Serial Line IP Implementation for Linux Kernel TCP/IP Stack

Ajan Daniel Kutty

Bestin Jose

Ciju Rajan K

Daise Antony

Linto Antony

Serial Line IP Implementation for Linux Kernel TCP/IP Stack

by Ajan Daniel Kutty, Bestin Jose, Ciju Rajan K, Daise Antony, and Linto Antony

Copyright © 2004 ABCDL

Everyone is permitted to copy and distribute verbatim copies of this document, but changing it is not allowed.

Revision History

Revision 1.0   10 JUN 2004   Revised by: ABCDL
SLIP

Table of Contents

Acknowledgement

We take this opportunity to acknowledge all the people who have helped us whole heartedly in every stage of this project.

We are indebtedly grateful to the Head Of The Computer Science Department *Dr. M.N Agnisarman Namboodiri* for his support. We are also greatful to our Project guide *Lect. C.N. Sminesh* for his valuable guidance.

We also extend our sincere thanks to *Lect. P.V. Binoy* and *Anees Alappattu* for their valuable inspiration. We also extend our sincere thanks to all other faculty members of Computer Science & Engineering Department and the students of S8 CSE-A.

Abstract

This project implements a Serial Line IP driver (SLIP) for Linux kernel TCP/IP stack. The TCP/IP protocol family runs over a variety of network media: IEEE 802.3 (Ethernet) and 802.5 (token ring) LAN's, X.25 lines, satellite links, and serial lines. There are standard encapsulations for IP packets defined for many of these networks, but there is no standard for serial lines. SLIP, Serial Line IP, is currently a de facto standard, commonly used for point-to-point serial connections running TCP/IP.

The project aims to implement the SLIP encoding scheme on the Data Link layer of the TCP/IP protocol so that peer to peer connection through serial port is possible. For this a network driver is to be written with in the Linux kernel subsection.

Chapter 1. Introduction

Linux is considered as the programmer's paradise. For a computer science student it gives the opportunity to explore the internals of an Operating System.This project deals the networking side of linux.

To connect two computers we usually need hardwares like ethernet cards, connecting cables etc and a software called network driver to drive all this componets together. Our project connects two computers through serial port. The connecting cable uses just three pins of the serial port. We developed a network driver for this purpose.

TCP/IP protocol, which is used for networking is organised as protocol layers. It has got five layers. The first layer deals with all the application programs. The second layer deals with the connection between the systems(connection oriented or connectionless). The third layer deals with the addressing. The fourth layer is called data link layer. The last layer deals with the physical connection. This layering structure helps us to implement any of these layers without affecting the implementation of the other layers. Our project modifies data link layer and physical layer.

Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel, and "plugged in" at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

By default each COM port is attached to a chip called UART (Universal Asynchreonous Receiver Transmitter) which is used to perform serial to parallel or parallel to serial conversion of data and transmission of data. This UART is programmed for sending the data over the serial lines.

This project implements Serial Line IP protocol at the data link layer of the TCP/IP protocol stack. This is achieved by writing a special type of driver called network driver and inserting it in to the kernel. When the new network driver is inserted in to the kernel, it replaces the traditional TCP/IP Data link layer and the physical layer. The network driver is used to control the UART chip. UART will act as the new physical layer. Nine pin D connector is used to connect the COM ports of the computers.

Chapter 2. Platform And Tools Required

Platform used is GNU Linux. Linux operating system provides simple and easy way to do system level programming by manipulating the kernel. In order to accomplish this project, it is required to insert the network drivers into the kernel. Hence Linux is perhaps our best choice.

2.1. The Major Tools Used

- *1. make :* The Make utility automatically determines which pieces of large program need to be recompiled and issues command to recompile them. Make can be used to describe any task where some files must be automatically updated when others change. All these rules must be specified in the file ’Makefile’. Make has got a number of runtime options. To use make, we must write a file called the ’makefile’ that describes the relationships among files in our program, and the states the commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files. Once a suitable makefile exists, each time we change some source files, this simple shell command: make suffices to perform all necessary recompilations. The make program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.
- *2. gdb :* GNU Debugger (gdb) is the standard debugger on modern Unix systems. A Debugger is a program, which helps to troubleshoot code, mainly by allowing to execute the code line by line and examining or modifying the values of variables. If we want to use gdb to debug our program, we should compile it in a particular way. We should give the option -g to the compiler.
- *3. gprof :* Profiling refers to finding out how your program has been spending its time. Profiling helps you identify the hot spots in the program, those sections of code, which gets executed most often. Improving the efficiency of these hot spots will speed up the program considerably.
- *4. insmod :* insmod installs a loadable module in the running kernel. insmod tries to link a module into the running kernel by resolving all symbols from the kernel exported symbol table.If the module file name is given without directories or extension, insmod will search for the module in some common default directories. The environment variable

MODPATH can be used to override this default. If a module configuration file such as /etc/modules.conf exists, it will override the paths defined in MODPATH. The -f option of insmod can be used to forcefully insert the module into the kernel.

- *5. lsmod :* lsmod is used to view all the loaded modules. It also displays the use count of that module. User count indicates the number of programs that uses a particular module.
- *6. rmmod :* rmmod tries to unload a set of modules from the kernel, with the restriction that they are not in use and that they are not referred to by other modules. If more than one module is named on the command line, the modules will be removed in the given order. This supports unloading of stacked modules.
- *7. ifconfig :* Ifconfig is used to configure the kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed. If no arguments are given, ifconfig displays the status of the currently active interfaces. If a single interface argument is given, it displays the status of the given interface only; if a single -a argument is given, it displays the status of all interfaces, even those that are down. Otherwise, it configures an interface. Ifconfig is also used to assign a particular IP address to an interface.
- *8. route :* Route manipulates the kernel IP routing tables. Its primary use is to set up static routes to specific hosts or networks via an interface after it has been configured with the ifconfig program. When the add or del options are used, route modifies the routing tables. Without these options, route displays the current content of the routing tables.

Chapter 3. System Overview

3.1. Kernel Overview

- *Process management* The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.
- *Memory management* The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple malloc/free pair to much more exotic functionalities.
- *Filesystems* Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, diskettes may be formatted with either the Linux-standard ext2 filesystem or with the commonly used FAT filesystem.
- *Device control* Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a device driver. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape streamer. This aspect of the kernel's functions is our primary interest in this book.
- *Networking* Networking must be managed by the operating system because most network operations are not specific to a process: incoming packets are asynchronous events.

The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

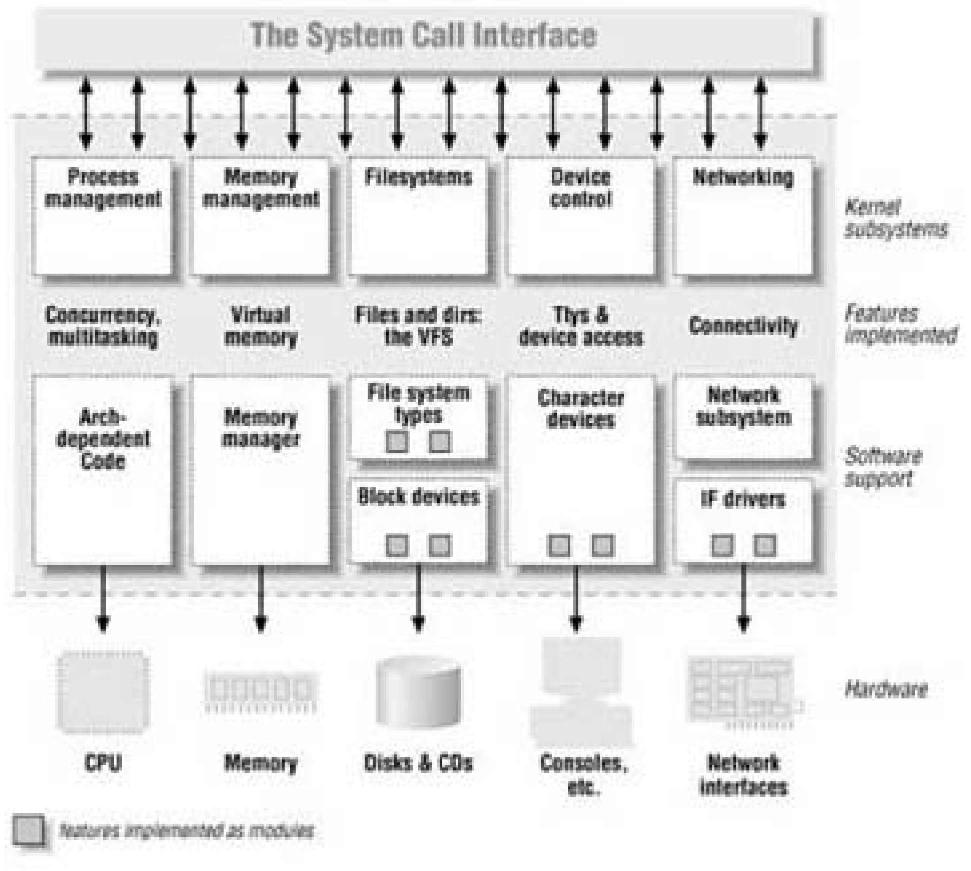


Figure 3-1. Kernel Structure

Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program.

3.2. Drivers

Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel, and "plugged in" at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

The Unix way of looking at devices distinguishes between three device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module. This division of modules into different types, or classes, is not a rigid one.

The three classes are the following:

- *Character devices* A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.
- *Block devices* Like char devices, block devices are accessed by file system nodes in the /dev directory. A block device is something that can host a file system, such as a disk. In most Unix systems, a block device can be accessed only as multiples of a block, where a block is usually one kilobyte of data or another power of 2. Linux allows the application to read and write a block device like a char device -- it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface.

Like a char device, each block device is accessed through a file system node and the difference between them is transparent to the user. A block driver offers the kernel the same interface as a char driver, as well as an additional block-oriented interface that is invisible to the user or applications opening the /dev entry points. That block interface, though, is essential to be able to mount a filesystem.

- *Network interfaces* Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Though both Telnet and FTP connections are stream oriented, they transmit using the same device; the device doesn't see the individual streams, but only the data packets.

Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as /dev/tty1 is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

3.3. Network Drivers

The role of a network interface within the system is similar to that of a mounted block device. A block device registers its features in the blk_dev array and other kernel structures, and it then "transmits" and "receives" blocks on request, by means of its request function. Similarly, a network interface must register itself in specific data structures in order to be invoked when packets are exchanged with the outside world.

There are a few important differences between mounted disks and packet-delivery interfaces. To begin with, a disk exists as a special file in the /dev directory, whereas a network interface has no such entry point. The normal file operations (read, write, and so on) do not make sense when applied to network interfaces, so it is not possible to apply the Unix "everything is a file" approach to them. Thus, network interfaces exist in their own namespace and export a different set of operations.

