in Inferno. The OHCI driver required a small change to the initialization sequence on the system where OHCI testing was done. Without this change, if no device was plugged into some port when the system came up, it would never detect one inserted after the system was up. The biggest change to these drivers was moving some of the *ilock*(10) and *iunlock*(10) calls around a bit. In particular, *wakeup*(10) could not be called while an ilock was held. Otherwise, there was a good chance the system would panic when an attempt was made to acquire an already-held ilock.

### USB Support Library

The most obvious change in porting the USB support library is the translation of it from C to Limbo. One major change in functionality was introduced, however. In particular the USB library provides support for serving a small file tree separate from the files provided by the HCI drivers. This functionality was dropped, with the plan to implement it later if needed. Otherwise, the library was a pretty straightforward merging of the C source files, translating them into one Limbo file. The resulting library is /dis/lib/usb/usb.dis.

### USB Daemon

As with the library, the daemon port consisted of merging and translating the C files for usbd into Limbo. However, the loading of the device database and the starting of child driver processes was modified. Specifically, the existing Inferno device database format and calling sequence for child drivers was retained. As a consequence of the removal of file server support in the USB support library, the file /dev/usbdctl was not implemented.

### USB Device Drivers

The original Inferno USB daemon took advantage of the nature of Inferno modules to implement the drivers for individual devices attached to USB ports. All such drivers have a common initialization entry point. When a newly inserted device is recognized, the relevant driver module is loaded and a new process is spawned on that driver's initialization function. To date, three such drivers have been implemented.

#### Keyboard Driver

This driver borrowed elements from several sources including the original Inferno USB keyboard driver, the Inferno native PS/2 keyboard driver, and the new Plan9 USB keyboard driver. It handles a basic ASCII keyboard with repeat. The implementation includes the function keys, the arrow keys, and the home, end, page up, and page down keys.

#### Mouse Driver

The mouse driver takes USB mouse messages and translates them into the form necessary to write them into /dev/pointer. To support this, devpointer.c was modified to allow more than one process to have /dev/pointer open at a time.

#### Mass Storage Driver

Much of the original Inferno mass storage driver was kept in this implementation. The driver has been tested on a USB memory stick and on a USB-connected CD-ROM drive.

## Baseline VGA Driver

The world of PC video controllers is notorious for its complex and often undocumented register interfaces. Identifying which controller is present, then applying the right magic formula of settings to get it into a particular mode is nightmarish. However, practically all VGA controllers implement the same interface as the original IBM VGA controller. That controller's two most useful modes were a 320x200x8 mode and a 640x480x4 mode. Although woefully inadequate for use on a desk top or most laptops, they are not entirely unreasonable for use in a touchpanel-based application, where the minimum size of objects is limited by the size of a person's finger. Furthermore, the 640x480 mode covers 63% of the width and 80% of the height of a typical 1024x600 netbook screen. Therefore, although no substitute for full VGA support, there is enough utility in implementing a common denominator VGA driver to invest the effort.

The driver vgabase.c supports these two video modes on all VGA controllers it's been tried on, including that of qemu. It provides the enable, drawinit, and flush functions of the VGAdev structure. It also provides the enable, load, and move functions of the VGAcur structure, implementing a software cursor. (The original VGA controller did not have a hardware graphics cursor.)

The driver itself handles the appropriate register initialization when the drawinit command is given to /dev/vgactl. Prior to that the resolution and depth were written into the VGAscr structure, and they are referenced to determine which of the two supported modes is required. Getting the exact register setting is a little tricky because the original IBM documentation was vague on a few points. However, there are a number of resources out there that help fill in the details, most notably the svgalib implementation. In the case of 640x480x4, the default colormap doesn't work very well, so the driver overrides it with a fixed colormap. It is possible to assign some colors along with shades of gray that are mostly acceptable with existing Inferno applications, a straight 16-level grayscale colormap is often preferred. Furthermore, the colormap can be overridden by an application program.

Implementing the flush operation is an exercise in deciphering the memory layout details for the frame buffer. In the 320x200x8 mode, things are much as one would expect. A block of the address space is assigned to the frame buffer and each pixel is stored in the frame buffer as a one-byte index into the colormap. The 640x480x4 mode is another matter entirely. In this mode there are four planes of memory each laid out as a 640x480x1 screen. Each pixel is made from one bit of each of these planes. Although a little messy, this by itself wouldn't be too bad. However, all four planes share the same address space, and other controller registers determine how each byte of data written is applied to the four planes. The way it gets handled in vgabase.c is to split the screen image out into the four bit-planes. Then for each row, the controller is programmed to load each plane in sequence and the row is written to that plane.

As mentioned above, the graphics cursor is implemented in software. Each time the cursor position is set, the contents of the screen are read and the and/or operations for the cursor are applied before the data is written back. To keep it updated, there is a 50mS timer that checks to see if flush has overwritten any part of the cursor, and if so repaints it.

## Future Work

The first avenue for further work is adding support for other types of USB devices. The Plan9 support includes support for Ethernet, audio, printer, and serial devices, none of which have been included in the Inferno support reported here. Also little testing has been done on the OHCI and UHCI drivers, so there's a good chance that some issues will arise.

For the VGA driver, the primary opportunity for improvement is performance. On slower machines, painting a screen in wm/man is noticeably slow. Whether more performance can be found is uncertain, but in some settings more is needed.

## References

[1] Ballesteros, F J, "Plan 9's Universal Serial Bus," Proceeding of the 4th International Workshop on Plan9, 2009.