Yellow Bus: A Universal Data Bus for BED Devices

by Adam Smith[1]

ID# 957-09-4405

Recitation Instructor: Professor Rinard, 11 AM

[1] This work was funded in part by my parents under grant number CHILD-3.

Table of Contents

List of Figures

List of Tables

This paper presents Yellow Bus, a single-master, flat architecture, polled bus for the Beta processor to use in communicating with other devices. Because the bus is operated on the Beta memory lines it is important to minimize the amount of that address space occupied. Two levels of addressing are used, session addresses and device IDs, to facilitate reliable device identification while minimizing the address space footprint of the system. In addition, fairness with other user-mode programs is achieved by using a microkernel architecture; all I/O is performed in user-mode. The interrupt-driven polling minimizes latency for reading data but also maximizes performance by minimizing unnecessary device polling. These and many other features make the Yellow Bus the right solution for any application requiring a high performance bus.

1 Introduction

Just like any search through a solution space, the design of this data bus is the result of many design iterations. This bus design could not have been done in a strictly linear fashion because the system has too many interdependencies; this constraint is true even with the powerful abstractions we develop at the end of the design description. The result is difficulty for the author, and for the reader as well. It is an undertaking to describe a very non-linear system in a linear, written, fashion. In an effort to ease the burden we will describe the system from a high level view before going into the details.

The Yellow Bus sends data out over the memory address lines of the Beta. Those wires include an addressing scheme which we use to identify a specific BED that we want to communicate with. Thus, communication is (mostly) in a one-to-one fashion. Although some time is spent specifying the format and rules for the signals sent across the wires, most engineering is focused on problems arising from the addition of new devices to the bus.

The difficulties surrounding the addition of new BED devices arises from the many moving parts of the system. It is easier to discuss the cars that the Ford Motor Plant produces than to talk about the plant itself. Similarly, the space of interactions that result from the addition of a device is far larger than the passing of a data packet between the layers that we construct.

Most of this paper's content is in section 2, the Design Description. First, the notion of data packets are introduced, which is then extended to discucc packet queues. The packet queues are a very nice abstraction barrier between the lower level processes and the higher level OS- and kernel-driven interactions. Thus, after packet queues are introduced, the discussion first focuses on the physical layer. Following that discussion, the onus moves to the software interactions that support the addition of devices and the subsequent OS file calls.

2 Design Description

2.1 Abstraction Using Packet Queues

This section describes the contents and format of a Yellow Bus packet. Packets are central to each piece of the bus design, both low level hardware and high level software, and thus we begin the overview here. After discussing individual packets we will introduce packet queues, which will frame the rest of the paper.

Packet Format

A packet is the atomic unit of data passed between a Beta and a BED. The packet format is shown in Figure 1. A summary of the fields is given in Table 1. Note that the maximum size of a packet is 32 Kbytes (1024 words), and the minimum size of a packet is 4 words.

Field Name	Offset (words)	Size (data words)	Data Type	Contents
Length	0	1	Unsigned integer	Number of words in packet (including the Length word)
Checksum	1	1	Unsigned integer	CRC32 of all words in packet after offset 2, inclusive
Protocol Version Number	2	1	Unsigned integer	1 – version released 3/17/2005
Packet Type ID	3	1	Unsigned integer	See section 2.3
Data	4	(Length – 4) <= 32,640	Binary blob	N/A

Table 1. Addressing on the 32 Memory Data Address lines

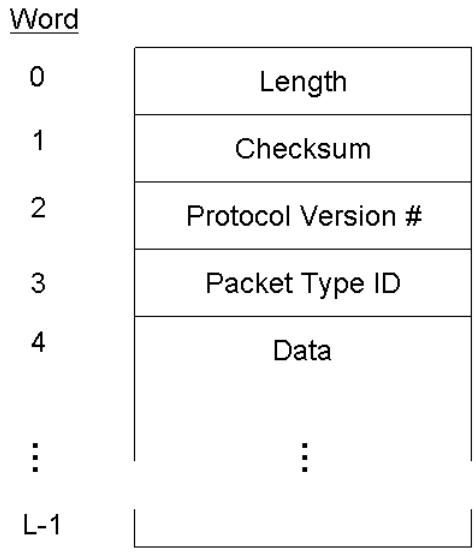

Figure 1. Packet contents and layout, each row represents a word of data, which is 32 bytes in our system. Each field is one word except for the Data field which can contain a variable number of words.