Although you may object that applications use the read and write system calls when using sockets, those calls act on a software object that is distinct from the interface. Several hundred sockets can be multiplexed on the same physical interface.

But the most important difference between the two is that block drivers operate only in response to requests from the kernel, whereas network drivers receive packets asynchronously from the outside. Thus, while a block driver is asked to send a buffer toward the kernel, the network device asksto push incoming packets toward the kernel. The kernel interface for network drivers is designed for this different mode of operation.

Network drivers also have to be prepared to support a number of administrative tasks, such as setting addresses, modifying transmission parameters, and maintaining traffic and error statistics. The API for network drivers reflects this need, and thus looks somewhat different from the interfaces we have seen so far.

The network subsystem of the Linux kernel is designed to be completely protocol independent. This applies to both networking protocols (IP versus IPX or other protocols) and hardware protocols (Ethernet versus token ring, etc.). Interaction between a network driver and the kernel proper deals with one network packet at a time; this allows protocol issues to be hidden neatly from the driver and the physical transmission to be hidden from the protocol.

When a driver module is loaded into a running kernel, it requests resources and offers facilities; there's nothing new in that. And there's also nothing new in the way resources are requested. The driver should probe for its device and its hardware location (I/O ports and IRQ line). The way a network driver is registered by its module initialization function is different from char and block drivers. Since there is no equivalent of major and minor numbers for network interfaces, a network driver does not request such a number. Instead, the driver inserts a data structure for each newly detected interface into a global list of network devices.

Each interface is described by a struct net_device item.

3.3.1. Details of structure net_device

The first struct net_device field we will look at is name, which holds the interface name (the string identifying the interface). The driver can hardwire a name for the interface or it can allow dynamic assignment, which works like this: if the name contains a %d format string, the first available name found by replacing that string with a small integer is used.

The net_device structure is at the very core of the network driver layer and deserves a complete description. At a first reading, however, you can skip this section, because you don't need a thorough understanding of the structure to get started. This list describes all the fields, but more to provide a reference than to be memorized. The rest of this chapter briefly describes each field as soon as it is used in the sample code, so you don't need to keep referring back to this section.

struct net_device can be conceptually divided into two parts: visible and invisible. The visible part of the structure is made up of the fields that can be explicitly assigned in static net_device structures. All structures in drivers/net/Space.c are initialized in this way, without using the tagged syntax for structure initialization. The remaining fields are used internally by the network code and usually are not initialized at compilation time, not even by tagged initialization. Some of the fields are accessed by drivers (for example, the ones that are assigned at initialization time), while some shouldn't be touched.

The Visible Head

The first part of struct net_device is composed of the following fields, in this order:

```
char name[IFNAMSIZ];
```

The name of the device. If the name contains a %d format string, the first available device name with the given base is used; assigned numbers start at zero.

```
unsigned long rmem_end;
unsigned long rmem_start;
unsigned long mem_end;
unsigned long mem_start;
```

Device memory information.These fields hold the beginning and ending addresses of the shared memory used by the device. If the device has different receive and transmit memories, the mem fields are used for transmit memory and the rmem fields for receive memory. mem_start and mem_end can be specified on the kernel command line at system boot, and their values are retrieved by ifconfig. The rmem fields are never referenced outside of the driver itself. By convention, the end fields are set so that end - start is the amount of available on-board memory.

```
unsigned long base_addr;
```

The I/O base address of the network interface. This field, like the previous ones, is assigned during device probe. The ifconfig command can be used to display or modify the current value. The base_addr can be explicitly assigned on the kernel command line at system boot or at load time. The field is not used by the kernel, like the memory fields shown previously.

```
unsigned char irq;
```

The assigned interrupt number. The value of dev->irq is printed by ifconfig when interfaces are listed. This value can usually be set at boot or load time and modified later using ifconfig.

```
unsigned char irq;
unsigned char if_port;
```

Which port is in use on multiport devices. This field is used, for example, with devices that support both coaxial (IF_PORT_10BASE2) and twisted-pair (IF_PORT_10BASET) Ethernet connections. The full set of known port types is defined in <linux/netdevice.h>;.

```
unsigned char dma;
```

The DMA channel allocated by the device. The field makes sense only with some peripheral buses, like ISA. It is not used outside of the device driver itself, but for informational purposes (in ifconfig).

```
unsigned long state;
```

Device state. The field includes several flags. Drivers do not normally manipulate these flags directly; instead, a set of utility functions has been provided. These functions will be discussed shortly when we get into driver operations.

```
struct net_device *next;
```

Pointer to the next device in the global linked list. This field shouldn't be touched by the driver.

```
struct net_device *next;
int (*init)(struct net_device *dev);
```

The device methods

As happens with the char and block drivers, each network device declares the functions that act on it. Operations that can be performed on network interfaces are listed in this section. Some of the operations can be left NULL, and some are usually untouched because ether_setup assigns suitable methods to them.

Device methods for a network interface can be divided into two groups: fundamental and optional. Fundamental methods include those that are needed to be able to use the interface; optional methods implement more advanced functionalities that are not strictly required. The following are the fundamental methods:

```
int (*open)(struct net_device *dev);
```

Opens the interface. The interface is opened whenever ifconfig activates it. The open method should register any system resource it needs (I/O ports, IRQ, DMA, etc.), turn on the hardware, and increment the module usage count.

```
int (*stop)(struct net_device *dev);
```

Stops the interface. The interface is stopped when it is brought down; operations performed at open time should be reversed.

```
int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
```

This method initiates the transmission of a packet. The full packet (protocol headers and all) is contained in a socket buffer (sk_buff) structure.

3.3.2. Socket Buffer

Whenever the kernel needs to transmit a data packet, it calls the hard_start_transmit method to put the data on an outgoing queue. Each packet handled by the kernel is contained in a socket buffer structure (struct sk_buff), whose definition is found in <linux/skbuff.h>. The structure gets its name from the Unix abstraction used to represent a network connection, the socket. Even if the interface has nothing to do with sockets, each network packet belongs to a socket in the higher network layers, and the input/output buffers of any socket are lists of struct sk_buff structures. The same sk_buff structure is used to host network data throughout all the Linux network subsystems, but a socket buffer is just a packet as far as the interface is concerned.

The socket buffer is a complex structure, and the kernel offers a number of functions to act on it. The functions are described later in "The Socket Buffers"; for now a few basic facts about sk_buff are enough for us to write a working driver.

The socket buffer passed to hard_start_xmitcontains the physical packet as it should appear on the media, complete with the transmission-level headers. The interface doesn't need to modify the data being transmitted. skb->data points to the packet being transmitted, and skb->len is its length, in octets.

The Important Fields

The fields introduced here are the ones a driver might need to access. They are listed in no particular order.

```
struct net_device *rx_dev;
struct net_device *dev;
```

The devices receiving and sending this buffer, respectively.

```
union { /* ... */ } h;
union { /* ... */ } nh;
union { /*... */} mac;
```

Pointers to the various levels of headers contained within the packet. Each field of the unions is a pointer to a different type of data structure. h hosts pointers to transport layer headers (for example, struct tcphdr *th); nh includes network layer headers (such as struct iphdr *iph); and mac collects pointers to link layer headers (such as struct ethdr *ethernet).

If your driver needs to look at the source and destination addresses of a TCP packet, it can find them in skb->h.th. See the header file for the full set of header types that can be accessed in this way. Note that network drivers are responsible for setting the mac pointer for incoming packets. This task is normally handled by ether_type_trans, but non-Ethernet drivers will have to set skb->mac.raw directly, as shown later in "Non-Ethernet Headers".

```
unsigned char *head;
unsigned char *data;
unsigned char *tail;
unsigned char *end;
```

Pointers used to address the data in the packet. head points to the beginning of the allocated space, data is the beginning of the valid octets (and is usually slightly greater than head), tail is the end of the valid octets, and end points to the maximum address tail can reach. Another way to look at it is that the available buffer space is skb->end - skb->head, and the currently used data space is skb->tail - skb->data.

```
unsigned long len;
```

The length of the data itself (skb->tail - skb->data).

```
unsigned char ip_summed;
```

The checksum policy for this packet. The field is set by the driver on incoming packets, as was described in "Packet Reception".

```
unsigned char pkt_type;
```

Packet classification used in delivering it. The driver is responsible for setting it to PACKET_HOST (this packet is for me), PACKET_BROADCAST, PACKET_MULTICAST, or PACKET_OTHERHOST (no, this packet is not for me). Ethernet drivers don't modify pkt_type explicitly because eth_type_trans does it for them.

The remaining fields in the structure are not particularly interesting. They are used to maintain lists of buffers, to account for memory belonging to the socket that owns the buffer, and so on.

Functions Acting on Socket Buffers

Network devices that use a sock_buff act on the structure by means of the official interface functions. Many functions operate on socket buffers; here are the most interesting ones:

```
struct sk_buff *alloc_skb(unsigned int len, int priority);
```

```
struct sk_buff *dev_alloc_skb(unsigned int len);
```

Allocate a buffer. The alloc_skb function allocates a buffer and initializes both skb->data and skb->tail to skb->head. The dev_alloc_skb function is a shortcut that calls alloc_skb with GFP_ATOMIC priority and reserves some space between skb->head and skb->data. This data space is used for optimizations within the network layer and should not be touched by the driver.

```
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
```

Free a buffer. The kfree_skb call is used internally by the kernel. A driver should use dev_kfree_skb instead, which is intended to be safe to call from driver context.

```
unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);
```

These inline functions update the tail and len fields of the sk_buff structure; they are used to add data to the end of the buffer. Each function's return value is the previous value of skb->tail (in other words, it points to the data space just created). Drivers can use the return value to copy data by invoking ins(ioaddr, skb_put(...)) or memcpy(skb_put(...), data, len). The difference between the two functions is that skb_put checks to be sure that the data will fit in the buffer, whereas __skb_put omits the check.

```
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);
```

These functions decrement skb->data and increment skb->len. They are similar to skb_put, except that data is added to the beginning of the packet instead of the end. The return value points to the data space just created. The functions are used to add a hardware header before transmitting a packet. Once again, __skb_push differs in that it does not check for adequate available space.

```
int skb_tailroom(struct sk_buff *skb);
```

This function returns the amount of space available for putting data in the buffer. If a driver puts more data into the buffer than it can hold, the system panics. Although you might object that a printk would be sufficient to tag the error, memory corruption is so harmful to the system that the developers decided to take definitive action. In practice, you shouldn’t need to check the available space if the buffer has been correctly allocated. Since drivers usually get the packet size before allocating a buffer, only a severely broken driver will put too much data in the buffer, and a panic might be seen as due punishment.

```
int skb_headroom(struct sk_buff *skb);
```

Returns the amount of space available in front of data, that is, how many octets one can "push" to the buffer.

```
void skb_reserve(struct sk_buff *skb, int len);
```

This function increments both data and tail. The function can be used to reserve headroom before filling the buffer. Most Ethernet interfaces reserve 2 bytes in front of the packet; thus, the IP header is aligned on a 16-byte boundary, after a 14-byte Ethernet header. snull does this as well, although the instruction was not shown in "Packet Reception" to avoid introducing extra concepts at that point.

```
unsigned char *skb_pull(struct sk_buff *skb, int len);
```

Removes data from the head of the packet. The driver won’t need to use this function, but it is included here for completeness. It decrements skb->len and increments skb->data;

this is how the hardware header (Ethernet or equivalent) is stripped from the beginning of incoming packets.

