

ROFLBus: A Really Organized, Fast, and Low-Cost Bus

John Lee

6.033 Design Project 1

Liskov/Ferreira TR1 (R07)

March 17, 2005

Contents

1	Design Overview	2
2	Design Description	5
2.1	Physical Addressing	5
2.2	Data Transfer Protocol	6
2.2.1	INFO	6
2.2.2	REQUEST	7
2.2.3	READBLK	8
2.2.4	WRITEBLK	9
2.3	Identification	10
2.4	Initialization and Maintenance	11
2.5	Filesystem API	16
3	Conclusion	19
	Acknowledgments	21

1 Design Overview

We present a single-master bus specification to allow Beta External Devices (BEDs) to communicate with the Beta via a 72-pin interface. The design of the ROFLBus seeks to achieve three primary goals: high throughput for block transfers, low manufacturing costs, and simplicity for the end user. We accomplish these goals with a slot-based bus design that allows for up to 31 devices to be connected at a time. Block data transfers can occur at a rate of one word per clock cycle, regardless of the device's latency. This allows for speeds of up to 40 MB/s at the cost of a slightly high latency. We require no additional hardware to support the operation of the ROFLBus and allow for a simple plug-and-play interface. These properties make the ROFLBus ideal for personal computers, where low cost and high throughput are necessary but the 31 device limit and the high latency are acceptable.

High throughput is achieved with a simple communication protocol that allows for variable-sized block transfers to occur at a rate of one word (32 bits) per clock cycle. The overhead involved in initiating transfers is strongly dependent on the latency of the device. Data is read from low latency devices (such as mice and keyboards) with an overhead of one clock cycle, which is simply the Beta's initial request for data. Devices with higher turn around times (such as hard drives) may have a more significant overhead, but we generally expect that the device's turn around time is much longer than the time spent initiating the transfer, and can usually neglect the extra overhead. The data transfer protocol is described in detail in Section 2.2.

A slight inconvenience of differentiating between requests to low-latency and high-latency devices is that the definition of "low-latency" depends on the clock speed of the Beta. If a user upgrades his Beta, for example, the turn around time of certain devices may exceed the one-clock-cycle threshold and must be treated as a high-latency device. The responsibility of coping with this situation falls on the BED manufacturer: a simple solution would be to place a DIP switch (indicating the device latency) on the hardware and document the clock speeds for which each setting works. However, we do not enforce this because a need for such

Pin	Function
1	Memory Indicator (MIND) (synonymous with MRD[0])
2	n th bit from the end of Memory Read Data (MRD[n])
3	Memory Write Enable (WR)
4	Memory Output Enable (MOE)
5	Power (VDD)
6	Ground (GND)
7	Clock (CLK)
8	Interrupt Request (IRQ)
9-40	Memory Write Data (MWD)
41-72	Memory Read Data (MRD)

Figure 1: Pin assignments for the n th slot. Note that pin 2 is unique to each slot.

a switch may not apply to certain devices.

We emphasize the need for fast block data transfers because of the growing demand for digital multimedia and portable data storage. Small data transfers with high latency devices result in poor performance with the ROFLBus, but if the amount of overhead is small compared to the size of the transferred data, the average transfer rate approaches 40MB/s on a 10MHz processor. We find this trade-off to be favorable for personal computers, since many are used as entertainment hubs that require fast communication with external devices.

To minimize costs to the end user, we require no additional hardware to support the operation of the bus. Because only up to 31 BEDs can be used at a time, device naming is done by simply enumerating each slot, thus eliminating the need for persistent memory, both on the Beta (to remember the devices that have been attached) and on BEDs (to remember which slot it was attached to). As a consequence, there is no confusion when BEDs are transferred between machines, or if one machine has two instances of the same type of BED. We feel that these benefits significantly outweigh the limitation to 31 devices because the upper limit can

