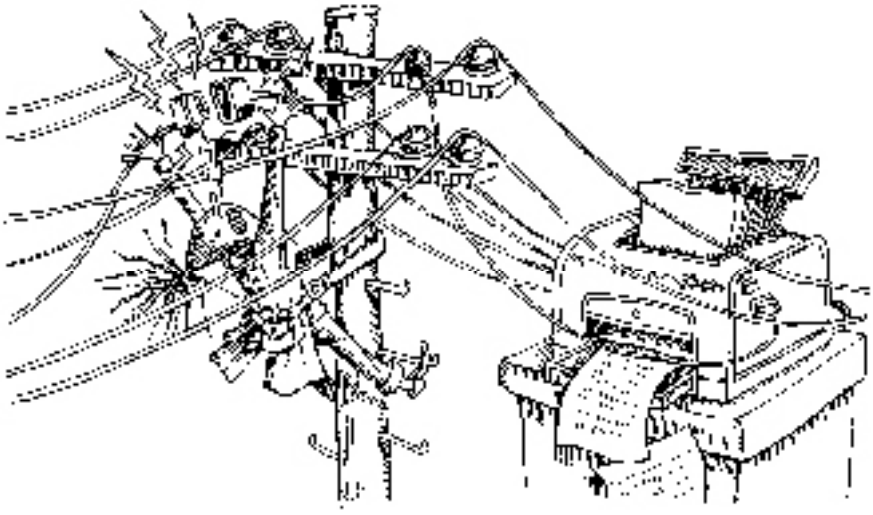
50 Serial Devices and Terminals



An operating system with over 40 years of history is sure to be dragging some cruft along with it. Some would put support for serial devices into this category, arguing that it's a technology from a bygone era that is best forgotten. Compared to today's multi-megabit serial interfaces such as USB, traditional serial ports may indeed seem too slow and twiddly to be useful.

In fact, an understanding of serial interfaces is an essential component of any system administrator's tool box. For better or worse, the UNIX command-line interface is based on the ancient concept of a serial terminal and the associated commands and control structures remain in use today. Even if you have never been within 50 paces of a hardwired terminal, you're still using the same basic OS facilities that supported it. For example, the console window on your UNIX or Linux desktop is really a pseudo-terminal, as is the device to which you appear to be connected when you log in through the network.

Actual RS-232C serial ports are still around, too. They're no longer the general facility they used to be, but they remain important in several situations. They're the common denominator for bootstrapping all types of hardware devices, from lights-out enterprise-class server managers to embedded systems the size of a thumbnail, including custom hardware projects. They're a medium you can use to communicate

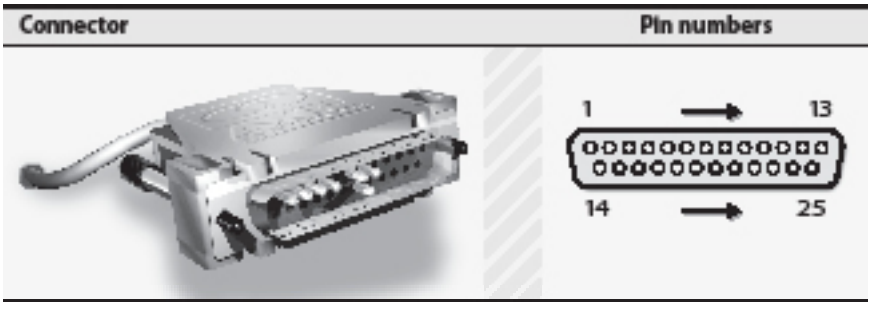
with legacy systems. There are even cases in which you might run into an actual hardwired terminal, such as on a manufacturing floor.

This chapter describes how to connect and use RS-232-based serial devices in the modern world. The first few sections address serial hardware and cabling considerations. Then, starting on page 1452, we talk about the software infrastructure that supports both hardwired terminals and the pseudo-terminals that emulate them. Finally, we cover the use of a UNIX or Linux system to communicate with the serial consoles of other devices.