The kernel defines several other functions that act on socket buffers, but they are meant to be used in higher layers of networking code, and the driver won't need them.

3.3.3. Installing an Interrupt Handler

If you want to actually "see" interrupts being generated, writing to the hardware device isn't enough; a software handler must be configured in the system. If the Linux kernel hasn't been told to expect your interrupt, it will simply acknowledge and ignore it.

Interrupt lines are a precious and often limited resource, particularly when there are only 15 or 16 of them. The kernel keeps a registry of interrupt lines, similar to the registry of I/O ports. A module is expected to request an interrupt channel (or IRQ, for interrupt request) before using it, and to release it when it's done. In many situations, modules are also expected to be able to share interrupt lines with other drivers, as we will see. The following functions, declared in <linux/sched.h>, implement the interface:

```
int request_irq(unsigned int irq,
   void (*handler)(int, void *,
   struct pt_regs *),
   unsigned long flags,
   const char *dev_name,
   void *dev_id);
   void free_irq(unsigned int irq, void *dev_id);
```

The value returned from request_irq to the requesting function is either 0 to indicate success or a negative error code, as usual. It's not uncommon for the function to return -EBUSY to signal that another driver is already using the requested interrupt line. The arguments to the functions are as follows:

`unsigned int irq`

This is the interrupt number being requested.

`void (*handler)(int, void *, struct pt_regs *)`

The pointer to the handling function being installed. We'll discuss the arguments to this function later in this chapter.

`unsigned long flags`

As you might expect, a bit mask of options (described later) related to interrupt management.

`const char *dev_name`

The string passed to request_irq is used in /proc/interrupts to show the owner of the interrupt.

`void *dev_id`

This pointer is used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). When no sharing is in force, dev_id can be set to NULL, but it a good idea anyway to use this item to point to the device structure. We’ll see a practical use for dev_id in "Implementing a Handler", later in this chapter.

The bits that can be set in flags are as follows:

SA_INTERRUPT SA_SHIRQ SA_SAMPLE_RANDOM

This bit indicates that the generated interrupts can contribute to the entropy pool used by /dev/random and /dev/urandom. These devices return truly random numbers when read and are designed to help application software choose secure keys for encryption. Such random numbers are extracted from an entropy pool that is contributed by various random events. If your device generates interrupts at truly random times, you should set this flag. If, on the other hand, your interrupts will be predictable (for example, vertical blanking of a frame grabber), the flag is not worth setting -- it wouldn’t contribute to system entropy anyway. Devices that could be influenced by attackers should not set this flag; for example, network drivers can be subjected to predictable packet timing from outside and should not contribute to the entropy pool. See the comments in drivers/char/random.cfor more information.

The interrupt handler can be installed either at driver initialization or when the device is first opened. Although installing the interrupt handler from within the module’s initialization function might sound like a good idea, it actually isn’t. Because the number of interrupt lines is limited, you don’t want to waste them. You can easily end up with more devices in your computer than there are interrupts. If a module requests an IRQ at initialization, it prevents any other driver from using the interrupt, even if the device holding it is never used. Requesting the interrupt at device open, on the other hand, allows some sharing of resources.

It is possible, for example, to run a frame grabber on the same interrupt as a modem, as long as you don't use the two devices at the same time. It is quite common for users to load the module for a special device at system boot, even if the device is rarely used. A data acquisition gadget might use the same interrupt as the second serial port. While it's not too hard to avoid connecting to your Internet service provider (ISP) during data acquisition, being forced to unload a module in order to use the modem is really unpleasant.

The correct place to call request_irq is when the device is first opened, before the hardware is instructed to generate interrupts. The place to call free_irq is the last time the device is closed, after the hardware is told not to interrupt the processor any more. The disadvantage of this technique is that you need to keep a per-device open count. Using the module count isn't enough if you control two or more devices from the same module.

This discussion notwithstanding, shortrequests its interrupt line at load time. This was done so that you can run the test programs without having to run an extra process to keep the device open. short, therefore, requests the interrupt from within its initialization function (short_init) instead of doing it in short_open, as a real device driver would.

3.4. Serial Line IP (SLIP)

The TCP/IP protocol family runs over a variety of network media: IEEE 802.3 (ethernet) and 802.5 (token ring) LAN's, X.25 lines, satellite links, and serial lines. There are standard encapsulations for IP packets defined for many of these networks, but there is no standard for serial lines. SLIP, Serial Line IP, is a currently a de facto standard, commonly used for point-to-point serial connections running TCP/IP. It is not an Internet standard. Distribution of this memo is unlimited.

SLIP has its origins in the 3COM UNET TCP/IP implementation from the early 1980's. It is merely a packet framing protocol: SLIP defines a sequence of characters that frame IP packets on a serial line, and nothing more. It provides no addressing, packet type identification, error detection/correction or compression mechanisms. Because the protocol does so little, though, it is usually very easy to implement.

SLIP is commonly used on dedicated serial links and sometimes for dialup purposes, and is usually used with line speeds between 1200bps and 19.2Kbps. It is useful for allowing mixes of hosts and routers to communicate with one another (host-host, host-router and router- router are all common SLIP network configurations).

3.4.1. Deficiencies

There are several features that many users would like SLIP to provide which it doesn't. In all fairness, SLIP is just a very simple protocol designed quite a long time ago when these problems were not really important issues. The following are commonly perceived shortcomings in the existing SLIP protocol:

- *addressing:* both computers in a SLIP link need to know each other's IP addresses for routing purposes. Also, when using SLIP for hosts to dial-up a router, the addressing scheme may be quite dynamic and the router may need to inform the dialing host of the host's IP address. SLIP currently provides no mechanism for hosts to communicate addressing information over a SLIP connection.
- *type identification:* SLIP has no type field. Thus, only one protocol can be run over a SLIP connection, so in a configuration of two DEC computers running both TCP/IP and DECnet, there is no hope of having TCP/IP and DECnet share one serial line between them while using SLIP. While SLIP is "Serial Line IP", if a serial line connects two multi-protocol computers, those computers should be able to use more than one protocol over the line.
- *error detection/correction:* noisy phone lines will corrupt packets in transit. Because the line speed is probably quite low (likely 2400 baud), retransmitting a packet is very expensive. Error detection is not absolutely necessary at the SLIP level because any IP application should detect damaged packets (IP header and UDP and TCP checksums should suffice), although some common applications like NFS usually ignore the checksum and depend on he network media to detect damaged packets. Because it takes so long to retransmit a packet which was corrupted by line noise, it would be efficient if SLIP could provide some sort of simple error correction mechanism of its own.
- *compression:* because dial-in lines are so slow (usually 2400bps), packet compression would cause large improvements in packet throughput. Usually, streams of packets in a single TCP connection have few changed fields in the IP and TCP headers, so a simple compression algorithms might just send the changed parts of the headers instead of the complete headers.

3.5. UART

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. The UART takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes.

Serial transmission is commonly used with modems and for non-networked communication between computers, terminals and other devices.There are two primary forms of serial transmission: Synchronous and Asynchronous.

Asynchronous Serial Transmission

Asynchronous transmission allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters in advance and special bits are added to each word which are used to synchronize the sending and receiving units.

When a word is given to the UART for Asynchronous transmissions, a bit called the "Start Bit" is added to the beginning of each word that is to be transmitted. The Start Bit is used to alert the receiver that a word of data is about to be sent, and to force the clock in the receiver into synchronization with the clock in the transmitter. These two clocks must be accurate enough to not have the frequency drift by more than 10% during the transmission of the remaining bits in the word. After the Start Bit, the individual bits of the word of data are sent, with the Least Significant Bit (LSB) being sent first. Each bit in the transmission is transmitted for exactly the same amount of time as all of the other bits, and the receiver “looks” at the wire at approximately halfway through the period assigned to each bit to determine if the bit is a 1 or a 0. For example, if it takes two seconds to send each bit, the receiver will examine the signal to determine if it is a 1 or a 0 after one second has passed, then it will wait two seconds and then examine the value of the next bit, and so on.

The sender does not know when the receiver has “looked” at the value of the bit. The sender only knows when the clock says to begin transmitting the next bit of the word.

When the entire data word has been sent, the transmitter may add a Parity Bit that the transmitter generates. The Parity Bit may be used by the receiver to perform simple error checking. Then at least one Stop Bit is sent by the transmitter.

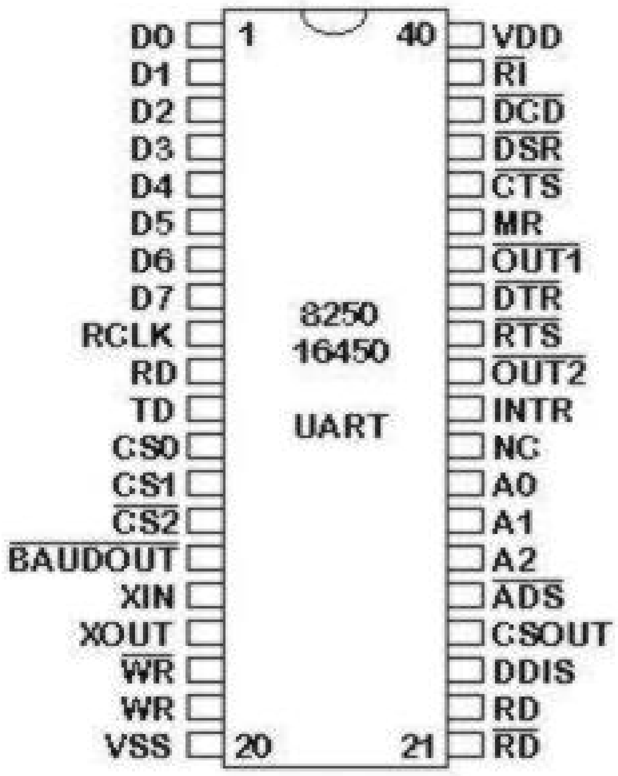


Figure 3-2. UART 8250/16450/16550 Connections

When the receiver has received all of the bits in the data word, it may check for the Parity Bits (both sender and receiver must agree on whether a Parity Bit is to be used), and then the receiver looks for a Stop Bit. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be garbled and will report a Framing Error to the host processor when the data word is read. The usual cause of a Framing Error is that the sender and receiver clocks were not running at the same speed, or that the signal was interrupted.

