# Usb serial design and experience in Plan 9

Gorka Guardiola Múzquiz Francisco J Ballesteros Enrique Soriano Salvador Laboratorio de Sistemas Universidad Rey Juan Carlos

### **ABSTRACT**

Most new desktop computers do not come with a serial uart. Embedded ones sometimes do and it is indispensable as an aid for debugging them, specially when porting an operating system like Plan 9. Serial USB devices are sometimes a way out of this conundrum. Some devices like the Sheevaplug even include a serial to usb inside as part of the package. Four thousand lines of code, and a change in libthread later Plan 9 supports three of the most popular chipsets. Getting a serial console in a new machine in Plan 9 is now hopefully easier. This article describes the USB serial architecture on Plan 9, the rationale behind the design decisions made when designing it and our experience building it.

### Introduction

New computers, embedded or not, do not come with an external serial uart port, which is an essential aid for debugging a kernel. On them, the only alternatives available which do not require at least some amount of opening the box and soldering (on some machines there are secret hidden serial ports) is by using the USB ports of the machine. The Universal Serial Bus (or USB) [2] protocol stack for Plan 9 has been completely rewritten recently. The new stack described in [1] greatly simplifies the writing of USB drivers, completely abstracting USB transactions and converting them in reads and writes to files representing endpoints of the device. So we wrote some drivers for the different alternatives available and integrated them with a uniform filesystem interface similar to the one offered by the regular serial drivers in the kernel.

#### USB as an alternative for a serial uart

When there is a serial uart (and port) on the machine, printing something can be as simple as writing to a couple of registers, provided that the uart comes already configured by the BIOS or uboot or whatever software loader booted the computer. When using the uart in polling mode, there is no need to have interrupts or the clock working both of which are an important part of the porting process. We would like to have this kind of benefits but using a USB port.

When trying to get a console through a USB port there are several alternatives available. The first one is the EHCI debug port, described in the Appendix C of [6]. This is Intel's official substitute for the serial port. It has the advantage that the machine to be debugged can use it at boot time because it does not use USB in the normal way, which

is too complex and needs the kernel to be at least at the stage where it can run processes. This device would be (if it worked as advertised) much closer to a serial port. We implemented the host side of this device as part of the kernel USB driver and the client side in user space as part of the USB serial stack. This implementation is integrated in the regular usb serial stack, but it loses bytes. We implemented in an early version of the USB serial and we have not continued using it, so it needs some extra work. The reason we did not continue using it, even though it looked like an promising solution, is because it has too many shortcomings. First of all, there can be no other devices conected to the USB host if it is being used as a debug port. In a modern machine this means no keyboard and no mouse. Secondly, not all EHCI hosts support it and many machines only have an OHCI [3] host. All these shortcomings make it impractical to use and a poor substitute to a serial uart in many cases.

Another alternative is also a USB serial communication standard [5] which apparently is used in many phones and modems but we have not encountered in machines or in serial cables. Like any normal usb device with no special compatibility mode (like the mouse or the keyboard) it is probably too complex to use it at boot time. This standard is currently unsupported in Plan 9.

The last alternative for getting a serial console is to use something like CEC [4] which is also impractical to use at boot time, because it needs a working ethernet driver which means again having working interrupts and clock and the complex register interface which comprises the ethernet driver itself.

Seeing this absence of solutions, many vendors have started making either serial to USB chips (with non-standard chipsets inside) and integrating them inside machines and devices to expose a serial console through a USB port. By this we mean that they take their secret serial uart and port, and plug to it a serial to usb converter that acts as a usb device when plugged on a USB host. The two most popular of these chips are the FTDI and Prolific chipsets in their different configurations. Both are supported in Plan 9. The Prolific chip is found mostly inside serial to USB cables and the FTDI is integrated in all sorts of devices.

### Architecture

Vendors tend to add all sorts of bells and whistles to their USB serial devices, but in the end there is only so much you can do in a serial communication. Taking this into account, we decided to keep all the USB console drivers inside the same program in order to keep ourselves from repeating all the common parts, like the filesystem and the argument processing. We have tried to support roughly the intersection of the features provided by all the vendors and the user interface (by which we mean the filesystem interface) as simple as possible. This has kept us from the temptation of adding extra features.