50.1 THE RS-232C STANDARD

Most slow-speed serial ports conform to some variant of the RS-232C standard. This standard specifies the electrical characteristics and meaning of each signal wire, as well as the pin assignments on the traditional 25-pin (DB-25p) serial connector shown in Exhibit A.

Exhibit A A male DB-25 connector



Full RS-232C[1] is never used in real-world situations since it defines numerous signals that are unnecessary for basic communication. DB-25 connectors are also inconveniently large. As a result, 9-pin DB-9 connectors are now commonly used instead of the original 25-pin flavor. In cases where structured cabling is used, RJ-45 connectors are also a convenient alternative. Both of these connectors are described in the section titled *Alternative connectors* starting on page 1446.

Exhibit A shows a male DB-25. As with all serial connectors, the pin numbers on a female connector are a mirror image of those on a male connector so that like-numbered pins mate. The diagram is drawn from the orientation shown, as if you were facing the end of the cable, about to plug the connector into your forehead.

Note that in Exhibit A, only seven pins are actually installed, which is typical. The RS-232 signals and their pin assignments on a full-size DB-25 connector are shown

1. To be technically correct, this standard should now be referred to as EIA-232-E. However, no one will have the slightest idea what you are talking about.

in Table 50.1. Only the shaded signals are ever used in practice (at least on computer systems); all others can be ignored.

Table 50.1 RS-232 signals and pin assignments on a DB-25

Pin	Name	Function	Pin	Name	Function
1	FG	Frame ground	14	STD	Secondary TD
2	TD	Transmitted data	15	TC	Transmit clock
3	RD	Received data	16	SRD	Secondary RD
4	RTS	Request to send	17	RC	Receive clock
5	CTS	Clear to send	18	–	Not assigned
6	DSR	Data set ready	19	SRTS	Secondary RTS
7	SG	Signal ground	20	DTR	Data terminal ready
8	DCD	Data carrier detect	21	SQ	Signal quality detector
9	–	Positive voltage	22	RI	Ring indicator
10	–	Negative voltage	23	DRS	Data rate selector
11	–	Not assigned	24	SCTE	Clock transmit external
12	SDCD	Secondary DCD	25	BUSY	Busy
13	SCTS	Secondary CTS			

Unlike connector standards such as USB and Ethernet that were designed to be mostly idiot-proof, RS-232 requires you to know what types of devices you are connecting. Two interface configurations exist: DTE (Data Terminal Equipment) and DCE (Data Communications Equipment). DTE and DCE share the same pinouts, but they specify different interpretations of the RS-232 signals.

Every device is configured as either DTE or DCE; a few devices support both, but not simultaneously. Computers, terminals, and printers are generally DTE, and most modems are DCE. DTE and DCE serial ports can communicate with each other in any combination, but different combinations require different cabling.

There is no sensible reason for both DTE and DCE to exist; all equipment could use the same wiring scheme. The existence of two conventions is merely one of the many pointless historical legacies of RS-232.

DTE and DCE can be confusing if you let yourself think about the implications too much. When that happens, just take a deep breath and reread these points:

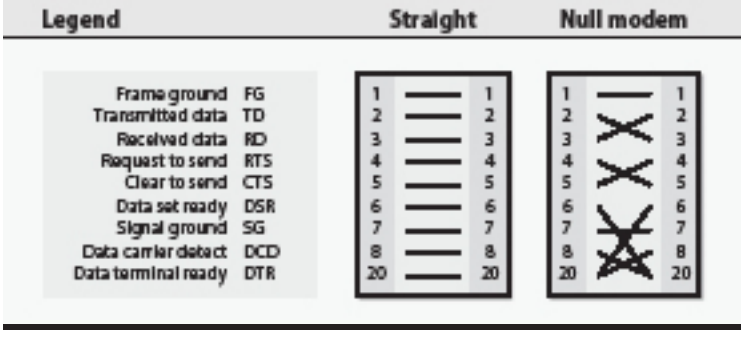
- The RS-232 pinout for a given connector type is always the same, regardless of whether the connector is male or female (matching pin numbers always mate) and regardless of whether the connector is on a cable, a DTE device, or a DCE device.
- All RS-232 terminology is based on the model of a straight-through connection from a DTE device to a DCE device. By "straight through," we

mean that TD on the DTE end is connected to TD on the DCE end, and so on. Each pin connects to the same-numbered pin on the other end.