Regardless of whether the data was received correctly or not, the UART automatically discards the Start, Parity and Stop bits. If the sender and receiver are configured identically, these bits are not passed to the host. If another word is ready for transmission, the Start Bit for the new word can be sent as soon as the Stop Bit for the previous word has been sent. Because asynchronous data is “self synchronizing”, if there is no data to transmit, the transmission line can be idle.

Baud is a measurement of transmission speed in asynchronous communication. Traditionally, a Baud Rate represents the number of bits that are actually being sent over the media, not the amount of data that is actually moved from one DTE device to the other. The Baud count includes the overhead bits Start, Stop and Parity that are generated by the sending UART and removed by the receiving UART. This means that seven-bit words of data actually take 10 bits to be completely transmitted. Therefore, a modem capable of moving 300 bits per second from one place to another can normally only move 30 7-bit words if Parity is used and one Start and Stop bit are present.

If 8-bit data words are used and Parity bits are also used, the data rate falls to 27.27 words per second, because it now takes 11 bits to send the eight-bit words, and the modem still only sends 300 bits per second.

The base address of the UARTs and thus also of the registers are stored in the BIOS data area.

Interface Base Address IRQ

COM1 3f8h IRQ4

COM2 2f8h IRQ3

DLAB stands for Divisor Latch Access Bit. When DLAB is set to ’1’ via the line control register, two registers become available from which you can set your speed of communications measured in bits per second.

Baud rate is calculated as

*Baud rate = Main reference frequency/(16*divisor)*

The UART 8259/16459/16550 comes in a standard DIP case with 40 pins.The registers of UART are given below.

- *Receiver Buffer Register:* It is used to store the data byte received. It’s offset is 0(DLAB=0). The data in this register can be accessed by input functions.
- *Transmitter Hold Register:* If we want to write a byte to Transmitter Hold Register, it is automatically transferred to transmitter shift register and output as a serial data stream. It’s offset is 0(DLAB=0).

- *Interrupt Enable Register:* The Interrupt Enable Register controls the interrupt request. The high order nibble register is always equal to zero. It can not be altered. The first bit of Interrupt Enable Register is for interrupting when a data byte is received. If a data byte is received at the Receiver Buffer Register an interrupt is raised. Similarly the second bit of this register is fot interrupting on Transmitter Buffer empty.
- *Interrupt Identification Register:* With the Interrupt Identification Register, we can detirmine whether an interrupt is currently pending or not. An active interrupt is indicated by a cleare pending bit. This is especially useful when polling is used for the interface concern.
- *Data Format Register:* The eighth bit(DLAB bit) of Data Format Register is Divisor Latch Access bit. If it is set, we can access the Divisor Latch Registers. If it is not set, we can access the receiver or transmitter register. Data Format Register is used to define a data byte. It can be done in several ways. For example it can be like 8-data bit, 1-stop bit and no parity. Other the combinations can be generated by altering the values in the Data Format Register. It's offset is 3.
- *Divisor Latch Registers:* There are two Divisor Latch Registers: Low order Divisor Latch Register and High Order Divisor Latch Register. This Divisor Latch Registers are used to set the baud rate of the transmission. It's offset is 0 and 1(DLAB=1).
- *Modem Control Register:* Modem Control Register supervises the UART modem control logic. The three most significant bits are not used. A reading access always returns a value zero. By setting the loop bit we can enable the feedback loop check of UART. It's offset is 4.
- *Modem Status Register:* Using the Modem Status Register, we can detirmine the status of the RS-232 input signals. It's offset is 6.
- *Serialization Status Register:* This register is used to detirmine whether a data byte is available in the Receiver Buffer Register or whether the Transmitter Buffer is empty. If the TXE bit is zero, then data is still present either in the Transmitter Hold Register or in the Transmitter Shift Register. If the TBE bit is set, then the Transmitter Hold Register is empty, otherwise a data byte is being held there. It's offset is 5.

Chapter 4. Basic Design

This project modifies the Data Link Layer and Physical Layer of the TCP/IP protocol stack. The Data Link Layer is replaced by SLIP protocol. The UART is used as the Physical Layer.

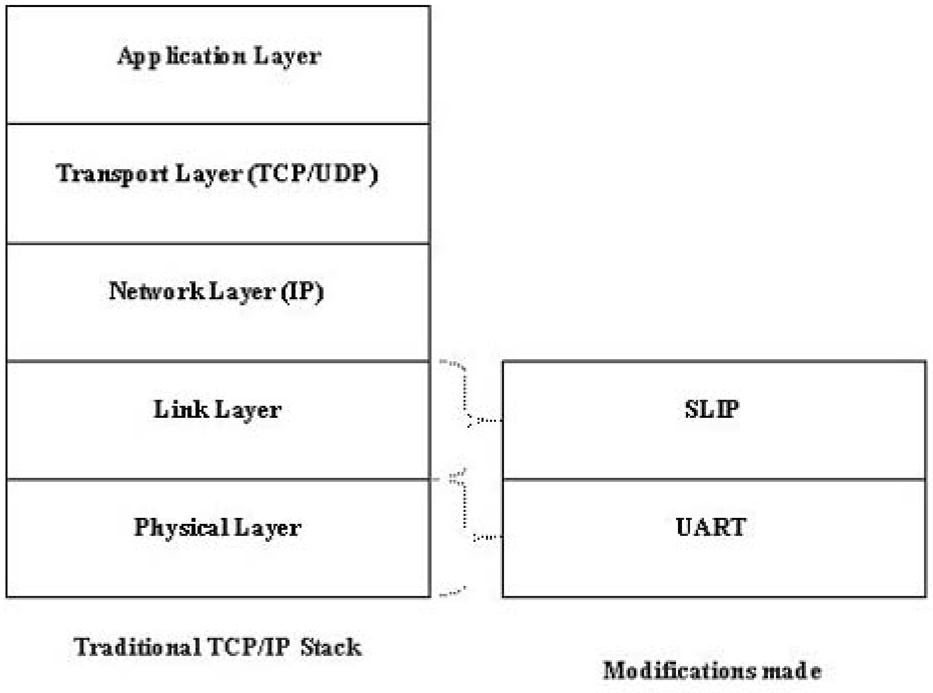


Figure 4-1. TCP stack and modificatins on it

This project is organized as a network driver module.

When the module is inserted in to the kernel, the init_module() function gets executed. In the function init_module() we have to register the network driver object. Then register the interrupt handler. After registering the handler we have to initialize the UART. Similarly when the module is to be removed from the kernel,the function cleanup_module() gets invoked. In this function we have to unregister the network driver and free the interrupt handler.

The UART is programmed to transfer the data at a baud rate of 9600 baud. The data packet contains 8 bits, one stop bit and no parity. The UART is programmed to raise the interrupt, when a new data packet is arrived to the system. The IRQ number of COM1 is 4 and that of COM2 is 3.

The data to be transmitted would be arriving at the network interface, encapsulated in the skbuff structure. This skbuff structure has got an important function. The upper layers use this structure to pass the data to the lower layers. When this structure reaches the data link layer the data field of the skbuff is passed to the send_packet() function. This function reads data from the data field of the skbuff structure character by character. Each character is converted in to a data packet consisting a header and a trailer. This header and trailer is added according to the SLIP protocol. This also performs character stuffing. Then the data packets are transmitted through the serial port.

When the incoming data arrives at the serial port, the interrupt handler gets invoked. The handler calls the function recv_packet(). This function performs the slip decoding. That is this function removes the header and trailer of the data packet, and places the character in to a buffer. When the data reception is complete the interrupt handler allocates the memory for an object of the skbuff structure. It sets the various fields of the skbuff structure and copies the buffer in to the data field of the skbuff object and gives this object to the upper protocol layers using the function netif_rx().

When the network driver is activated using the ifconfig command the function whose adress is given in the 'open' field of the structure net_device gets invoked. This function increments the usage count of the network driver module. Similarly when the network driver is brought down the function whose address is given in the 'stop' field of the structure net_device gets invoked. This function decrements the usage count of the network driver module.

Chapter 5. Implementation Details

5.1. Setting up the Hardware

Most PC interface for serial data exchange follow the RS-232 standard. For this purpose eleven of the RS-232 signals are required. Furthermore, there is a 9 pin connection defined by IBM which has been used in this project.

On the 9-pin connector the protective ground and the signal for the data signal rate are missing as compared to RS-232, but the remaining 9-signals are sufficient for a serial asynchronous data exchange between two DTEs . The pins 3/2 and 2/3 transfer the data signal, the rest of the connections are intended for the control signals.

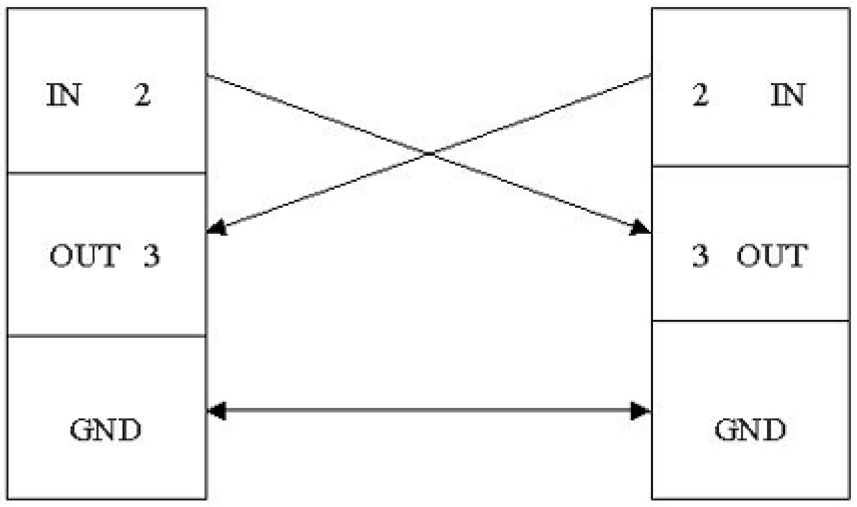


Figure 5-1. Connection diagram

In the above connection diagram Pin 2 is receive, the Pin 3 is transmit and Pin 5 is ground. Join Pin 5 of both connectors with a cable (this is the common ground). Pin 2 of one connector should be joined with Pin 3 of the other and viceversa (This forms RxT and TxR connections).