Packets were chosen as atomic units of data for many reasons; the metadata in the structure gives BEDs and Betas more information about how to manipulate the data it sends and receives. The length field of the packet allows the receiver to know how long the current data stream is, which allows the system to offer[2] the sender a temporarily uninterrupted data stream. This arrangement is in contrast to a system which uses streams of raw data which are paused and resumed at the whim of the Beta. Therefore, debugging and multiplexing becomes easier. Imagine a fax, copy, scan, print, and teleportation BED device connected to the Beta. If a process for each of these functions on the BED is sending data to a corresponding process on the Beta, then it is nearly impossible for a data stream to reliably switch bandwidth from the fax to the copy processes if the stream is subject to random pausing and resuming.

The checksum field allows the receiver of a packet to protect against bit errors during transmission. These bit errors are not likely to be caused by electrical problems. Instead, bit errors are likely to be caused by missing data words, extra data words, or other devices mistakenly writing data to the bus lines.

The protocol version number allows us to change the packet specification without making large sacrifices for backwards compatibility. Even though it does not provide immediate benefit upon release of the first version of this document, prevention can be a large part of computer system design.

Packet type IDs give us the ability to dispatch packets to the appropriate handler. On this low level the packet type IDs are related to handshaking, device ID initialization, and generic data transfer. Higher level protocols are likely to have more dispatching capabilities, as in the port numbers of TCP.

[2] A weaker word than guarantee

The Data field is the only thing in common with the packet-averse stream approach. It is useful to move user data around; the only structure or communication support it might have is in the case that a higher level protocol runs on top of Yellow Bus.

Packet Queues

There are four queues in the Yellow Bus system for each BED – an incoming and outgoing queue for each side, as shown in Figure 2. Each queue contains only packets, which are atomic units of data exchanged between the software on the Beta and each BED.

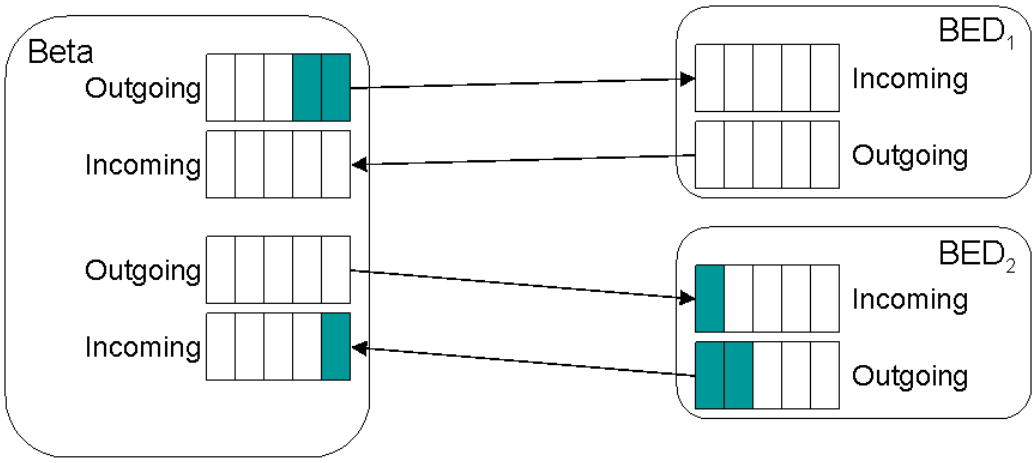

Figure 2. Packet queues in an example Yellow Bus configuration with two BEDS

The goal of the Yellow Bus is to allow programs running on the Beta and BED to exchange data. The queues in the system abstract the design of the Yellow Bus into two core pieces:

1) The physical connection level which transmits packets from one side to the other. Like all things, at glance this seems like an easy thing to do. However, we must deal with addressing, achieving concurrency with the single Beta processor, and fairly scheduling the various pipes in the system.

