NAME

ppbus - Parallel Port Bus system

SYNOPSIS

device ppbus

device lpt
device plip
device ppi
device pps
device lpbb

DESCRIPTION

The *ppbus* system provides a uniform, modular and architecture-independent system for the implementation of drivers to control various parallel devices, and to utilize different parallel port chipsets.

DEVICE DRIVERS

In order to write new drivers or port existing drivers, the ppbus system provides the following facilities:

- architecture-independent macros or functions to access parallel ports
- mechanism to allow various devices to share the same parallel port
- a user interface named ppi(4) that allows parallel port access from outside the kernel without conflicting with kernel-in drivers.

Developing new drivers

The ppbus system has been designed to support the development of standard and non-standard software:

Driver	*Description*
ppi	Parallel port interface for general I/O
pps	Pulse per second Timing Interface
lpbb	Philips official parallel port I2C bit-banging interface

Porting existing drivers

Another approach to the ppbus system is to port existing drivers. Various drivers have already been ported:

Driver	*Description*

lpt lpt printer driver
plip lp parallel network interface driver

ppbus should let you port any other software even from other operating systems that provide similar services.

PARALLEL PORT CHIPSETS

Parallel port chipset support is provided by ppc(4).

The ppbus system provides functions and macros to allocate a new parallel port bus, then initialize it and upper peripheral device drivers.

ppc makes chipset detection and initialization and then calls ppbus attach functions to initialize the ppbus system.

PARALLEL PORT MODEL

The logical parallel port model chosen for the ppbus system is the PC's parallel port model. Consequently, for the i386 implementation of ppbus, most of the services provided by ppc are macros for inb() and outb() calls. But, for another architecture, accesses to one of our logical registers (data, status, control...) may require more than one I/O access.

Description

The parallel port may operate in the following modes:

- compatible mode, also called Centronics mode
- bidirectional 8/4-bits mode, also called NIBBLE mode
- byte mode, also called PS/2 mode
- Extended Capability Port mode, ECP
- Enhanced Parallel Port mode, EPP
- mixed ECP+EPP or ECP+PS/2 modes

Compatible mode

This mode defines the protocol used by most PCs to transfer data to a printer. In this mode, data is placed on the port's data lines, the printer status is checked for no errors and that it is not busy, and then a data Strobe is generated by the software to clock the data to the printer.

Many I/O controllers have implemented a mode that uses a FIFO buffer to transfer data with the Compatibility mode protocol. This mode is referred to as "Fast Centronics" or "Parallel Port FIFO mode".

Bidirectional mode

The NIBBLE mode is the most common way to get reverse channel data from a printer or peripheral. Combined with the standard host to printer mode, it provides a complete bidirectional channel.

In this mode, outputs are 8-bits long. Inputs are accomplished by reading 4 of the 8 bits of the status register.

Byte mode

In this mode, the data register is used either for outputs and inputs. Then, any transfer is 8-bits long.

Extended Capability Port mode

The ECP protocol was proposed as an advanced mode for communication with printer and scanner type peripherals. Like the EPP protocol, ECP mode provides for a high performance bidirectional communication path between the host adapter and the peripheral.

ECP protocol features include:

Run_Length_Encoding (RLE) data compression for host adapters

FIFOs for both the forward and reverse channels

DMA as well as programmed I/O for the host register interface.

Enhanced Parallel Port mode

The EPP protocol was originally developed as a means to provide a high performance parallel port link that would still be compatible with the standard parallel port.

The EPP mode has two types of cycle: address and data. What makes the difference at hardware level is the strobe of the byte placed on the data lines. Data are strobed with nAutofeed, addresses are strobed with nSelectin signals.

A particularity of the ISA implementation of the EPP protocol is that an EPP cycle fits in an ISA cycle. In this fashion, parallel port peripherals can operate at close to the same performance levels as an equivalent ISA plug-in card.

At software level, you may implement the protocol you wish, using data and address cycles as you want.

This is for the IEEE1284 compatible part. Then, peripheral vendors may implement protocol handshake with the following status lines: PError, nFault and Select. Try to know how these lines toggle with your peripheral, allowing the peripheral to request more data, stop the transfer and so on.

At any time, the peripheral may interrupt the host with the nAck signal without disturbing the current transfer.

Mixed modes

Some manufacturers, like SMC, have implemented chipsets that support mixed modes. With such chipsets, mode switching is available at any time by accessing the extended control register.