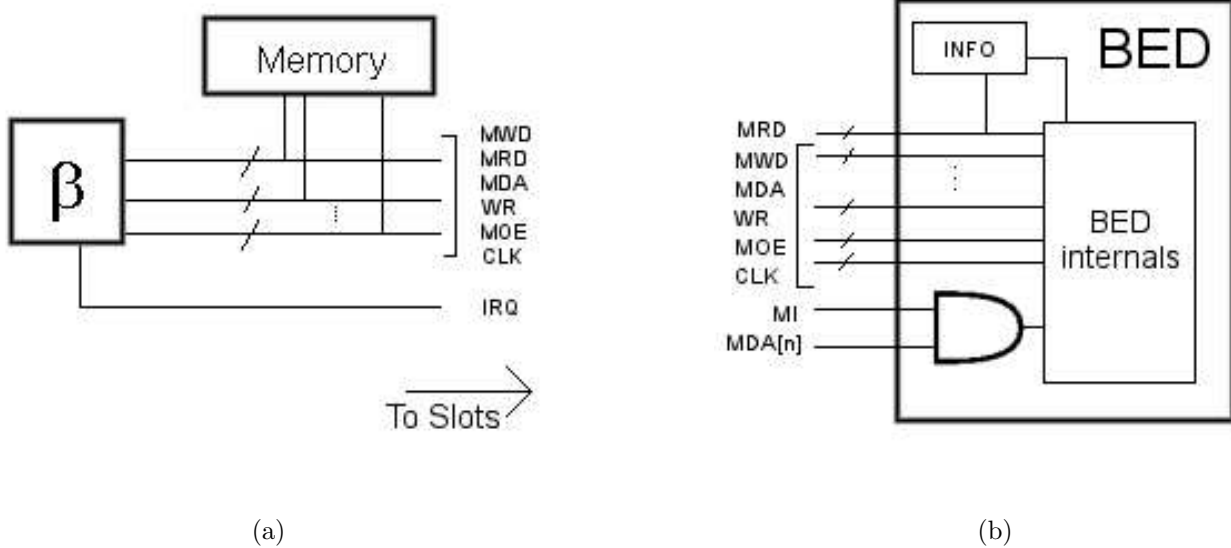


Figure 2: The bus lines and inputs to the BED slots. (a) The outputs of the Beta. (b) The inputs to the n th slot and a suggested BED design: to quickly respond to certain requests by the Beta, this particular implementation stores some identification information in a separate ROM.

still easily accommodate most users.

Because the slots are enumerated, filesystem device names are simple: the n th slot has a device name `/dev/bedn`. Thus, device names are preserved across reboots and will always have the same filesystem name as long as they are not physically moved to a different slot. While we do not protect against accidental physical slot switches, we expect this not to be a problem: devices that a user expects to always have the same name (such as hard drives or video cards) will generally not be moved after it has been installed. An advantage of this design over a bus that does not enumerate slots is that no persistent memory is required to store names assigned to devices.

To properly handle insertion, initialization, and removal of devices, the Beta polls the bus lines frequently to learn the state of the devices on the ROFLBus. We choose this design because it requires no extra hardware to maintain the state of the devices. As a result, our design has a much higher latency for detecting new and removed devices, but we find this to

be acceptable because most devices are not repeatedly inserted and removed from the bus in a short period of time.

2 Design Description

As described in Section 1, the ROFLBus is a slot-based design which allows for up to 31 devices to be connected at a time. The Beta initiates all transactions, thereby eliminating contention for the state of the bus lines by multiple BEDs. If a BED requires attention, it must set the Interrupt Request (IRQ) line to HIGH and wait for the Beta to address it. Each slot has 72 pins, which are assigned as shown in Figure 1.

2.1 Physical Addressing

A BED address is 32 bits long and contains exactly two 1s. The first bit, which we refer to as the memory indicator (MIND) bit, is 1 for all BEDs and 0 for references to data memory. The position of the other 1 in the string indicates the slot number: the n th bit from the end is 1 for the device in the n th slot. When the Beta addresses a device on the ROFLBus, it places this string on the Memory Data Address (MDA) lines.

Each slot has two address pins: one is the MIND bit, which is always the first bit of MDA, and the other is one of the lines of MDA. Thus, BEDs need not know what their own addresses are: a BED knows that the Beta is addressing it when both the MIND bit and the corresponding slot bit are set. Although the size of the data memory's address space is halved due to the reservation of MIND, we favor this design because of the resulting simplicity for BED manufacturers: devices do not need additional memory or logic to store and manipulate address information.