2) The software running in both ends must be able to place data into outgoing queues and read data from incoming queues. The issues here include specifying initialization procedures, how the OS interacts with drivers, coping with changing session addresses across restarts, et cetera.

The remainder of this paper is structured around these two layers. First, section 2.2 will discuss the physical layer issues. Next, section 2.3 will outline the higher level software that allows common programs to communicate with BEDs using a simple API.

2.2 Physical Layer

This section explains the hardware level dynamics which allow the Beta and BED devices to exchange data. The goal is to allow each device to move data from its outgoing queue to the other end's incoming queue.

Wiring

The Yellow Bus uses the physical connections specified in the assignment.

BED Device Addressing

The Beta's memory lines only support 32-bit addressing. If a unique address is assigned to each BED device produced (either at manufacturing or when first used), then the addressing scheme would need to have 40 bits (to support one trillion devices). Not only does BED addressing on the 32-bit memory lines pose a problem, but the Yellow Bus must share an address space with data memory. We do not want to penalize users of the Yellow Bus by significantly reducing the amount of memory the Beta can concurrently support. This could especially be a problem if current Beta users have already installed the maximum amount of memory possible; if we allocate a sizeable portion of the address space to the Yellow Bus then these users would lose storage space.

Therefore the Yellow Bus must not require a large portion of the Beta's memory address space. This requirement is met by using BED Session Addresses, as outlined in Table 2 below. Only an extremely small faction of the address space is used by the Yellow Bus. This small space, however, allows the Beta to address each BED separately, except for when the BEDs are first connected to the system.

The idea is that we will assign an address to each new BED on the bus. This address is only used for one connection session. It is reset if the Beta reboots, the BED is unplugged, et cetera. The size of this address space is 256 addresses, which is the limiting factor for the number of BEDs that can be attached to a Beta at any given time. Additionally, there are two special addresses (the Default BED Address and the Broadcast Address), which are discussed in section 2.3.

Function	Lowest Address	Highest Address	Size of Range
Data Memory	0x00000000	0xFFFFFDFF	~ 4 Billion
BED Session Addresses	0xFFFFFE00	0xFFFFFEFF	256
Default BED Address	0xFFFFFFFF		1
Beta to BED Broadcast Address	0xFFFFFFFE		1
[Reserved]	0xFFFFFF00	0xFFFFFFFD	254

Table 2. Addressing on the 32 Memory Data Address lines

Interrupt Requests

The Beta's memory I/O channels are designed for master-slave operation; the processor always initiates requests (read or write) to which the memory responds. This scheme works great for Beta to BED communication. It breaks down, however, when BED devices need to asynchronously initiate requests to the processor. For example, a keyboard BED will need to send data to the processor at unpredictable times.

The Yellow Bus solves this problem by supporting interrupts on the IRQ input of the Beta. All BED devices share the same IRQ line by a parallel electrical connection. If a BED wishes to initiate communication with the Beta then it can raise the IRQ line to a logical high, which is read by the Beta. Because the BEDs share the same IRQ line, the value read by the Beta will be the logical OR of the interrupts from all BEDs. Thus, if an interrupt is raised, the Beta does not know which devices raised the interrupt, or how many there were. The Beta only knows that *some* device has data to send. Therefore, the Beta's response to an interrupt is to poll each BED device in turn for data.