- Signals are named relative to the perspective of the DTE device. For example, the name TD (transmitted data) really means "data transmitted from DTE to DCE." Despite the name, the TD pin is an *input* on a DCE device. Similarly, RD is an input for DTE and an output for DCE.
- When you wire DTE equipment to DTE equipment (computer-to-terminal or computer-to-computer), you must trick each device into thinking the other is DCE. For example, both DTE devices expect to transmit on TD and receive on RD. You must cross-connect the wires so that one device's transmit pin goes to the other's receive pin, and vice versa.
- Three sets of signals must be crossed in this fashion for DTE-to-DTE communication (if you choose to connect them at all). TD and RD must be crossed. RTS and CTS must be crossed. And each side's DTR pin must be connected to both the DCD and DSR pins of the peer.
- To add to the confusion, a cable crossed for DTE-to-DTE communication is often called a "null modem" cable. You might be tempted to use a null modem cable to hook up a modem, but since modems are DCE, that won't work! A cable for a modem is called a "modem cable" or a "straight cable."

Exhibit B shows pin assignments and connections for both null-modem and straight-through cables. Only signals used in the real world are shown.

Exhibit B Pin assignments and connections for DB-25 cables



50.2 Alternative connectors

The following sections describe the most common modern connector systems, DB-9 and RJ-45. Despite their physical differences, these connectors provide access to

the same electrical signals as a DB-25. Devices that use different connectors are always compatible if the right kind of converter cable is used.

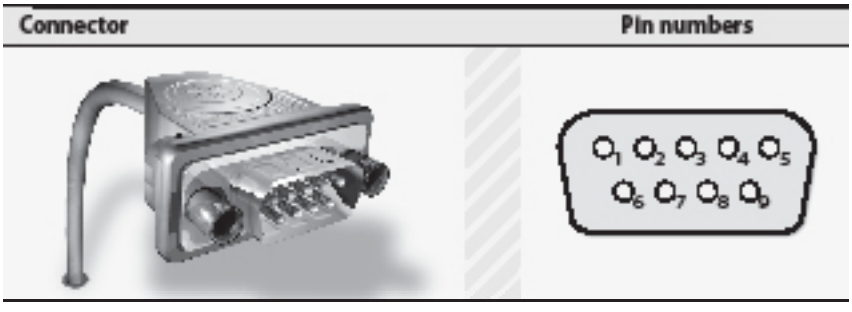
The DB-9 variant

The DB-9 is the most common modern-day embodiment of RS-232. It's a 9-pin connector that looks like a "DB-25 junior" and supplies the eight most commonly used signals. Pin 9 is left unconnected.

Table 50.2 DB-9 Pinout

DB-9	Signal	Function
1	DCD	Data carrier detect
2	RD	Received data
3	TD	Transmitted data
4	DTR	Data terminal ready
5	SG	Signal ground
6	DSR	Data set ready
7	RTS	Request to send
8	CTS	Clear to send

Exhibit C A male DB-9 connector



The RJ-45 variant

An RJ-45 is an 8-wire modular telephone connector. The use of RJ-45s makes it easy to run serial communications through your building's existing wiring if the wiring plant was installed with twisted-pair Ethernet in mind.

RJ-45 jacks for serial connections are usually not found on computers or on garden-variety serial equipment, but they are often used as intermediate connectors for routing serial lines through patch panels. RJ-45s are compact, self-securing, cheap,

and easy to crimp onto the ends of custom-cut cables. An inexpensive crimping tool is required.

Several systems map the pins on an RJ-45 connector to those on a DB-25. Table 50.3 shows the official RS-232D standard, which is used only haphazardly.