2.2 Data Transfer Protocol

All bus transactions begin with a 32-bit command sent by the Beta via Memory Write Data (MWD). There are four types of commands, which are encoded in the first two bits (prefix) of MWD: `INFO`, `REQUEST`, `READBLK`, and `WRITEBLK`. The remaining 30 bits are used as arguments for these commands. The response that the Beta expects from the BED varies according to which command was sent. The `INFO` command extracts information about the hardware; `READBLK` reads a certain number of words (specified in the argument) from the BED, one per clock cycle; `REQUEST` sends a query for a number of words but does not wait for a reply; `WRITEBLK` sends a number of words (one per cycle) to the BED.

Command	Prefix	Argument
<code>INFO</code>	00	Unused
<code>REQUEST</code>	01	Number of words requested
<code>READBLK</code>	10	Number of words expected on MRD
<code>WRITEBLK</code>	11	Number of words that will be sent on MWD

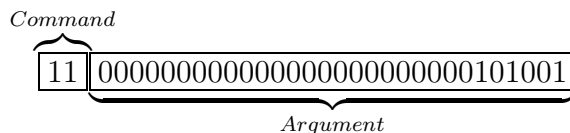


Figure 3: Possible commands and prefixes. Shown above is an example of a `WRITEBLK` command, sent over MWD to notify a BED that 41 clock cycles' worth of 32-bit words follow.

2.2.1 INFO

The `INFO` command is simple: it queries a BED for state and identification information. A valid response string is 32 bits: the first indicates whether this device is raising IRQ (`IIND`), the second indicates whether the BED is a high- or low-latency device (`LIND`), and the remaining bits identify the hardware that is connected to the slot. We use 7 bits to denote the type of device, 15 bits to identify the manufacturer, and the remaining 8 to identify the particular model. We note that the 30-bit identification string will never be nonzero. `IIND` is 1 if the device is raising IRQ and 0 otherwise; `LIND` is 1 if the device is a high-latency device

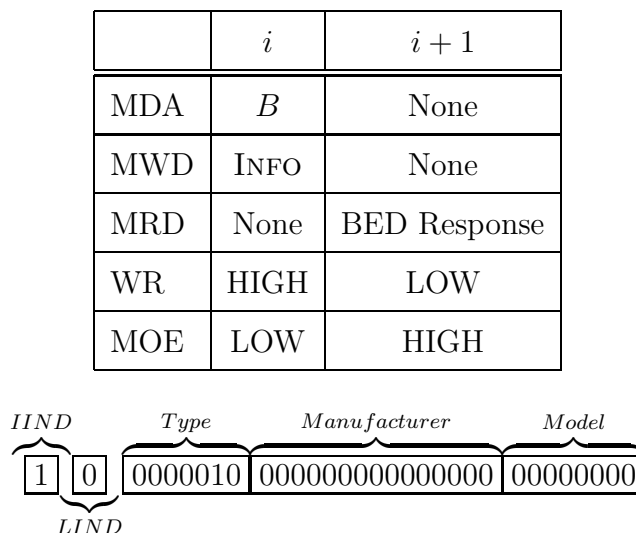


Figure 4: Time line of the INFO transaction, where B is the address of the BED. Shown above is an example response from a low-latency device that is currently raising IRQ.

and 0 otherwise. The remaining 30-bit identification string is described in Section 2.3.

To send an INFO query, the Beta addresses a particular BED (by setting MDA), sets WR, and puts the 32-bit INFO command on MWD. On the next clock cycle, the Beta unsets WR, sets MOE, and stops driving MWD. The BED is expected to place its identification information on MRD. The resulting value is stored in a register specified by the particular instruction. The entire INFO process takes a total of two clock cycles. A sample time line of an INFO query is shown in Figure 4.

2.2.2 REQUEST

The REQUEST command sends a query to a BED but does not require an immediate response. The argument to this command is the number of words that the Beta wants to read from the device. The intended usage of this command is for high-latency devices that have a turn around time of more than one clock cycle. It takes exactly one clock cycle to send a REQUEST: during this clock cycle, the Beta addresses a BED, puts the REQUEST command and its argument on MWD, and sets WR.