The process is illustrated from the Beta's perspective in Figure 3, and from the BED's perspective in Figure 4. Although the `poll()` function will be defined later, it is a piece of code which asks a specific BED if it has any data.

```
Beta_interruptRoutine() {

  // invoked on Beta when IRQ = 1
  //  or when this routine has not been invoked in 200ms

  globalInterruptEnable = False
  schedulePollTask()
}

Beta_pollTask() {
  // pull data from BEDs
  for each x in activeBEDs
    poll(x.sessionAddress, scheduler.maxNumPackets(x))

  // poll the default BED addr (discussed in section 2.3)
  Beta_pollDefaultAddress() // defined later

  globalInterruptEnable = True
}
```

Figure 3. Beta pseudocode – IRQ service routine. All attached BED devices (and the default address, in case a new device is attached) must be serviced since the Beta does not know which BED raised the interrupt.

```
BED_sendData(Packet p)

  // invoked by BED link layer

  IOQueue.outgoing.addPacket(p)

  IRQ = 1   // only if IRQ is used by this BED
```

Figure 4. BED pseudocode – notifying the Beta that a packet is ready

The alternative to our design is to not use interrupts; instead, occasionally poll each device to see if it has new data to send. The advantage of our design is that interrupts provide tremendous savings in latency. In the Yellow Bus, the BED can raise the IRQ line and be serviced shortly thereafter. In the non-interrupt design, the device must wait some period (e.g. 200ms) to be polled for data. The interrupt design also provides savings in efficiency; if none of the BED devices have data to send then they will not be polled for data.

Furthermore, the Yellow Bus' interrupt design does not preclude the use of the polling model on the master. For example, suppose that there is another type of processor, called Gamma, which does not have interrupt support. Then a Gamma could be used as the master of a Yellow Bus simply by using the polling model – with no changes to the BED devices! This could also be leveraged if the Beta's interrupt request line was needed for some other application.

The inverse is not true. We cannot neglect the interrupt line on a BED device without requiring any changes to the Beta. This design decisions is driven primarily be the low cost of allocating a GPIO pin on each BED to the IRQ line. The design cost is negligible, as is the processor resource demands. The IRQ line is also needed in the bootup phase for each BED, as discussed in section 2.3.

Finally, the interrupt design allows for better maximum throughput performance as a result of scheduling. If the scheduler from Figure 3 allowed only ten packets to be sent per polling cycle, then in the non-interrupt model each BED would have to wait at least 200ms to send each group of 10 packets. This is true even if there is only one BED on the bus. On the other hand, in the interrupt-driven model, if the BED's outgoing packet queue still has data after a polling cycle, it will be served again immediately because it will hold IRQ high.

Concurrency

Multiple devices might need to send data at any given time. Support for servicing each device in a manageable fashion is achieved by abstraction. At any given time each BED device is addressed individually and has its own I/O queue. Thus, the Beta takes turns communicating with each active BED in its own context, much as a multitasking operating system handles multiple programs in a personal computer. As with anything, there are exceptions. The first exception occurs when a BED device is first plugged into the Yellow Bus. It has not been assigned a BED Session Address, but instead listens on the Default BED Address. It is possible for more than one device to be plugged in at the same time or otherwise concurrently listening on the Default BED Address, and we will deal with this case (very carefully) in section 2.3. The second exception is the broadcast BED address which allows a Beta to speak to all BEDs (regardless of the BEDs' Session Address). This is useful for rebooting a Beta, as we will see later.

Aside from hardware-level concurrency, we need to ensure that user-level programs are not starved of cycles. For example, if a hard drive BED is transferring a one gigabyte file 10 packets for each cycle, then it will constantly keep the interrupt request line high. This is undesirable because the Beta will continually serve those interrupts, not allowing user mode programs to execute. The incoming data queue for that BED will fill up quickly since the end user program will not be able to pull data off of the queue and process it. Furthermore, the Beta will not respond to user input or be able to run other programs. To fix this, we handle the polling of BEDs in user-mode. This microkernel-inspired architecture allows the task scheduler to balance spending cycles on I/O with allowing other user programs to run. In the example of the big file, the consuming program will be able to pull data off the queue since it will be in the same task scheduling pool as the process populating the queue with incoming data. The user level execution of polling is achieved by scheduling the polling task when an interrupt occurs, and then returning. Of course we set GIE (Global Interrupt Enable) to false so that another interrupt cannot occur until the first one has been serviced; this way we do not build up a queue of polling tasks.