Table 50.3 Pins for an RJ-45 to DB-25 straight cable

RJ-45	DB-25	Signal	Function
1	6	DSR	Data set ready
2	8	DCD	Data carrier detect
3	20	DTR	Data terminal ready
4	7	SG	Signal ground
5	3	RD	Received data
6	2	TD	Transmitted data
7	5	CTS	Clear to send
8	4	RTS	Request to send

Exhibit D A male RJ-45 connector

A well-thought-out standard for RJ-45 to DB-25 wiring was created by Dave Yost. If you're planning to use a significant amount of serial cabling, be sure to check it out at yost.com/computers/RJ45-serial.

50.3 Hard and soft carrier

UNIX expects to see the DCD signal, carrier detect, go high (positive voltage) when a serial device is attached and turned on. If your serial cable has a DCD line and your computer really pays attention to it, you are using what is known as hard carrier. Most systems also allow soft carrier; that is, the computer pretends that DCD is always asserted.

For certain devices (such as traditional hardwired terminals), soft carrier is a great blessing. You can get away with using only three wires for each serial connection: transmit, receive, and signal ground. However, modem connections really need the DCD signal. If a terminal is connected through a modem and the carrier signal is lost, the modem should hang up (especially on a long distance call!).

You can specify soft carrier for a serial port in the configuration file for whatever client software you use in conjunction with the port (e.g., **gettydefs** or **inittab** for a login terminal or **printcap** for a printer). You can also use **stty -clocal** to enable soft carrier on the fly.

For example,

```
suse$ sudo stty -clocal < /dev/ttyS1
```

enables soft carrier for the port **ttyS1**.

50.4 HARDWARE FLOW CONTROL

The CTS and RTS signals make sure that a device does not send data faster than the receiver can process it. For example, if a modem is in danger of running out of buffer space (perhaps because the connection to the remote site is slower than the serial link between the local machine and the modem), it can tell the computer to shut up until more room becomes available in the buffer.

Flow control is essential for high-speed modems and is also very useful for serial printers. On systems that do not support hardware flow control (either because the serial ports do not understand it or because the serial cable leaves CTS and RTS disconnected), flow control can sometimes be simulated in software with the ASCII characters XON and XOFF. However, software flow control must be explicitly supported by high-level software, and even then it does not work very well.

XON and XOFF are <Control-Q> and <Control-S>, respectively. This is a problem for **emacs** users because <Control-S> is the default key binding for the **emacs** search command. To fix the problem, bind the search command to another key or use **stty start** and **stty stop** to change the terminal driver's idea of XON and XOFF.

Most terminals ignore the CTS and RTS signals. By jumpering pins 4 and 5 together at the terminal end of the cable, you can fool the few terminals that require a handshake across these pins before they will communicate. When the terminal sends out a signal on pin 4 saying "I'm ready," it gets the same signal back on pin 5 saying "Go ahead." You can also jumper the DTR/DSR/DCD handshake like this.

As with soft carrier, hardware flow control can be set through configuration files or with the **stty** command.

On Sun hardware, flow control for built-in serial ports must be set up with the **eeprom** command.

On some HP platforms, you may need to set flow control for built-in serial ports with the Guardian Service Processor (GSP).

50.5 SERIAL DEVICE FILES

Serial ports are represented by device files in or under **/dev**. Even today, many computers have one or two serial ports built in, mainly as a communication mechanism of last resort. In the past, such ports were usually known by names such as **/dev/ttya** and **/dev/ttyb**, but naming conventions have diverged over time, and those ports are now often named **/dev/ttyS0** or**/dev/tty1**.

Sometimes, more than one device file refers to the same serial port. For example, **/dev/cua/a** on a Solaris system refers to the same port as **/dev/term/a**. However, the minor device number for **/dev/cua/a** is different:

```
solaris$ ls -lL /dev/term/a /dev/cua/a
crw- - - - - - - -    1      uucp     uucp     37, 131072     Jan 11
    16:35 /dev/cua/a
crw- rw- rw-          1      root     sys      37, 0   Jan 11 16:35 /dev/
    term/a
```

As always, the names of the device files do not really matter. Device mapping is determined by the major and minor device numbers, and the names of device files are merely a convenience for human users.