The purpose of a REQUEST is to notify the BED that the Beta wants to read a certain number

of words. Because the latency of devices varies, the expected usage of this command is such that the BED prepares the data while the Beta continues running other instructions; when the BED is ready to send the data, it raises IRQ and waits for the processor to address it and read the prepared data using a READBLK command (described in Section 2.2.3). The Beta is responsible for keeping track of REQUESTs that were sent. The method by which the Beta does this is described in Section 2.4.

2.2.3 READBLK

The READBLK command retrieves data from a BED and places it in a specified memory location, reading multiple words into consecutive blocks of memory. Like REQUEST, READBLK alerts a BED that it wants to read data, but expects the device to respond immediately with a number of words equal to the argument that was sent, one per clock cycle. This is generally used to follow up on a REQUEST to a high-latency device, or to read directly from a low-latency device. The operating system is responsible for keeping track of which devices have declared themselves as high-latency devices through the LIND bit of INFO (described in Section 2.2.1).

To issue this command, the Beta addresses the BED, sets WR, and puts the command and the argument n on MWD. For the following n cycles, the Beta sets MDA to the data memory address that the BED will send its data to. The processor releases MWD, and the device is expected to put one word on MWD per clock cycle. On each of the cycles, the Beta increments MDA so that consecutive words are read into adjacent parts of memory. An example time line of this process is illustrated in Figure 5.

This command, used by itself, has an overhead of exactly one clock cycle (the time it takes to send the command). After sending the command, the Beta only has to perform ADD instructions to increment MDA, since the BED drives the bus lines which the data memory reads from. Thus, the average throughput approaches one word per clock cycle as n becomes large.

With this command, the BED drives the bus when the Beta is not addressing it. Although

this may seem to give rise to bus contention problems, we note that `READBLK` is the only command where this happens. Thus, since the Beta never addresses any device other than memory during the execution of this command (it increments MDA during this time), it is guaranteed that only one BED will be driving the bus.

	i	$i + 1$	$i + 2$	$i + 3$	$i + 4$
MDA	B	m	$m + 4$	$m + 8$	$m + 16$
MWD	<code>READBLK(4)</code>	BED word 1	BED word 2	BED word 3	BED word 4
WR	HIGH	HIGH	HIGH	HIGH	HIGH
MOE	LOW	LOW	LOW	LOW	LOW

Figure 5: A sample `READBLK` transaction. It begins on clock cycle i , and 4 words are sent from the BED at address B to consecutive word locations starting from address m .

2.2.4 WRITEBLK

This command is used to send data in a contiguous block of memory to a BED. Like `READBLK`, it has an overhead of one clock cycle and transfers data at a rate of one word per cycle after the command is sent. The process is similar to `READBLK`: the Beta sets MDA to the BED address, sets WR, and sets MWD to the command and the argument n . In the following n cycles, the Beta sets MDA to the data memory address that contains the word to send to the BED, sets MOE, and unsets WR. The Beta increments MDA every clock cycle. An example time line of this process is illustrated in Figure 6.

Note that during the execution of this command, the BED reads data from the bus lines when the Beta is not addressing it. We emphasize the fact that a BED will only read data from the bus lines if it has seen a `WRITEBLK` from MWD in a previous clock cycle. Because the Beta increments MDA for each of the n clock cycles without addressing another BED, we are guaranteed that only one BED will be trying to read data from the bus at a time.