All the devices are very similar and have at least two data endpoints (one IN and one OUT or one IN/OUT) to push data data back and forth, an optional IN interrupt endpoint for informing about changes on the extra control signals and the control endpoint which is used for setting the values of this control signals.

The common infrastructure is contained in the files main.c and serial.[ch]. The main.c file is just the USB stack boilerplate main function in case the program is used standalone instead of integrated inside usbd(4). The serial.[ch] files make up the common infrastructure comprising the detection of the device and the definition of the data structures. These are Serial which represents the device, Serialport,

representing each of the ports and Serialops, function pointer interface. The rest of the files are concrete implementations of this interface one per chip.

Devices identifying themselves as the same chip can have different endpoints and interface, so the configuration of the device has to be done carefully. Specially since we want to fire independent processes per port to take care of each port independently. We pushed as much as we could of the detection process to the common code, but for the FTDI we needed to know the type of the chip before we could set up the configuration which means filling up the Serial and Serialport structures. The main problem is that in the FTDI device there are extra endpoints which correspond to the JTag interface. But which endpoint correspond to which JTag depends on the model of FTDI chip, which can only be known after poking into it. Because of this, we added a match function to the interface which is called per device and which can be used to fill the description in Serial to describe the device completely. After completely matching the device the different processes needed for the device can be run.

# Concurrency

To keep things simple, we used a single qlock(2) per device to ensure mutual exclusion in the access to the shared data. Reads and writes to the data endpoints are done in the flow of execution of the process attending the client (created as part of the USB library filesystem infrastructure). A different process needs to be created to read the interrupt endpoint to see if there is any change in the control lines and update the the values for the filesystem clients. This updates may be be periodical depending on the device. The FTDI chip makes things a little more interesting, because control signal states is sent and received in-band as a header to the serial data. This makes the concurrency model more complicated, because when you read to update the signals, you may get data also which you have to buffer until you get a client reading from the data file. We used a buffered channel for this, which also decouples the signal updater from the process taking care of the clients. The process taking care of the clients reads from the buffered channel and serves an RPC over a couple of channels for clients wanting data. This process ensures mutual exclusion on the data which may have to be recompacted if the client asks for data which spans more than one read, so it is copied contiguosly to a buffer and pointers to it send to the clients as they come. The only problem with this approach is that cleaning up if there is an error or the device disappears quite complicated.

## Cleaning up

Whenever there are problems reading from an endpoint there are different approaches which can be taken, some chip dependant and some more drastical than others. The chip may provide a way to reset it, there may be a way to reset just one port on the chip and there is a way to reset the whole usb device. Also, errors come in different flavours, the device may stop responding or signal that it has been detached. Whenever there is an error of any kind, serialrecover is called. This function tries to recover from the error or signals that it is fatal otherwise. At the moment, when there is an error we try to find if it is because the device was unplugged, if so, we just initiate cleanup. Otherwise we drain all the data in the pipes or buffers of the device and try to reset everything inside the chip to keep things simple. This approach probably needs some rework. If the device does not work after this, we detach it and close all the associated channels. Detaching the device takes care of all the processes blocked on USB files and closing the channels does likewise with the processes blocked sending or receiving on them. At first, we tried other approaches with exiting instead of using *chanclose*(2) (which did not

exist at the time). Using signals to unblock processes from channels had subtle race conditions when freeing the data and using read-write locks or extra reference counted structures (the main ones are already reference counted) got things to be even more complicated. In the end we implemented close for channels, which has simplified enormously the cleanup dance. Chanclose unblocks any processes blocked on a channel and prevents new processes from blocking in them. The return value of close signals an error or unblocking because of a signal. We do not use signals on this program, so we consider a signal an error too, so there is no ambiguity here. If there was, we could have used *chanclosing*(2) to resolve it. *Chanclose*(2) and *chanclosing*(2) are both now a standard part of the *thread*(2) library in Plan 9.