What's in a Poll?

Each initialized BED device has a Session Address associated with it, as well as an I/O queue and any number of other things. A function named `poll()` is referenced in Figure

3, and Figure 4, which presumably prompts a BED for data and puts whatever data is received (if any) into an input queue on the Beta. This is illustrated using pseudocode in Figure 5 below.

An ancillary function of the procedure is to prune devices that are unplugged from the Beta. If a BED has no data to send then it should put -1 on the memory data output lines. If the BED is sending a packet, then it should output the total number of words (at least 4 words, as illustrated in section 2.1). Thus, any operational BED listening on the appropriate Session Address will never output a zero. If the Beta reads a zero, then the device must be unplugged from the Beta. As a result, it is unmounted from the system by invoking `BED.unmount`, and is removed from the list of active BEDs.

```
Beta_poll(BED b, int maxNumPackets) {

  for i = 1 to maxNumPackets
    if b.IOQueue.incoming.full  // incoming buffer is full
      return
    Packet p = receivePacket(b)
    if p == null               // no packet sent
      return
    b.IOQueue.incoming.add(p)
}

Beta_receivePacket(BED b) {
  // 1 word = 32 bits

  // the first word in a packet is the packet's length
  totalLengthInWords = readMemory(b.sessionAddress)

  if totalLengthInWords == -1    // -1 if BED has no packets
    return null

  if totalLengthInWords == 0
    b.unmount
    activeBEDs.remove(b)         // BED b is dead
    return null

  Packet p = new Packet(totalLengthInWords)
  for i = 1 to totalLengthInWords - 1
    p.append(readMemory(b.sessionAddress)

  return p
)
```

Figure 5. `poll()` pseudocode – checking for, and saving, data

Achieving Bidirectional Communication

So far all of the discussion has been centered around learning when BED devices have data, and retrieving that data from the BED when appropriate as determined by the task scheduler.

The other mode of communication, sending data from the Beta to a BED, is simpler. Since the Beta naturally knows when it has data to send and is in control of the bus, no polling is needed. Instead, whenever a packet is ready to be sent, it is added to the queue and a task to send it out is added to the user-mode execution scheduler.

```
Beta_sendData(BED destination, Packet p) {

  // invoked when packet p is ready for a single BED
  destination.IOQueue.outgoing.add(p)
  scheduleSendTask(destination)
}

Beta_sendTask(BED destination) {
  globalInterruptEnable = False

  Beta_sendPacket(destination,
         destination.IOQueue.outgoing.get())

  globalInterruptEnable = True
}

Beta_sendPacket(BED destination, Packet p) {
  for i = 1 to p.length
    writeMemory(destination.sessionAddress, p.getWord(i))
}
```

Figure 6. Beta pseudocode – sending a packet from the Beta to a BED. The process includes scheduling a user-mode program to send the data.

The BED Side

The missing piece in the pseudocode presented so far is the BED procedure which responds to the `readMemory()` and `writeMemory()` calls made on the Beta. This is built using interrupts to detect when the BED's session address appears on the memory address lines at the edge of a clock cycle.

If the Beta is reading "memory," then the BED performs the following. Upon the interrupt, the BED determines if it is in the middle of sending a packet. If so, then the BED outputs the next word in the packet to the memory data lines. Otherwise, the BED tries to retrieve a packet from its outgoing queue. If there is not a packet in the queue, then the BED outputs a negative one value onto the 32-bit memory data lines. If there is a packet available on the queue, the BED outputs the first word in that packet and continues to send the words in that packet with each interrupt until the packet has been sent.

A similar procedure takes place when the Beta writes to “memory,” and the BED receives the data.