READBLK and WRITEBLK have a high average throughput when a large amount of data is transferred. If we only want to transfer one word of data, the amount of overhead is equal to the transfer time, meaning that it takes two cycles to transfer one word. It may seem reasonable to simply add new commands to handle this case—for example, a WRITEONE command that sends the data in the argument, allowing for a total transfer time of one clock cycle. However, this would allow us to only transfer 29 bits of information (3 bits must be reserved for the command if we add new commands) at once, which in turn can cause problems with memory alignment. Although logic can be added to pad the word, adding such a command as WRITEONE introduces an unnecessary amount of complexity. We feel that the extra clock cycle is worth the simplicity, especially if only 29 bits of data can be sent via WRITEONE.

	i	$i + 1$	$i + 2$	$i + 3$	$i + 4$
MDA	B	m	$m + 4$	$m + 8$	$m + 16$
MWD	WRITEBLK(4)	Mem word 1	Mem word 2	Mem word 3	Mem word 4
WR	HIGH	LOW	LOW	LOW	LOW
MOE	LOW	HIGH	HIGH	HIGH	HIGH

Figure 6: A sample WRITEBLK transaction. The transaction begins on clock cycle i , and 4 words are sent starting from memory address m to the BED at address B .

2.3 Identification

The ROFLBus does not require manufacturers to produce BEDs with unique IDs for each piece of hardware. Furthermore, we do not require BEDs to know what its own address is. Because the slots are enumerated and only one bit of MDA goes to each slot, the address of each slot is unique. An advantage of this design is that it allows for a simple plug-and-play interface for the end user because users need not manually assign an address to each BED.

As described in Section 2.2.1, each BED must respond to INFO queries with information about its type, manufacturer, and model. A partial list of types is given in Figure 7. The 15-bit

Type string	Device	Type string	Device
0000001	Keyboard	0001000	Printer
0000010	Mouse	0001001	Scanner
0000011	Hard drive	0001010	Sound card
0000100	CD-ROM drive	0001011	Video card
0000101	DVD-ROM drive	0001100	Ethernet card
0000110	Floppy disk drive	0001101	TV tuner card
0000111	Mass storage device	0001110	Joystick/Gamepad

Figure 7: Listing of possible type strings. New types of devices may be added in future revisions of the ROFLBus specification.

manufacturer strings are assigned by the Federal Communications Commission (FCC) in the form of the FCC ID code. The 8-bit model string is left to the manufacturer to set. Type strings are assigned such that a response to the INFO command always has at least one high bit. This is because a query to a slot that does not respond (i.e., MRD stays at 0) within one clock cycle is considered to be an empty slot—either no device is present, or it is not responding properly. Manufacturers must ensure that BEDs can respond to INFO queries immediately.

2.4 Initialization and Maintenance

As mentioned in Section 1, the Beta frequently polls devices on the ROFLBus to extract state information. However, to avoid redundant processing and minimize the number of requests made over the bus, we reserve 94 words of memory (addresses 0x04 through 0x17C) to store information about the state of each of the slots. The first 32 of these words is referred to as the Connected Device List (CDL). The first word (address 0x04) contains a 1 in the bit positions that correspond to connected devices. The remaining 31 words of the CDL contain the responses to INFO on each of the slots. While having the extra word at 0x04 may seem redundant, we can speed up many transactions using it and is a worthwhile benefit at the cost of only one word of memory.

The 62 words from addresses 0x80 to 0x17C form the Request State Table (RST). Whenever

Memory address	Contents
0x04	String with 1s in the bit positions corresponding to connected devices
0x08	INFO string of first device
0x0C	INFO string of second device
⋮	⋮
0x80	Memory location corresponding to last REQUEST to first device
0x84	Argument of last REQUEST to first device
0x88	Memory location corresponding to last REQUEST to second device
0x8C	Argument of last REQUEST to second device
⋮	⋮

Figure 8: Structure of the CDL and RST.

the Beta issues a REQUEST to a device, it records the argument in RST and the location in memory to write to, thus requiring 2 words per slot. Figure 8 illustrates the structure of the CDL and RST in memory and Figure 9 gives pseudocode showing how the Beta maintains the CDL and RST.

There are only two reasons for a BED to set IRQ: it either has data ready to send, or has just been inserted. While we do not require BEDs to know what their address is, we do require them to know if the Beta has addressed it since the device was inserted. When the Beta sees that IRQ is raised, it begins querying each slot on the ROFLBus with an INFO command. If it receives no response, it sets the corresponding bit in the CDL to 0. If it does receive a response, it checks to see if that particular device is raising IRQ. If it is, and it has no entry in the CDL, the Beta adds it; otherwise, we know that there already was a BED in that slot and it wants to send data to the Beta. Figure 9 shows how the Beta makes requests to the slots on the ROFLBus and detects new BEDs.