5.2. Programming the UART

In the C programming language the I/O space address can be accessed using the functions inb() an outb(). The function declaration of inb() and outb() is given below.

```
char inb(char address);
outb(char pattern, char address);
```

The function inb() reads a byte from the address specified.The function outb() writes the pattern to the address specified. Only super user can access this functions. The user space needs a higher privilage level to gain access to I/O space. This can be achieved using the function iopl(). The declaration is given below.

```
int iopl(int level);
```

The call to iopl() changes the input output privilage level to the level specified.

The program should be compiled with the -O option. O stands for the optimization option.

Before using UART it must be properly initialized. The UART programming can be divided into

5.2.1. UART Initialization

The first step in initializing the UART is setting the baud rate. Data format register is used for this purpose. Baud rate=main reference frequency/(16*divisor) In the PC a main reference frequency of 1.8432Mhz is used generated by an external oscillator.

In this case baud rate equal to 115200/divisor. The divisor is stored in the divisor latch register. The divisor latch register is accessed by setting the DLAB bit of data format register. After writing the divisor latch register DLAB bit is cleared.

The next step is to define the data format. Data format register can be used for this purpose. Here the format of 8 data bit, One stop bit and no parity is used.

5.2.2. UART Interrupt Handling

With the interrupt enabled register we can control the interrupt request. Two types of interrupts are possible. In the first case an interrupt is occured when the reived data is ready. In the second case the interrupt is occured when the transmitter buffer is empty. In the Interrupt enable register, the high order nibble register is equal to zero and cannot be altered.

The first bit is used to enable the interrupt of the first type and the second bit is used for enabling the interrupt of the second type. If the interrupts are enabled, the corresponding IRQ lines will be high, whenever an interrupt is occured. COM1 is configured with IRQ4 and COM2 is configured with IRQ4.

Using the interrupt identification register we can identify the interrupt. Two types of interrupts are possible in this case as early mentioned. The second and third bits of this register are used for the interrupt identification.

With modem control register we can set master interrupt bit to enable individual interrupt.

5.2.3. Data transmission and reception

Serialization status register is used to know whether data byte is available at the receiver buffer register. If the data byte is available corresponding data is retrieved. To transmit the data the corresponding data byte is loaded into the transmit hold register, if the transmit buffer empty flag of serialization status register is set. These tests are used to avoid character overrun errors.

5.3. SLIP Encoding and Decoding

As the serial hardware is very simple and does not impose any kind of "packet structure" on data, it is the responsibility of the transmitting program to let the receiver know where a chunk of data begins and where it ends. The simplest way would be to place two "marker" bytes at the begining and end. For this purpose SLIP protocol is used. This encoding scheme is explained in RFC 1055.

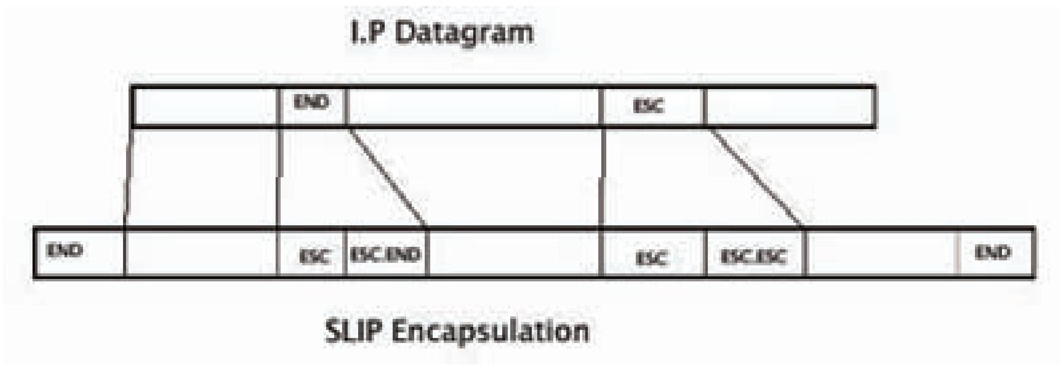


Figure 5-2. SLIP Encapsulation

The SLIP protocol defines two special characters: END and ESC. END is octal 300 (decimal 192) and ESC is octal 333 (decimal 219) not to be confused with the ASCII ESCape character; for the purposes of this discussion, ESC will indicate the SLIP ESC character. To send a packet, a SLIP host simply starts sending the data in the packet. If a data byte is the same code as END character, a two byte sequence of ESC and ESC_END octal 334 (decimal 220) is sent instead. If it the same as an ESC character, an two byte sequence of ESC and ESC_ESC octal 335 (decimal 221) is sent instead. When the last byte in the packet has been sent, an END character is then transmitted.

The function send_packet() is used to encode the data using SLIP protocol. The function prototype is void send_packet(unsigned char *p, int len); This function reads character by character from the buffer p and perform the encoding and transmit the data packet over the serial line.

The function recv_packet() is used for SLIP decoding. The function prototype is given below. void recv_packet(void); This function is coded in a finite state machine model. The decoded characters are stored into slip_buffer[] array.

5.4. Network Driver

5.4.1. Initializing net_device

The function mydev_init() is used to initialize the net_device structure. The prototype of the function is int mydev_init(struct net_device *dev); The important fields of net_device needed are given below open: Which is used to store the address of the function which gets invoked when the network driver is activated. Here address of the function mydev_open() is stored in the field open. stop: Which is used to store the address of the function which gets invoked when the network driver is disabled. Here address of the function mydev_release() is stored in the field open. mtu: Maximum Transmission Unit is used to store the size of maximum size of the data packet. hard_start_xmit: This field is used to store the address of the function who transmit the data over the serial port. Here the address of the function mydev_xmit() is stored, which inturn calls the function send_packet for transmitting data. type: This field is used to store the type of the protocol used. Here the flag ARPHRD_SLIP is stored.

5.4.2. Registering net_device object

After performig the initialization of net_device we have to register the object for making it as a standard network interface. For this we have to create an object of net_device say mydev. In the 'init' field of mydev we have to place the address of the function which performs the initialization of the net_device objet. This can be achieve by using the code given below. struct net_device mydev = {init: mydev_init}; Then register the object mydev using the function register_netdev(). If the registering function return a nonzero value, there might be some error. If the function returns a zero, it indicates that the network driver is successfully registred.

5.4.3. Registering IRQ

The function request_irq() is used to request the IRQ. The first argument is the IRQ number, for which the interrupt handler is going to be installed. The second argument is the address of the interrupt handler routine. The third argument is a flag. Fourth argument is the name of the owner and the last argument is a pointer to the driver object. The prototype is given below. int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long flags, const char *dev_name, void *dev_id); Here the flag is set to SA_INTERRUPT, which indicates a "fast" interrupt handler.

When the module is removed from the kernel, the IRQ line must be freed. The function free_irq() is used for this purpose. The prototype is given below. void free_irq(unsigned int irq, void *dev_id);

5.4.4. mydev_open()

Function prototype is given below int mydev_open(struct net_device *) This function is called when the networkdriver is activated. This function increments the usage count of the module using the macro MOD_INC_USE_COUNT. This function also starts a queue using the function netif_start_queue(). This queue is used for the communication with the upper layer protocols.

5.4.5. mydev_release()

Function prototype is given below int mydev_release(struct net_device *) This function is called when the networkdriver is disabled. This function decrements the usage count of the module using the macro MOD_DEC_USE_COUNT. This function also stops the queue which was created by mydev_open() for the communication with the upper layers using the function netif_stop_queue().

5.4.6. mydev_xmit()

Function prototype is given below static int mydev_xmit(struct sk_buff, struct net_device *); This function is called when the data is ready to transmit. This function has 2 arguments. The first argument is a pointer to the skbuff structure, which contains the data to be transmitted. This function calls the function calls the function send_packet() for encoding the using SLIP protocol an the transmission of data packets. Finally this routinr free s the memory used by the skbuff object using dev_kfree_skb().

5.4.7. Interrupt handler

the function uart_int_handler() is acting as the interrupt handler. Whenever an interrupt is occured, this function gets called. Then the interrupt identification register is checked to know whether it is a sending interrupt or a receiving interrupt. According to the interrupt the corresponding sending or receiving function is called.

If the interrupt occured is a receiving interrupt then the following actions are executed.An object for the structure skbuff is created and memory is allocated to the object using the function dev_alloc_skb(). The various fields of the skbuff object are set accordingly. The received data which is in slip_buffer[] ic copied to the data field of the struct skbuff object. Then the struct skbuff object is given to the upper layers using the function netif_rx() function.

5.4.8. Getting the Statistical Information

The ifconfig displays the number of received/transmitted packets, total number of bytes received/transmitted etc. To get the above statistical information, we have to do the following. The net_device structure contains a "private" pointer field, which can be used for holding information. We will allocate an object of type struct net_device_stats and store it’s address in the private data area. As and when we receive/transmit data, we will update certain fields of this structure. When ifconfig wants to get statistical information about interface, it will call a function whose address is stored in the get_stats field of the net_device object. This function should simply return the address of the net_device object which holds the statistical information.

The important fields of the structure net_device_stats are tx_bytes : Which gives the number of transmitted bytes. rx_bytes : Which gives the number of received bytes. tx_packets : Which gives the number of transmitted packets rx_packets : Which gives the number of received packets.

When we transmit or receive the data packets we will update these fields.

Chapter 6. Configuring & Installing

The project installation involves the configuring of network drivers and gateway which are explained in detail in the forthcoming sections.Each section is also provided with the configuration as per our network configuration involving four systems.

6.1. Configuring the Network Driver

The ifconfig command is used for manipulating network interfaces. Here is what the command displays on our machine :

lo Link encap:Local Loopback inet addr:127.0.0.1 Mask:255.0.0.0 UP LOOPBACK RUNNING MTU:16436 Metric: RX packets:0 errors:0 dropped:0 overruns:0 frame:0 TX packets:0 errors:0 dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:0 RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

This machine does not have any real networking hardware installed - but we do have a puresoftware interface - a so called "loopback interface". The interface is assigned an IP addressof 127.0.0.1.

It is possible to bring the interface down by running *ifconfig lo down*. Once the interface is down, ifconfig will not display it in it’s output But it is possible to obtain information about inactive interfaces by running *ifconfig -a*. It is possible make the interface active once again by running *ifconfig lo up*it is also possible to assign a different IP address - *ifconfig lo 127.0.0.2*.

Before an interface can be manipulated with ifconfig, it is necessary that the driver code for the interface is loaded into the kernel. In the case of the loopback interface, the code is compiled into the kernel. Usually, it would be stored as a module and inserted into the kernel whenever required by running commands like modprobe.

Implementation

In this project we have written a network driver for creating two modules named serial1 and serial2. This modules are inserted into the running kernel by running the command *insmod -f serial1.o* and *insmod -f serial2.o*. The insertion of two modules into the kernel results in the creation of two interfaces to which IP address can be assigned.