Both of these procedures are shown as pseudocode in Figure 7.

```
BED_interrupt(memWriteEnable, memOutputEnable, _
              memReadData, memWriteData) {

  // invoked when memory (address lines equal the
  //   current session address) AND (clock cycle)

  static wordBuffer, packetLength

  if memWriteEnable == True         // receive data
    if packetLength == 0            // start of new packet
      packetLength = memWriteData   // first word of packet
                                    //  is length of packet
    wordBuffer.appendWord(memWriteData)
    if (--packetLength) == 0
      IOQueue.incoming.add(new Packet(wordBuffer))

  if memOutputEnable == True
    if wordBuffer.length == 0
      wordBuffer = IOQueue.outgoing.get()

    if wordBuffer.length == 0  // no packets to send
      memReadData = -1
      IRQ = 0                  // we no more cause interrupt
    else
      memReadData = wordBuffer.popWord()
}
```

Figure 7. BED pseudocode – handling sending and receiving packets.

Throughput Analysis

Assume that there is only one BED device sharing the memory address lines and that the I/O user tasks get adequate cycles assigned from the task scheduler. In this case the physical layer design easily achieves 30 megabytes per second throughput at a 10 MHz clock rate. 30 MB can be transferred with about one million read cycles (or write, as the reader prefers). Ignoring overhead, the Beta would always be communicating with its BED. The overhead comes from scheduling the Beta_poll() task and checking to see if the incoming queue is full. Since the overhead is not more than nine times the time to read from the memory lines, the Yellow Bus’ throughput meets the benchmark.

2.3 Adding Devices – Relationships and Handshakes[3]

There are two stages of processes that occur when a BED is first plugged into a Yellow Bus. The first stage establishes a relationship with the BED, including giving it a session address, setting up I/O queues, and adding it to the list of active BEDs. The second stage is a handshake; the BED tells the Beta its Device ID and other relevant information, to which the Beta responds by initializing the appropriate driver and setting up OS I/O. The product of the handshake is that the Beta has (in the case of no errors) invoked a driver to handle I/O with the device, after which external user programs can interact with the BED.

Pinging the Default BED Address

When a BED device is first plugged into a Yellow Bus, its session address is the default BED address (0xFFFFFFFF as defined in section 2.2). During each poll of the BEDs, the Beta polls this default BED address, looking for newly attached devices. Ideally a new BED could respond and be immediately set up. Unfortunately, evaluation of the edge cases rules out this option. Specifically, if more than one BED device is attached at exactly the same time then there could be ambiguity as to who is interacting with the Beta on this default address.

The interrupt request line is used as the mediator. When a new BED is attached its processor picks a random time interval between 1 and 500ms. After waiting this period of time the BED checks to see if the IRQ line is high. If the IRQ line is high then the BED goes back to step one; it picks another interval and waits longer. If the IRQ line is low, however, the BED raises the IRQ line to high and has the default BED address token. If a newly attached BED has the default address token it can interact with the Beta on that address.

This scheme resolves contention between two or more new BEDs since they randomly compete for the token via the IRQ line. Furthermore, already-initiated BEDs do not need to worry about this protocol; if they have data to send then they can raise the IRQ line without damage.

Once a single newly attached BED has the default address token, an interaction must occur between the BED and Beta so that a session address can be established for further communication. This is detailed in Figure 8. The communication does not use packets; instead only two words are exchanged. This decision is based on simplicity and the lack of motivation for the benefits of packets in this context (e.g. multiplexing multiple streams).

[3] Removing devices as they are unplugged is a fairly trivial task addressed in section 2.2.

```
Beta_pollDefaultAddress () {

  answer = readMemory(DEFAULT_BED_ADDRESS)
  if answer == -1                              // a new BED!
    newBEDAddress = getAvailableAddress()
    writeMemory(DEFAULT_BED_ADDRESS, newBEDAddress())
    BED b = new BED(newBEDAddress)
    activeBEDs.add(b)
    b.handshake()
}
```

Figure 8. Polling the default BED address, and handling responses

Device IDs and Model IDs

The operating system should be able to identify devices across sessions. This requirement is important for naming consistency. Using 2 word (64 bit) random Device IDs, the Yellow Bus supports persistent naming without requiring that devices be manufactured with device-specific serial numbers.