The second reason a BED could set IRQ is if it is ready to send data. This could be either because it has generated an event, or has finished preparing data from a previous REQUEST.

```

cdl = Mem[0x04]
prep = 1 << 31
i = 1
c = 1
while i != prep
  resp = INFO(i)
  if resp == 0
    cdl = cdl & ~i
  else if (prep & cdl) == prep
    if (cdl & i) == 0
      cdl = cdl + i
      Mem[0x04+4c] = resp
    else
      HANDLE_READ(c)
    end if
  end if
  i = i << 1
  c = c + 1
end while

```

Figure 9: The interrupt handler pseudocode, showing how the Beta detects new BEDs. The variables here represent registers, and `Mem[n]` is the n th byte of data memory.

In the latter case, the procedure that called `REQUEST` will have made an entry in the `RST` for that particular device. This way, the Beta can distinguish between new events and responses to old `REQUESTS`. If the value in the `RST` is nonzero, then the Beta sends a `READBLK` to the device with an argument equal to the value in the `RST`. Otherwise, the Beta calls `READBLK` with an argument of 1 (read one word). Figure 10 gives pseudo code for the `HANDLE_READ` subroutine, which determines why a BED set `IRQ` and reads in the appropriate amount of data.

If there was no entry in `RST`, then we store the event that the BED reported in the `RST`. Otherwise, if a `REQUEST` for data was sent previously, the `RST` contains the number of words to read as well as the address of the first location in data memory to store the words. This means that whenever the Beta sends a `REQUEST`, it must also set two values in data memory. Thus, it takes three clock cycles to correctly send a `REQUEST`: two to update data memory, and one to send the command.

```

HANDLE_READ(n):
  addr = RST + (n << 2)
  rs = Mem[addr]
  loc = addr+4
  if rs == 0
    READBLK(1, loc, ADDR(n))      | read 1 word into loc
  else
    READBLK(rs, Mem[loc], ADDR(n)) | read rs words into Mem[loc]
    Mem[loc] = 0
  end if
  Mem[addr] = 0                    | a ST operation

ADDR(n):
  | returns the address of the nth slot

```

Figure 10: Pseudo code for the `HANDLE_READ` subroutine referenced in Figure 9. `RST` is the address of the first word of the Request State Table, which is `0x80`.

This design may give rise to a race condition where the device raises `IRQ` to send an event after the Beta updates `RST` but before it actually sends the command. We resolve this simply by requiring BEDs to always yield to the Beta’s commands. In this situation, the BED must release `IRQ`; it may raise it once again and send the event after the Beta has performed the appropriate `READBLK`.

Note that whenever a BED fires an event on its own (i.e., raises `IRQ` but not in response to a `REQUEST`), it can only send one word of data to the Beta. This may cause trouble for devices that frequently send events that are more than one word in length—it must either repeat the event-sending process several times (which may take a very long time), or it can send one word of data indicating the number of words that are in the event that it generated. It is up to the driver (see Section 2.5) to interpret the words sent from the BED.

A limitation with the way the ROFLBus handles reads from BEDs is that if a `REQUEST` is sent to a BED, the next time it sets `IRQ` must be a reply to that request (i.e., it cannot report any other events to the Beta until it sets `IRQ` to tell the Beta to perform a `READBLK` on

it). However, because REQUEST is to be used for high-latency devices only, we expect that BEDs will generally not need to generate any other events—all they should be doing between the time of a REQUEST and the time of the READBLK is preparing the data. To work around this limitation, BEDs can raise IRQ again after the Beta has called READBLK to notify the Beta of an event.

Detection of a new device only involves updating the CDL. Since it takes two clock cycles to perform each INFO, one cycle to read the response, one cycle to determine if IIND is 1, and two cycles to increment the counters, in the worst case it will take 6 clock cycles per slot to determine if a new device has been inserted. If a device is inserted into the last (31st) slot, it takes 186 clock cycles to detect its insertion. However, if another device is also raising IRQ, then it the Beta will handle the reads, causing detection to take longer. We feel that this is not an unreasonable detection time; power users wishing to prioritize devices may do so by putting the more important ones in lower-numbered slots—thus, reads from those devices take priority over the insertion of new devices. The devices will still work correctly regardless of the device ordering—the only difference is the access time.