IEEE1284-1994 Standard

Background

This standard is also named "IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers". It defines a signaling method for asynchronous, fully interlocked, bidirectional parallel communications between hosts and printers or other peripherals. It also specifies a format for a peripheral identification string and a method of returning this string to the host outside of the bidirectional data stream.

This standard is architecture independent and only specifies dialog handshake at signal level. One should refer to architecture specific documentation in order to manipulate machine dependent registers, mapped memory or other methods to control these signals.

The IEEE1284 protocol is fully oriented with all supported parallel port modes. The computer acts as master and the peripheral as slave.

Any transfer is defined as a finite state automaton. It allows software to properly manage the fully interlocked scheme of the signaling method. The compatible mode is supported "as is" without any negotiation because it is compatible. Any other mode must be firstly negotiated by the host to check it is supported by the peripheral, then to enter one of the forward idle states.

At any time, the slave may want to send data to the host. This is only possible from forward idle states (nibble, byte, ecp...). So, the host must have previously negotiated to permit the peripheral to request transfer. Interrupt lines may be dedicated to the requesting signals to prevent time consuming polling methods.

But peripheral requests are only a hint to the master host. If the host accepts the transfer, it must firstly negotiate the reverse mode and then starts the transfer. At any time during reverse transfer, the host may terminate the transfer or the slave may drive wires to signal that no more data is available.

Implementation

IEEE1284 Standard support has been implemented at the top of the ppbus system as a set of procedures that perform high level functions like negotiation, termination, transfer in any mode without bothering you with low level characteristics of the standard.

IEEE1284 interacts with the ppbus system as little as possible. That means you still have to request the ppbus when you want to access it, the negotiate function does not do it for you. And of course, release it later.

ARCHITECTURE

adapter, ppbus and device layers

First, there is the *adapter* layer, the lowest of the ppbus system. It provides chipset abstraction throw a set of low level functions that maps the logical model to the underlying hardware.

Secondly, there is the *ppbus* layer that provides functions to:

1. share the parallel port bus among the daisy-chain like connected devices
2. manage devices linked to ppbus
3. propose an arch-independent interface to access the hardware layer.

Finally, the *device* layer gathers the parallel peripheral device drivers.

Parallel modes management

We have to differentiate operating modes at various ppbus system layers. Actually, ppbus and adapter operating modes on one hands and for each one, current and available modes are separated.

With this level of abstraction a particular chipset may commute from any native mode to any other mode emulated with extended modes without disturbing upper layers. For example, most chipsets support NIBBLE mode as native and emulated with ECP and/or EPP.

This architecture should support IEEE1284-1994 modes.

FEATURES

The boot process

The boot process starts with the probe stage of the ppc(4) driver during ISA bus (PC architecture) initialization. During attachment of the ppc driver, a new ppbus structure is allocated, then probe and attachment for this new bus node are called.

ppbus attachment tries to detect any PnP parallel peripheral (according to *Plug and Play Parallel Port Devices* draft from (c)1993-4 Microsoft Corporation) then probes and attaches known device drivers.

During probe, device drivers are supposed to request the ppbus and try to set their operating mode. This mode will be saved in the context structure and returned each time the driver requests the ppbus.

Bus allocation and interrupts

ppbus allocation is mandatory not to corrupt I/O of other devices. Another usage of ppbus allocation is to reserve the port and receive incoming interrupts.

High level interrupt handlers are connected to the ppbus system thanks to the newbus **BUS_SETUP_INTR**() and **BUS_TEARDOWN_INTR**() functions. But, in order to attach a handler, drivers must own the bus. Consequently, a ppbus request is mandatory in order to call the above functions (see existing drivers for more info). Note that the interrupt handler is automatically released when the ppbus is released.

Microsequences

Microsequences is a general purpose mechanism to allow fast low-level manipulation of the parallel port. Microsequences may be used to do either standard (in IEEE1284 modes) or non-standard transfers. The philosophy of microsequences is to avoid the overhead of the ppbus layer and do most of the job at adapter level.

A microsequence is an array of opcodes and parameters. Each opcode codes an operation (opcodes are described in microseq(9)). Standard I/O operations are implemented at ppbus level whereas basic I/O operations and microseq language are coded at adapter level for efficiency.

SEE ALSO

lpt(4), plip(4), ppc(4), ppi(4)

HISTORY

The **ppbus** manual page first appeared in FreeBSD 3.0.

AUTHORS

This manual page was written by Nicolas Souchu.