### Naming

The USB library eases the writing of filesystems, and enables to write fast a filesystem to serve a directory in a way which is compatible with usbd. We decided serve one directory per device with two files inside per port. We took this as a chance to simplify the standard serial interface in Plan 9, which had a data file eiaN and two control files, eiaNstatus eiaNctl, which is unnecesary and an old remnant of an interface nobody uses any more. EiaNctl, gives back when read from a number which is useless and eiaNstatus is just used for reading. Both can be merged into one file, so we reduced the interface to only two files, one for the data and one for control. Other than this, the interface is very similar to any other serial port. The next decision we had to take was how to name the files. The first name coming to mind, eiU0/eiaU0.4ctl the zero meaning the zeroth device and the fourth port seemed redundant. On the other side if we want to bind the directory in /dev we want each file to have a unique name. The issue is not completely resolved yet and at the time and the names may still change in the close future. Another problem we originally had was that the device numbers were taken sequentially by the serial driver and never reused to ensure we did not run into the same number again. This complicated all sort of scripts. To better the situation, we started reusing numbers which had stopped being used. Other drivers ran into this problem and finally the library was changed and a unique device is given to the device with a counter which is different per type of device. This makes knowing what USB device you are talking to more predictable and simple.

Another naming issue is the problem of where to mount the filesystem served by the driver. This is a general issue with USB drivers. If the filesystem is mounted in /dev and the driver exits, it might leave /dev unusable which is not good. This is not a problem if the device is embedded inside usbd because it takes care of the root directory but the mount point may get stale if the program is running standalone. To prevent this problem we mount the tree in /n in that case.

### Performance problems

Whenever reading or writing to an endpoint, a number of usb packets are sent. The last packet is smaller than the maximum packet to signal the delimiter of the transaction. These delimiters are respected by the USB drivers and signaled by a short read. Some hardware has problems if an amount greater than the maximum packet is sent at once, so at first we tried to make all the writes to be smaller than the maximum packet size, which is something between 8 and 256 bytes. We figured that the hardware was slow enough that the loss of performance due to making so many system calls would not matter and we could keep things simple. We were dead wrong. The hardware, specially in the case of the FTDI is very fast and doing things this way, we cannot keep up and

lose bytes. We have to be able to read at once something of the order of kilobytes and be careful to have some buffering to be able to keep up with the data bursts.

#### Related work

To our knowledge, the Plan 9 implementation of the usb serial is one of simplest ones out there. In the usb serial drivers we have come accross, mainly from Linux, BSD or Mac OS the line discipline is mixed in with the driver and the USB transactions have to be taken care of by hand (this last part is a nice property of the Plan 9 USB stack not the USB serial driver). Having the possibility of running the drivers standalone as user programs and the way the drivers fit easily into the architecture make programming and debugging them dead easy.

#### Future work

More work needs to go into dealing in a more gentle way with errors so that even ports in the same device can be kept working if some of them have failed. Also, more testing needs to be done for different models of the devices, specially with multiport, which is completely untested at the moment. Last, support for te cdc-acm standard needs to be added. As with the other devices, to accomplish this, the Linux [FreeBSD [7] the standard already mentioned and any documentation which can be obtained from the vendor can be used as a guide. We recommend starting from be FreeBSD implementation, moving on to the Linux one and only then reading any documentation. Reading any USB documentation without any previous experience can be very overwhelming, disorienting and discouraging.

#### References

- 1. F. J. Ballesteros, Plan 9's Universal Serial Bus, IWP9, 2009.
- 2. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC and Philips, USB 2.0 Specification, 2000.
- 3. Compaq, Microsoft and N. Semiconductor, OpenHCI Open Host Controller Interface Specification for USB, 1995.
- 4. Coraid, Coraid Ethernet Console, 2010.
- 5. U. I. Forum, Universal Serial Bus Class Definitions for Communication Devices, 1999.
- 6. Intel, Enhanced Host Controller Interface Specification for Universal Serial Bus, Revision 1.0, 2002.
- 7. F. reference, FreeBSD cdc-acm driver, 2010.