We have considered schemes that allow the Beta to immediately recognize which device is raising IRQ rather than polling the slots but have found this to require additional hardware. Having separate hardware for this purpose appears to allow for lower latencies compared to a polling scheme operating on its worst case scenario, we believe that the cost incurred by adding the hardware is not worth decreased latency. A mere 168 clock cycles is not very noticeable, and for users of personal computers this flaw should be acceptable.

To detect when devices are removed, we periodically query the devices that have an entry in the CDL. On every Q th clock cycle we read CDL, determine which slots are currently filled, and send an INFO to them. We choose Q to be high enough such that it does not bog down the operation of the rest of the system, but low enough so that the removal of a device is detected within a reasonable amount of time—a reasonable value for Q is 500,000 (poll every $\frac{1}{20}$ th of a second). This process takes two clock cycles (to send the command) plus the overhead of

looping through CDL. The run time is similar to when we detect new devices, except that we do not spend extra cycles sending INFO to devices not in the CDL.

2.5 Filesystem API

As described in Section 1, filesystem names are very simple: the n th device has a filesystem name `/dev/bed n` . Though the naming scheme does not lend itself to simplicity on the end user's part, all of the INFO strings of connected devices are available in memory for the operating system to parse. When the operating system detects that a new device has been inserted (i.e., the CDL was updated), the identification string can be retrieved from memory. This string, which was described in Section 2.3, is used to load the appropriate device driver. In this section, we show how device drivers handle calls to `open()`, `read()`, `write()`, and `close()`, and demonstrate how a driver for a sound card BED might be implemented.

Our naming scheme is advantageous because device names never get shuffled as long as the BEDs are connected to the same slots. While this may be problematic for devices that users frequently insert and remove, we prefer this design for two reasons. First, the problem can be solved on a higher level by applications which abstract the device names away from the user. Second, while we expect most BEDs to be connected to a single machine and not removed often, ROFLBus manufacturers can put labels next to the physical slots indicating which slot it is, thus allowing users to deduce the filesystem name easily.

For the sound card BED example, we define a simple communication protocol as follows: the first word of data sent to a BED is a command which indicates the operation to be performed on the BED and arguments. Any words following this are assumed to be sound data. In this case, we define four possible commands: `PREPARE`, `PLAY`, `RECORD`, and `END`. Thus, we must use 2 bits to denote the command, leaving 30 bits for arguments. Figure 11 summarizes these commands.

When an application calls `open()`, the driver places in memory a `PREPARE` command. In the case of playing a sound, the application could call `open()` with the flag `O_WRONLY`, so the

Command	Prefix	Argument	Purpose
PREPARE	00	0 or 1	Alert BED that either PLAY (0) or RECORD (1) follows
PLAY	01	# words <i>s</i>	Play <i>s</i> words of sound data
RECORD	10	# words <i>s</i>	Record <i>s</i> words of sound data and send it back
END	11	None	Alert BED that transaction is done

Figure 11: Table of commands for an example sound card BED communication protocol.

driver would create a PREPARE command with an argument of 0 (see Figure 11). It then calls on the Beta to send a WRITEBLK with this word to the device. If we wanted the BED to record a sound, we could call `open()` with `O_RDONLY` and send a PREPARE message with an argument 1. Our simple sound card BED can only either play or record, but cannot do both at the same time; thus, the driver will return an error code if a flag other than `O_RDONLY` or `O_WRONLY` is specified. Pseudocode for the `open()` function is given in Figure 12.

In general, `open()` will send a WRITEBLK of length 1, indicating to the device that a communication is starting. This may not be necessary for stateless devices, and thus we do not require `open()` to send a WRITEBLK. However, all device drivers should check the CDL to make sure the device exists and has the correct INFO string. Generic drivers may support multiple INFO strings. In the case of our sound card example, calling `open()` sends a PREPARE command. Upon receipt of this command, the BED allocates a buffer. Note that in the example code, we make use of a variable called `IN_USE`: this ensures that one can only call `open()` on the device once before calling `close()`. This sound card design may not reflect reality, but we include it in this discussion for the sake of example.