Once the IP address are properly assigned by running the command *ifconfig serial1 IP-address* and *ifconfig serial2 IP-address* respectively for serial1 and serial2, both the interfaces can be made active running the command *ifconfig serial1 up* and *ifconfig serial2* *up*.After doing all the steps mentioned above we can use the interfaces serial1 and serial2 for communication.

6.2. Configuring the Gateway

Although this project is intended to develop a peer to peer connection by means of the serial port, it can also be extended to a network of more than two computers.A simple way of doing this is to configure the routing table of each node.We configure the network in such a way that the system adjacent to the any system will be its gateway to go into the other the network.

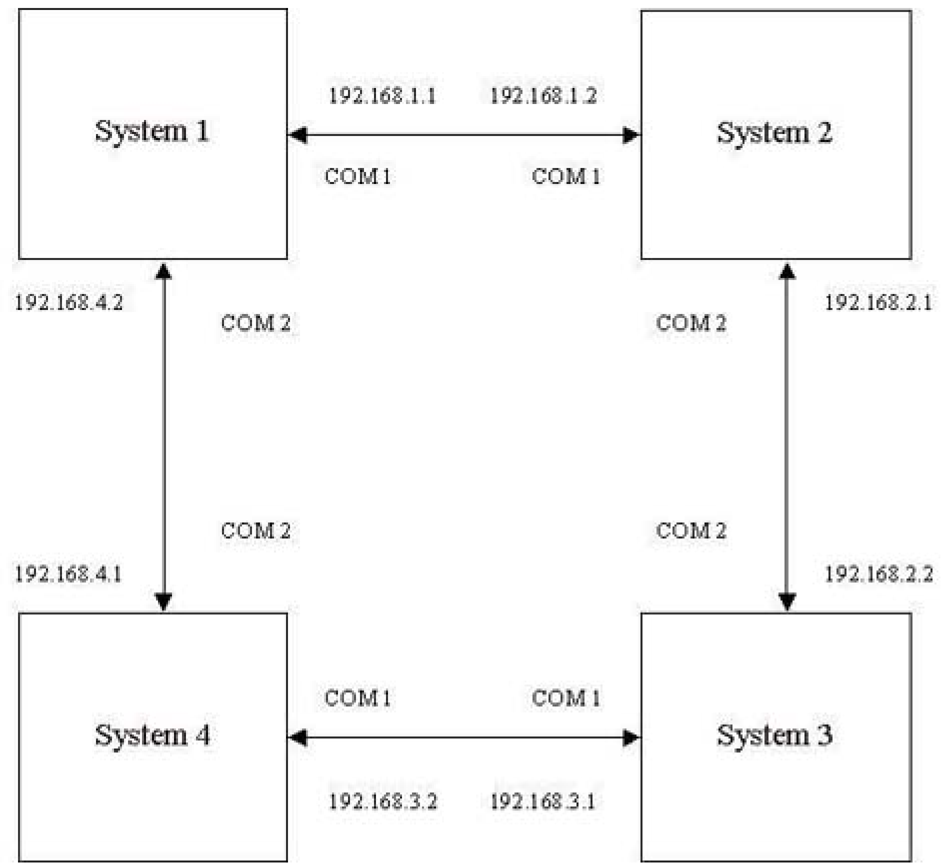


Figure 6-1. A Simple Network of Four Systems

Each system has it’s own routing teble , it can be seen by running the command *route*Suppose our network configuration is as shown in the figure above.Here each system is loaded with two drivers named *serial1* and *serial2* by the steps explained in the previous section.The IP-address are assigned to the system as shown in the figure.

In the current topology shown above in the figure different network exists between two systems as listed below

192.168.1.0 between system1 and system2

192.168.2.0 between system2 and system3

192.168.3.0 between system3 and system4

192.168.4.0 between system4 and system1

Here all systems has to forward packet to the system adjacent to it, if the corresponding packet doesnot belong to that particular system.Inorder to accomplish that we have to run the command *echo 1 >/etc/sys/ip/ip_forward*.Now inorder to establish a full fledged network between four networks we have to configure each system individually.

Implementation

Configuring System1

Here system1 is directly connected to system2 and system4 ,ie it has a direct connection to networks 192.168.2.0 and 192.168.4.0. Inorder to reach system3 we have to configure either system2 or system4 as gateway.Here system2 is taken as gateway.It is done by running the command *route add net 192.168.2.0 mask 255.255.255.0 -dev serial1*

Configuring System2

Here system2 is directly connected to system1 and system3 ,ie it has a direct connection to networks 192.168.1.0 and 192.168.3.0.Inorder to reach system4 we have to configure either system1 or system3 as gateway. Here system3 is taken as gateway.It is done by running the command *route add net 192.168.4.0 mask 255.255.255.0 -dev serial2.*

Configuring System3

Here system3 is directly connected to system2 and system4 ,ie it has a direct connection to networks 192.168.2.0 and 192.168.4.0.Inorder to reach system1 we have to configure either system2 or system4 as gateway.Here system4 is taken as gateway.It is done by running the command *route add net 192.168.1.0 mask 255.255.255.0 -dev serial1.*

Configuring System4

Here system4 is directly connected to system1 and system3 ,ie it has a direct connection to networks 192.168.1.0 and 192.168.3.0.Inorder to reach system2 we have to configure either system1 or system3 as gateway.Here system1 is taken as gateway.It is done by running the command *route add net 192.168.2.0 mask 255.255.255.0 -dev serial2.*

6.3. Observation

Connection to all systems is checked by running the command Ping. By running ping it is observed that the packet takes 1 hop to reach from system1 to System2 and System4 whereas it takes 2 hops to reach System3. Same is the case with all the other systems, it takes 1 hop to reach its adjacent systems and takes 2 hops to reach the other system in our network configuration.

Chapter 7. Future Scope

Since the networking through serial port is bit slow, same idea can be implemented through the USB port which is quite fast as compared to the serial port.Implementing networking by means of the USB port involves the writtng of a USB driver. This USB driver is embedded with the network driver written for serialport.

7.1. Some Basic Concepts of USB Port

The Universal Serial bus was originally designed with following intentions:

- Connection of the PC to the telephone
- Ease-of-use
- Port expansion.

The USB is strictly hierarchical and it is controlled by one host. The host uses a master / slave protocol to communicate with attached USB devices. This means that every kind of communication is initiated by the host and devices cannot establish any direct connection to other devices. This seems to be a drawback in comparison to other bus architectures but it is not because the USB was designed as a compromise of costs and performance. The master / slave protocol solves implicitly problems like collision avoidance or distributed bus arbitration. The current implementation of the USB allows 127 devices to be connected at the same time and the total communication bandwidth is limited to 12Mbit/s. Howewer use of low speed devices, management of USB "interrupts" and other overheads mean that actual throughput cannot exceed about 8.5Mbit/s under near ideal conditions, and typical performance may be around 2Mbit/s.

7.2. Usb Devices and Transfer Characteristics

There are a wide range of USB devices intended for a wide range of purposes, and this means that implementation details can vary widely.

A device can be self powered, bus powered or both. The USB can provide a power supply up to 500mA for its devices. If there are only bus powered devices on the bus the maximum power dissipation could be exceeded and therefore self powered devices exist. They need to have their own power supply. Devices that support both power types can switch to self powered mode when attaching an external power supply.

Even the maximum communication speed can differ for particular USB devices. The USB specification differentiates between low speed and full speed devices. Low speed devices (such as mice, keyboards, joysticks etc.) communicate at 1.5MBit/s and have only limited capabilities. Full speed devices (such as audio and video systems) can use up to 90% of the 12Mbit/s which is about 10Mbit/s including the protocol overhead.

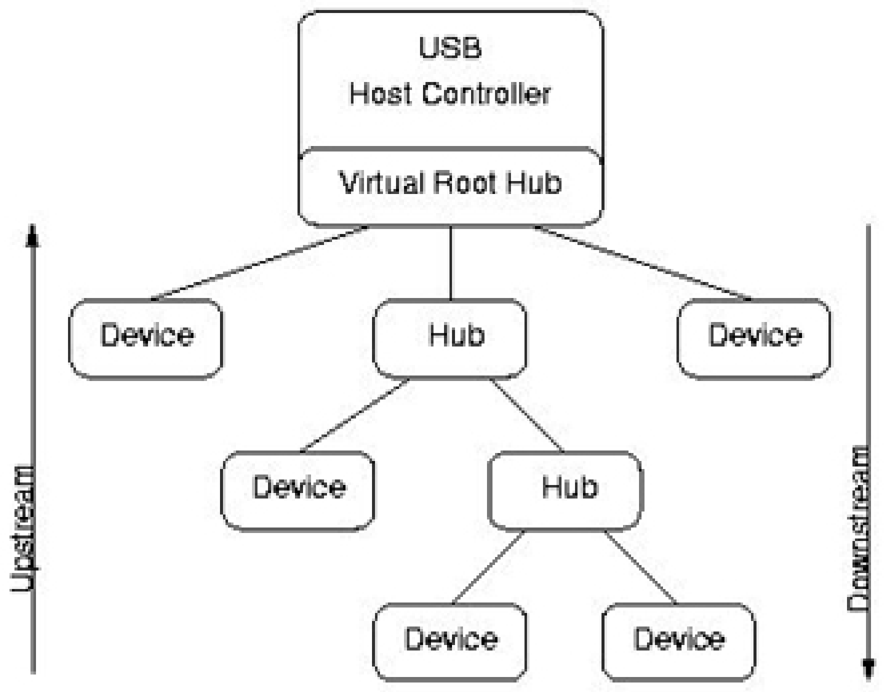


Figure 7-1. USB Topology

7.3. Data Flow Types

The communication on the USB is done in two directions and uses four different transfer types. Data directed from the host to a device is called downstream or OUT transfer. The other direction is called upstream or IN transfer. Depending on the device type different transfer variants are used:

- Control transfers are used to request and send reliable short data packets. It is used to configure devices and every one is required to support a minimum set of control commands.