Model IDs, on the other hand, are set during the manufacturing process. This value is the same single word (32 bit) random number for each device of the same type. It is used to identify the appropriate driver for the operating system to attach to the device. The value for each model is independently chosen by the manufacturer, which works out since the manufacturer can designate the device driver to service the chosen model ID.

Beginning the Handshaking Process

At the end of the handshaking process the new BED will have told the Beta its Device ID and Model ID, and the Beta will have loaded the appropriate device driver. If the Device ID is zero (the factory default value) then the BED has never been used before and needs to be initialized with a new random number. This simple process is shown in the first section of code in Figure 9.

The hidden aspect of the initial exchange is the packet type ID field of the packets associated with the handshake. A value of “1” in the packet type ID field indicates that the packet is a system level command rather than user program data. The type of commands and arguments to the commands are included in the Data field of each packet, as detailed in Table 3.

```
do {
  [b.deviceID, b.modelID] = getIDs(b)
  if b.deviceID == 0
    setDeviceID(b, rand*(2^64))
} while (b.deviceID == 0)

// initialize driver here
```

Figure 9. Extremely high level code for handshaking and intialization

Packet Description	Packet Type ID	Sender	Contents of Data Field in Packet (word: data)
Get IDs	1	Beta	00: 00 00 00 00
Reply with IDs	1	BED	00: 00 00 00 01 01: [deviceID.word1] 02: [deviceID.word2] 03: [modelID]
Set device ID	1	Beta	00: 00 00 00 02 01: [deviceID.word1] 02: [deviceID.word2]
Reset BED	1	Beta	00: 00 00 00 FF

Table 3. Addressing on the 32 Memory Data Address lines

Finishing the Handshake – Loading a Driver

Identifying the appropriate driver for a given BED device is straightforward; it is a table lookup on the available drivers and the model IDs that those drivers service. If there is not an appropriate driver on the Beta then an error message appears, as illustrated in Figure 10.

```
driver = lookupDriver(b.modelID)
if driver == null
  error.raise("Driver not available for this device")
else
  initialize(driver, b)
```

Figure 10. Driver lookup and initialization

Initializing the driver requires the following steps:

1) The driver asks the kernel if it has used a device with `b.deviceID` in the past. If the kernel has used that device before, the old name is used (for example, “hd1” would frequently fall into this case) and step #2 below is skipped. This step ensures that if the Beta reboots or a BED device is unplugged and plugged back in, the ordering of names will not be shifted around.

2) If the device is new to the system then the special file’s name must be determined. The driver has a predefined prefix for the device. For example, if it is a hard driven then the predefined prefix is “hd.” The driver asks the kernel for the next index in that prefix which has never been used before. A possible answer might be 3, in which case the new device’s name is “hd3”

3) The driver registers the device name with the operating system, requesting that file I/O requests to that device be sent to the driver.

From a high level perspective, the software abstractions we have built look like the illustration in Figure 11.

