Floppy

The Floppy module is a driver for a PC-standard dual-floppy interface, bridged over PCI. It supports up to two devices, on a single cable. The devices may be 5-inch or 3.5-inch disk drives.

Caveat

Although most of this code is quite generic for Floppy devices, the DMA code use for data transfer assumes that the drives are located through a (I440BX) PCI bridge chipset which implements the DMA engine. These interface calls (*pci_dma_start* etc.) could be macro-defined for another API providing the same functionality.

Also, there is no attempt to determine the format of the medium inserted into the drive. The code has only ever been tested against a 1.44MB 3.5-inch floppy.

Process Information

Prototype Name	floppy
Process Priority	driver-level
Process Name	does not matter as long as it matches the URLs used within the system to open the device.

Target File Definitions

FLOPPY_CLEAR_INT	This macro defines the means by which the interrupt handler can clear the pending floppy-drive interrupt.
FLOPPY_DMA_CHANNEL	The DMA channel assigned to the floppy drive by the DMA controller.
FLOPPY_DRIVE_SELA	The selector code for the 'A' drive on the cable (usually 0).
FLOPPY_DRIVE_SELB	The selector code for he 'B' drive on the cable (usually 1).
FLOPPY_IOBASE	The address of the register set for the fdd controller.
FLOPPY_IRQ	The interrupt number generated by the fdd controller.
FLOPPY_TYPES	This is a sequence of four integers. The first and third define the size of the floppy drive in the 'A' and 'B' positions on the cable, one of FLOPPY_TYPE_NONE, FLOPPY_TYPE_3 or FLOPPY_TYPE_5.

The second and fourth numbers define the type of medium used in the drive. Index '0' for TYPE_3 is 1.44MB, the only value which has been tested. To find the other values you will need to look in the source code for the module.

Process Operation

The initialisation routine initialises only the controller portion of the floppy sub-system, since the drives themselves are not accessed until they are used.

Dataflow handling

The operation of the floppy controller is complicated by the drive motor operation. In 'idle' state the drive is powered down. In order to access the medium, the drive must be powered on and recalibrated. The heads must be positioned over the correct track using a seek operation and the data must be read or written. Finally, the drive motor must be turned off. Since a single file usually required multiple operations, the drive motor is left running between operations, and is only turned off after a period of no activity. Also, it is necessary to wait for the motor to come up to speed before accessing the medium. These various transitions are handled by an internal state machine within the driver.

The other complication is that the media are themselves removable, even in mid-operation. Ejecting and inserting media does not always trigger a notification to the driver. To make this problem tractable, the driver assumes that media are *not* removed while the driver motor is running. Further, it assumes that media *may* be removed when the motor is off, and this will not be detected. To implement this mode of operation, the driver generates a *MEDIUM_EJECT event* whenever the drive motor is turned off. Upper layers (usually the DOS filesystem process) should treat this as an indication that the medium may have changed, and flush any cached blocks or directory entries.

The floppy controller maintains a queue of pending operations. Unlike network drivers, all operations to the driver are synchronous (both send and receive requests) and are processed in the order in which they are queued. An operation is started when it is queued to an idle channel. Subsequent requests are queued behind the active request. In the interrupt handler, the active request is completed and the reply returned to the caller. If there is another request in the queue, it is started. Only one operation is handled at a time, since there is only a single DMA channel shared between the two drives o the cable.

For read and write requests, the *b_immed* field of the mblk is used to contain the logical block number of the first block in the request. Internally, all devices use absolute LBA addressing; the block address is converted to CHS before programming the device registers.

Messages

The module has a queue handler and main process. and accepts the messages defined in the *Standard* messageset for dataflows into and out of the driver with the following processing:

CLOSE messages are handled by the main process. It is assumed that the close request is not sent while there are outstanding I/O requests to the driver. Any outstanding event requests are returned and the context information block is freed before the reply is returned to the sender.

EVENT messages are added to the file's event queue, and returned as *MEDIUM_EJECT* events as above.

FETMBLK messages are added to the queue of operations for the appropriate device.

FSLIB_DEV_ATTR messages are handled in the main process. The *port* field in the supplied URL selects the device for which information is to be returned. The *maxblk* and *blksize* parameters for the device are returned in the reply.

FLUSH messages are added to the queue of operations for the appropriate device. When the FLUSH message reaches the head of the queue, it is replied to from the interrupt handler, as all previously-queued operations must have completed by that time.

GETMBLK messages are added to the queue of operations for the appropriate device.

NEWMBLK messages return a buffer, of at least 512 bytes, to the caller from the queue handler.

OPEN messages are handled in the main process. The *port* field of the URL contains the index number of the device (0 for 'A', 1 for 'B'). The process returns a value in the *dest_context* field which must be supplied on subsequent messages.

OUTMBLK messages are added to the queue of operations for the appropriate device.

PACKETR messages are added to the queue of operations for the appropriate device.

PACKETW messages are added to the queue of operations for the appropriate device.

PUTMBLK messages with zero length cause the message's buffer to be freed in the queue handler, otherwise the message is added to the queue of operations for the appropriate device.

RETMBLK messages free the allocated block from within the queue handler and are replied to immediately.