The implementation of the `read()` and `write()` functions are relatively easy: the arguments to READBLK and WRITEBLK have a one-to-one mapping with `read()` and `write()`, respectively. The only exception is that if we wish to read a number of bytes that is not an integral number of words, we must pad the request. The user is responsible for making sure that the memory buffer that is filled with the `read()` command is adequately large.

```

#define MY_TYPE          0b0001010
#define MY_MANUFACTURER 0b0000000000000000
#define MY_MODEL         0b00000000

int open(const char *pathname, int flags)
{
    int n = get_bed_number(pathname); // extract n from ‘‘/dev/bedn’’
    int addr = get_bed_address(n);    // defined below
    int prep;

    if(!is_correct_type(n) || !device_exists(n) || IN_USE == 1)
        return -1;
    else if(flags == O_RDONLY) {
        prep = 0;
    }
    else if(flags == O_WRONLY) {
        prep = 1;
    }
    else return -1; // more sophisticated devices may support more flags,
                  // this one does not, so return an error

    int fd = get_new_file_descriptor(pathname); // provided by kernel
    IN_USE = 1;
    WRITEBLK(1, &prep, addr);
    return fd;
}

int is_correct_type(int n)
{
    // Return 1 if the first 30 bits of CDL[n] match the concatenation
    // of MY_TYPE, MY_MANUFACTURER, and MY_MODEL. Return 0 otherwise
    // Generic drivers need not check all 30 bits; only MY_TYPE
}

int device_exists(int n)
{
    // Return 1 if the nth bit in Mem[0x04] is 1, 0 otherwise
    // See Section 2.4 for details
}

int get_bed_address(int n)
{
    return (1<<31) + (1<<n); // see Section 2.1
}

```

Figure 12: Pseudocode for open() for an example sound card BED.

```

int write(int fd, const void *sound_data, size_t count)
{
    int addr = fd_to_bed_address(fd); // provided by kernel
    if(addr == -1) // error
        return -1;

    int numwords = ceil(count / 4);
    const int *buf = (int *)malloc(4*(numwords+1));
    buf[0] = make_command(PLAYSOUND, numwords); // see Figure 11
                                                // for format of command
    memcpy(&buf[1], sound_data, numwords); // prepare data
    WRITEBLK(numwords, buf, numwords);
    return numwords;
}

```

Figure 13: Implementation of `write()` for the example sound card BED, which causes it to play a sound.

The `write()` command for our sound card tells the BED to play sound data that is in memory. The first word we send is the `PLAYSOUND` command, which, according to our protocol, tells the BED how many words of sound data are to follow. Then, the data in the buffer is sent to the device. Pseudocode for `write()` is given in Figure 13; code for `read()` is not given, but is completely symmetric with `write()`.

The `close()` function, much like `open()`, may not necessarily require data to be written to a device, but must at least notify the kernel to invalidate the associated file descriptor. In our case, we send `WRITEBLK` to the sound card containing the `END` command, which frees the buffer allocated by `open()`. Pseudocode for the `close()` function is not provided but is symmetric with `open()`.

3 Conclusion

The ROFLBus is a slot-based design aimed at users of personal computers. It can perform fast block data transfers at up to 40 MB/s and gives users a simple plug-and-play interface. No additional hardware is required, making the ROFLBus a low-cost solution. This design

does not cope well with devices that use small block transfers because the amount of overhead is large relative to the length of the transfer. Despite this, we feel that the ROFLBus is a feasible design because the high latency is offset by the high throughput. Additionally, there can only be up to 31 devices connected to the ROFLBus at a time, but we believe that this is acceptable for personal computers, which can easily accommodate this limit.

Acknowledgments

I would like to thank Miguel Ferreira for helping to answer my questions, as well as the rest of the 6.033 staff for coming up with this unimaginably exciting idea for DP1. I'd rather be writing BED bus specifications than sleeping.

Jelani Nelson helped proofread the first page of this document and gave a couple (i.e., two) words of advice.