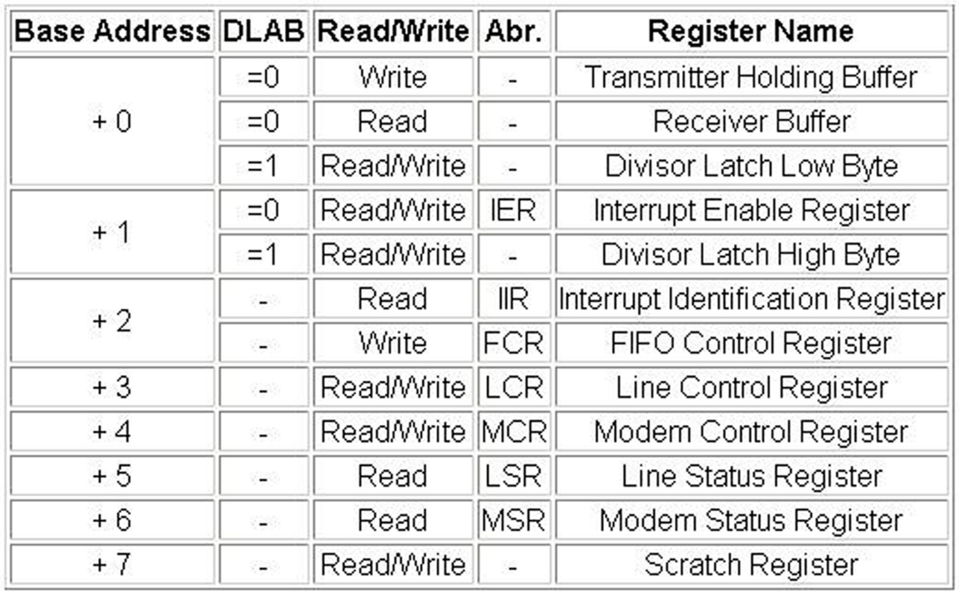
- Bulk transfers are used to request or send reliable data packets up to the full bus bandwidth. Devices like scanners or scsi adapters use this transfer type.
- Interrupt transfers are similar to bulk transfers which are polled periodically. If an interrupt transfer was submitted the host controller driver will automatically repeat this request in a specified interval (1ms - 127ms).
- Isochronous transfers send or receive data streams in realtime with guaranteed bus bandwidth but without any reliability. In general these transfer types are used for audio and video devices

Chapter 8. Conclusion

This project can be used extensively inorder to establish adhoc networks between systems. It provides a cheaper alternative for ethernet cards in situations where we are forced to establish a network temporarily for some specific purpose.Even though the serial port network is slow as compared to the ethernetcard, but provided the cost benefits it can be used very effectively.

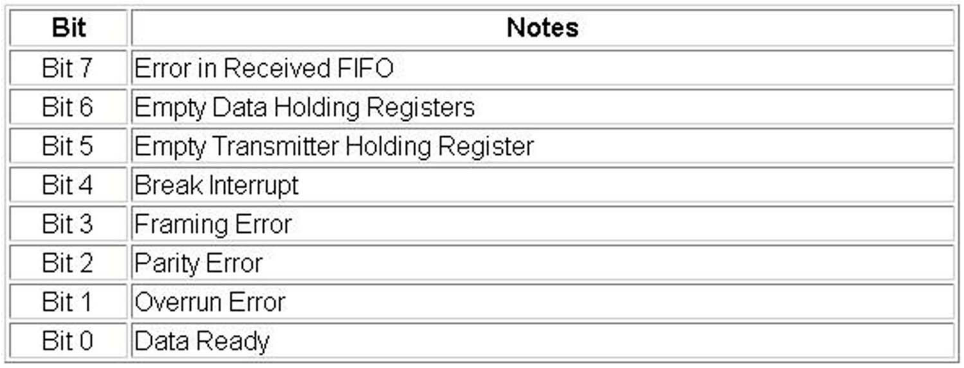
Besides this, this project is also a good example for those who are interested in understanding the complexities of the network interface. This project provides a study model for those interested in networking. Since the serial ports are used for communication, the data transmission rate is low. But the project demonstrate how powerful the network drivers are.

Appendix A. UART Registers



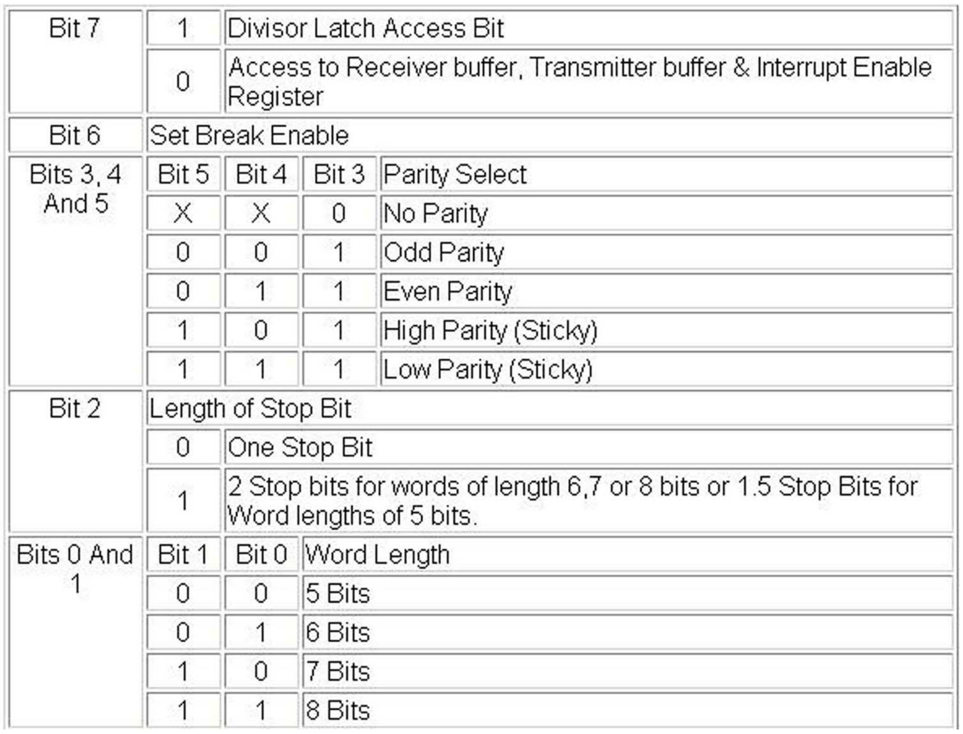
Base Address	DLAB	Read/Write	Abr.	Register Name
+ 0	=0	Write	-	Transmitter Holding Buffer
	=0	Read	-	Receiver Buffer
	=1	Read/Write	-	Divisor Latch Low Byte
+ 1	=0	Read/Write	IER	Interrupt Enable Register
	=1	Read/Write	-	Divisor Latch High Byte
+ 2	-	Read	IIR	Interrupt Identification Register
	-	Write	FCR	FIFO Control Register
+ 3	-	Read/Write	LCR	Line Control Register
+ 4	-	Read/Write	MCR	Modem Control Register
+ 5	-	Read	LSR	Line Status Register
+ 6	-	Read	MSR	Modem Status Register
+ 7	-	Read/Write	-	Scratch Register

Figure A-1. UART Registers



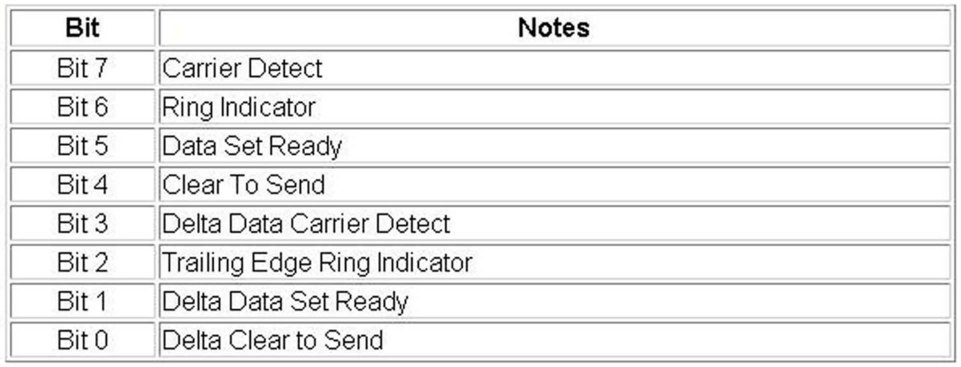
Bit	Notes
Bit 7	Error in Received FIFO
Bit 6	Empty Data Holding Registers
Bit 5	Empty Transmitter Holding Register
Bit 4	Break Interrupt
Bit 3	Framing Error
Bit 2	Parity Error
Bit 1	Overrun Error
Bit 0	Data Ready

Figure A-2. Line Status Register



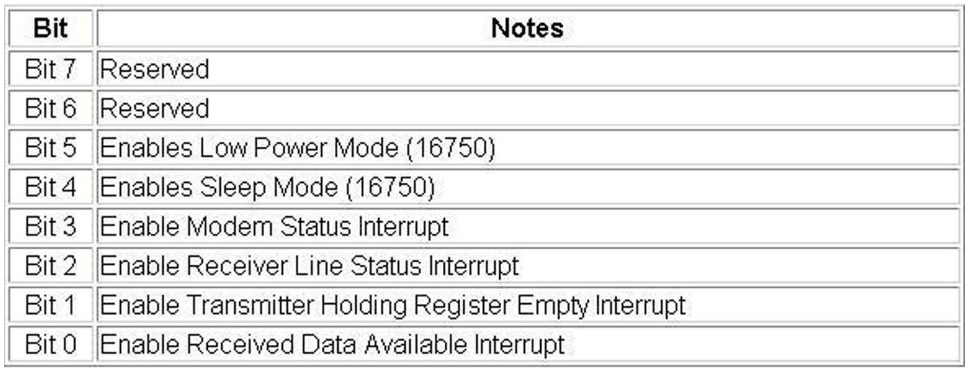
Bit 7	1	Divisor Latch Access Bit		
	0	Access to Receiver buffer, Transmitter buffer & Interrupt Enable Register		
Bit 6	Set Break Enable			
Bits 3, 4 And 5	Bit 5	Bit 4	Bit 3	Parity Select
	X	X	0	No Parity
	0	0	1	Odd Parity
	0	1	1	Even Parity
	1	0	1	High Parity (Sticky)
	1	1	1	Low Parity (Sticky)
Bit 2	Length of Stop Bit			
	0	One Stop Bit		
	1	2 Stop bits for words of length 6,7 or 8 bits or 1.5 Stop Bits for Word lengths of 5 bits.		
Bits 0 And 1	Bit 1	Bit 0	Word Length	
	0	0	5 Bits	
	0	1	6 Bits	
	1	0	7 Bits	
	1	1	8 Bits	