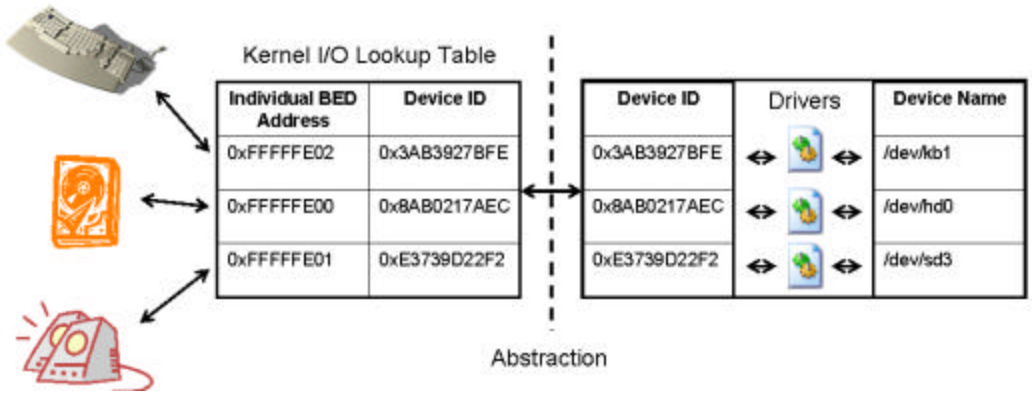

Figure 11. Abstractions and machinery of the Beta software BED support mechanisms

Beta Reset Procedure

Beta devices can restart and be otherwise volatile. The Yellow Bus provides a way for the master to ensure that all BEDs are in startup mode in case the Beta loses its state.

The astute and sequential reader will have noticed the Reset BED packet description in Table 3. A reset means that the affected BED(s) drop their current session address, reset their queues, and listen once again on the default BED address. The Beta can use the Reset BED packet to reset a single BED. The Beta can also use the packet on the broadcast address to reset all BEDs, without having to know the session address of each BED. Therefore the startup procedure is quite simple. If the Beta is restarted or otherwise suffers from significant error conditions, the entire Yellow Bus an be reset by sending a Reset BED packet on the Beta to BEDs broadcast address.

Dynamic Analysis

Some thought should be dedicated to the implications from the given design and why it works. The bus addresses are volatile across sessions, a consequence of the Beta's limited address space. To cope with this, we added Device IDs which are non-volatile and are used by all higher level software functions, as shown in Figure 11. This abstraction is the most important point of the higher level software description. If a BED is removed an later added, the higher level software sees the same device (according to its Device ID); the higher level software does not know about the different bus address.

Thus, the arbitrary removal, addition, rebooting, or transport between systems of both BEDs and the Beta will not change the important interface address – the special file name in the /dev folder.

OS Calls and an Example

Calls to I/O functions such as open() and write() are forwarded to the appropriate driver based on the a table populated by the registrations mentioned in step #3 of the driver initialization procedure above. The OS does not maintain state about BEDs; instead it only maintains mappings between file names and driver instances. The driver instances themselves maintain a pointer to the Device ID they are responsible for. Finally, a wrap-around layer provides simple Device ID to BED session address mapping based on the `activeBEDs` list.

Consider a BED device which is an LED alphanumeric sign. The sign can be programmed with messages by sending ASCII character strings, which are scrolled across the LEDs. A user program might make a call to:

```
fileDescriptor = open("/dev/sign0", forWrite).
```

In response to this the OS finds the LED sign driver and passes along the call. In this case the driver does nothing. The OS returns a file descriptor pointing to the special file referenced. Next, the user program might make a call such as:

```
write(fileDescriptor, stringBuffer, 5).
```

The OS once again passes this call along to the LED sign driver. The driver then decides to send an interpreted version of the ASCII string to the LED sign. After performing the appropriate manipulations, the device driver puts the resulting data into a packet. The driver concludes its work by invoking a send procedure which accepts as arguments the Device ID and the packet to be sent. This gets sent through the layers of indirection shown in Figure 11 and finally the packet gets put in the LED BED's outgoing queue, and is later sent.

3 Conclusion

This paper discussed the Yellow Bus, a data bus for a Beta processor to communicate with peripheral devices. The design was partitioned into two main sections – the physical layer interactions (including addressing, interrupt-driven polling, and scheduling) and the higher level software which provides an interface to the lower level BED I/O queues through the use of device-specific drivers. Many design decisions were made an discussed. Most of these decisions were motivated by simplicity and performance.

References

[this space intentionally left blank[4]]

[4] I do not have any references, of course excluding the project handout.