Figure A-3. Line Control Register



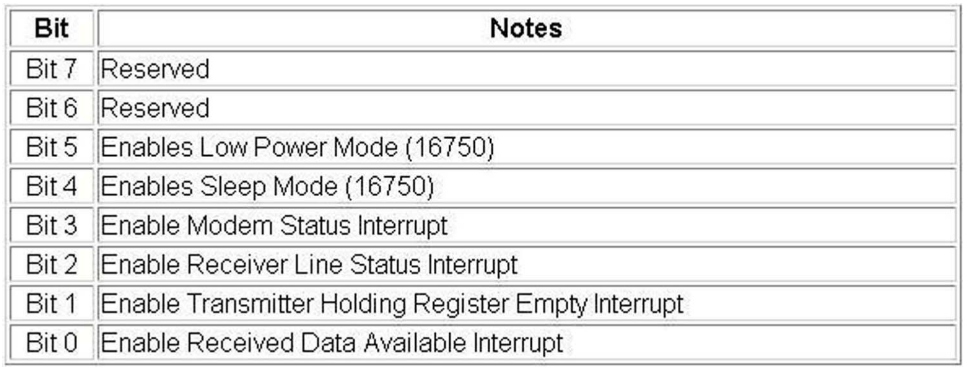
Bit	Notes
Bit 7	Carrier Detect
Bit 6	Ring Indicator
Bit 5	Data Set Ready
Bit 4	Clear To Send
Bit 3	Delta Data Carrier Detect
Bit 2	Trailing Edge Ring Indicator
Bit 1	Delta Data Set Ready
Bit 0	Delta Clear to Send

Figure A-4. Modem Status Register



Bit	Notes
Bit 7	Reserved
Bit 6	Reserved
Bit 5	Enables Low Power Mode (16750)
Bit 4	Enables Sleep Mode (16750)
Bit 3	Enable Modem Status Interrupt
Bit 2	Enable Receiver Line Status Interrupt
Bit 1	Enable Transmitter Holding Register Empty Interrupt
Bit 0	Enable Received Data Available Interrupt

Figure A-5. Modem Control Register



Bit	Notes
Bit 7	Reserved
Bit 6	Reserved
Bit 5	Enables Low Power Mode (16750)
Bit 4	Enables Sleep Mode (16750)
Bit 3	Enable Modem Status Interrupt
Bit 2	Enable Receiver Line Status Interrupt
Bit 1	Enable Transmitter Holding Register Empty Interrupt
Bit 0	Enable Received Data Available Interrupt

Figure A-6. Interrupt Enable Register

Appendix B. RFC of SLIP Protocol

Network Working Group J. Romkey

Request for Comments: 1055 June l988

A NONSTANDARD FOR TRANSMISSION OF IP DATAGRAMS OVER SERIAL LINES: SLIP

B.1. Introduction

The TCP/IP protocol family runs over a variety of network media: IEEE 802.3 (ethernet) and 802.5 (token ring) LAN’s, X.25 lines, satellite links, and serial lines. There are standard encapsulations for IP packets defined for many of these networks, but there is no standard for serial lines. SLIP, Serial Line IP, is a currently a de facto standard, commonly used for point-to-point serial connections running TCP/IP. It is not an Internet standard. Distribution of this memo is unlimited.

B.2. History

SLIP has its origins in the 3COM UNET TCP/IP implementation from the early 1980’s. It is merely a packet framing protocol: SLIP defines a sequence of characters that frame IP packets on a serial line, and nothing more. It provides no addressing, packet type identification, error detection/correction or compression mechanisms. Because the protocol does so little, though, it is usually very easy to implement.

Around 1984, Rick Adams implemented SLIP for 4.2 Berkeley Unix and Sun Microsystems workstations and released it to the world. It quickly caught on as an easy reliable way to connect TCP/IP hosts and routers with serial lines.

SLIP is commonly used on dedicated serial links and sometimes for dialup purposes, and is usually used with line speeds between 1200bps and 19.2Kbps. It is useful for allowing mixes of hosts and routers to communicate with one another (host-host, host-router and router- router are all common SLIP network configurations).

B.3. Aaailability

SLIP is available for most Berkeley UNIX-based systems. It is included in the standard

4.3BSD release from Berkeley. SLIP is available for Ultrix, Sun UNIX and most other Berkeley-derived UNIX systems. Some terminal concentrators and IBM PC implementations also support it.

SLIP for Berkeley UNIX is available via anonymous FTP from uunet.uu.net in pub/sl.shar.Z. Be sure to transfer the file in binary mode and then run it through the UNIX uncompress program. Take the resulting file and use it as a shell script for the UNIX /bin/sh (for instance, /bin/sh sl.shar).

B.4. Protocol

The SLIP protocol defines two special characters: END and ESC. END is octal 300 (decimal 192) and ESC is octal 333 (decimal 219) not to be confused with the ASCII ESCape character; for the purposes of this discussion, ESC will indicate the SLIP ESC character. To send a packet, a SLIP host simply starts sending the data in the packet. If a data byte is the same code as END character, a two byte sequence of ESC and octal 334 (decimal 220) is sent instead. If it the same as an ESC character, an two byte sequence of ESC and octal 335 (decimal 221) is sent instead. When the last byte in the packet has been sent, an END character is then transmitted.

Phil Karn suggests a simple change to the algorithm, which is to begin as well as end packets with an END character. This will flush any erroneous bytes which have been caused by line noise. In the normal case, the receiver will simply see two back-to-back END characters, which will generate a bad IP packet. If the SLIP implementation does not throw away the zero-length IP packet, the IP implementation certainly will. If there was line noise, the data received due to it will be discarded without affecting the following packet.

Because there is no ’standard’ SLIP specification, there is no real defined maximum packet size for SLIP. It is probably best to accept the maximum packet size used by the Berkeley UNIX SLIP drivers: 1006 bytes including the IP and transport protocol headers (not including the framing characters). Therefore any new SLIP implementations should be prepared to accept 1006 byte datagrams and should not send more than 1006 bytes in a datagram.

B.5. Deficiencies

There are several features that many users would like SLIP to provide which it doesn’t. In all fairness, SLIP is just a very simple protocol designed quite a long time ago when these problems were not really important issues. The following are commonly perceived shortcomings in the existing SLIP protocol:

- *addressing:* both computers in a SLIP link need to know each other's IP addresses for routing purposes. Also, when using SLIP for hosts to dial-up a router, the addressing scheme may be quite dynamic and the router may need to inform the dialing host of the host's IP address. SLIP currently provides no mechanism for hosts to communicate addressing information over a SLIP connection.
- *type identification:* SLIP has no type field. Thus, only one protocol can be run over a SLIP connection, so in a configuration of two DEC computers running both TCP/IP and DECnet, there is no hope of having TCP/IP and DECnet share one serial line between them while using SLIP. While SLIP is "Serial Line IP", if a serial line connects two multi-protocol computers, those computers should be able to use more than one protocol over the line.
- *error detection/correction:* noisy phone lines will corrupt packets in transit. Because the line speed is probably quite low (likely 2400 baud), retransmitting a packet is very expensive. Error detection is not absolutely necessary at the SLIP level because any IP application should detect damaged packets (IP header and UDP and TCP checksums should suffice), although some common applications like NFS usually ignore the checksum and depend on the network media to detect damaged packets. Because it takes so long to retransmit a packet which was corrupted by line noise, it would be efficient if SLIP could provide some sort of simple error correction mechanism of its own.
- *compression:* because dial-in lines are so slow (usually 2400bps), packet compression would cause large improvements in packet throughput. Usually, streams of packets in a single TCP connection have few changed fields in the IP and TCP headers, so a simple compression algorithms might just send the changed parts of the headers instead of the complete headers.

Some work is being done by various groups to design and implement a successor to SLIP which will address some or all of these problems.

B.6. SLIP Drivers

The following C language functions send and receive SLIP packets. They depend on two functions, send_char() and recv_char(), which send and receive a single character over the serial line.

```
/* SLIP special character codes
 */
#define END             0300    /* indicates end of packet */
#define ESC             0333    /* indicates byte stuffing */
#define ESC_END         0334    /* ESC ESC_END means END data byte */
#define ESC_ESC         0335    /* ESC ESC_ESC means ESC data byte */

/* SEND_PACKET: sends a packet of length "len", starting at
 * location "p".
 */
void send_packet(p, len)
        char *p;
        int len; {

  /* send an initial END character to flush out any data that may
   * have accumulated in the receiver due to line noise
   */
     send_char(END);

  /* for each byte in the packet, send the appropriate character
   * sequence
   */
        while(len--) {
                switch(*p) {
                /* if it’s the same code as an END character, we send a
                 * special two character code so as not to make the
                 * receiver think we sent an END
                 */
                case END:
                        send_char(ESC);
                        send_char(ESC_END);
                        break;

                /* if it’s the same code as an ESC character,
                 * we send a special two character code so as not
                 * to make the receiver think we sent an ESC
                 */
                case ESC:
                        send_char(ESC);
```

```
                                send_char(ESC_ESC);
                                break;
                        /* otherwise, we just send the character
                         */
                        default:
                                send_char(*p);
                                }

                        p++;
                        }

                /* tell the receiver that we’re done sending the packet
                 */
                send_char(END);
                }

        /* RECV_PACKET: receives a packet into the buffer located at "p".
         *      If more than len bytes are received, the packet will
         *      be truncated.
         *      Returns the number of bytes stored in the buffer.
         */
        int recv_packet(p, len)
                char *p;
                int len; {
                char c;
                int received = 0;

                /* sit in a loop reading bytes until we put together
                 * a whole packet.
                 * Make sure not to copy them into the packet if we
                 * run out of room.
                 */
                while(1) {
                        /* get a character to process
                         */
                        c = recv_char();

                        /* handle bytestuffing if necessary
                         */
```

```
switch(c) {

/* if it’s an END character then we’re done with
 * the packet
 */
case END:
        /* a minor optimization: if there is no
         * data in the packet, ignore it. This is
         * meant to avoid bothering IP with all
         * the empty packets generated by the
         * duplicate END characters which are in

         * turn sent to try to detect line noise.
         */
        if(received)
                return received;
        else
                break;

/* if it’s the same code as an ESC character, wait
 * and get another character and then figure out
 * what to store in the packet based on that.
 */
case ESC:
        c = recv_char();

        /* if "c" is not one of these two, then we
         * have a protocol violation.  The best bet
         * seems to be to leave the byte alone and
         * just stuff it into the packet
         */
        switch(c) {
        case ESC_END:
                c = END;
                break;
        case ESC_ESC:
                c = ESC;
                break;
                }
                /* here we fall into the default handler and let
                 * it store the character for us
                 */
                default:
                        if(received < len)
                                p[received++] = c;
                        }
                }
        }
```

Appendix C. References

1. Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O‘Reilly & Associates. 2000.

2. Bach, Maurice. *The Design of the Unix Operating System*. Prentice Hall. 1987.

3. Stevens, Richard. *Unix Network Programming.*

4. Stevens, Richard. *Advanced Programming in the UNIX Environment.*

5. Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel*. O‘Reilly & Associates. 2000.

6. http://www.ker nel.org

7. http://www.linuxdoc.org

8. http://www.ker nelnotes.org

9. http://www.linux.it/kerneldocs.