Multiple device files are primarily used to support modems that handle both incoming and outgoing calls. In the Solaris scheme, the driver allows **/dev/term/a** to be opened only when DCD has been asserted by the modem, indicating the presence of an active (inbound) connection (assuming that soft carrier is not enabled on the port). **/dev/cua/a** can be opened regardless of the state of DCD; it's used when connecting to the modem to instruct it to place a call. Access to each device file is blocked while the other is in use.

On HP-UX, serial device files are not always created automatically. You can use the **ioscan** command to force the system to look for them, something like

```
hp-ux$ sudo ioscan -C tty -fn
```

You can then create the device files with

```
hp-ux$ sudo mksf -H port-from-ioscan-output -d asio0 -a0 -i -v
```

AIX appears to be moving away from supporting serial interfaces entirely. In particular, if you have a system with multiple LPARs (see Chapter XXX), serial interfaces are not available by default. You may have to purchase special hardware to obtain serial connectivity in this case.

50.6 SETSERIAL: SET SERIAL PORT PARAMETERS UNDER LINUX

The serial ports on a PC can appear at several different I/O port addresses and interrupt levels (IRQs). These settings might be configured through the system's BIOS, or they might be set automatically through plug and play (PnP) code at boot time. On rare occasions, you may need to change a serial port's address and IRQ settings to accommodate some cranky piece of hardware that is finicky about its own settings and only works correctly when it has co-opted the settings normally used by a serial port. Unfortunately, the serial driver may not be able to detect such configuration changes without your help.

The traditional UNIX response to such diversity is to allow the serial port parameters to be specified when the kernel is compiled. Fortunately, Linux lets you skip this tedious step and change the parameters on the fly with the **setserial** command. **setserial -g** shows the current settings.

```
ubuntu$ setserial -g /dev/ttyS0
/dev/ttyS0, UART: 16550A, Port: 0x03f8, IRQ: 4
```

To set the parameters, you specify the device file and then a series of parameters and values. For example, the command

```
ubuntu$ sudo setserial /dev/ttyS1 port 0x02f8 irq 3
```

sets the I/O port address and IRQ for **ttyS1**. It's important to keep in mind that this command does not change the hardware configuration in any way; it simply informs the Linux serial driver of the configuration. To change the actual settings of the hardware, consult your system's BIOS.

setserial changes only the current configuration, and the settings do not persist across reboots. Unfortunately, there isn't a standard way to make the changes permanent; each of our example distributions does it differently.

The **/etc/init.d/setserial** script on Ubuntu systems is used for serial port initialization. It reads parameters for each port from **/var/lib/setserial/autoserial.conf**.

SUSE's **/etc/init.d/serial** script handles serial port initialization. Unfortunately, this script has no configuration file; you must edit it directly to reflect the commands you want to run. Bad SUSE! The script uses its own little metalanguage to construct the **setserial** command lines, but fortunately there are plenty of commented-out example lines to choose from.

Red Hat's **/etc/rc.d/rc.sysinit** script checks for the existence of **/etc/rc.serial** and executes it at startup time if it exists. No example file is provided, so you must create the file yourself if you want to make use of this feature. Just list the **setserial** commands you want to run, one per line. For completeness, it's probably a good idea to make the file executable and to put **#!/bin/sh** on the first line; however, these *touches d'élégance* aren't strictly required.

50.7 PSEUDO-TERMINALS

Hardwired CRT terminals may be nothing more than museum fodder these days, but their spirit lives on in the form of pseudo-terminals. These pairs of device files emulate a text terminal interface on behalf of services such as virtual consoles, virtual terminals (e.g., **xterm**), and network login services like **telnet** and **ssh**.

Here's how it works. Each of the of the paired device files accesses the same device driver inside the kernel. The slave device is named something like **/dev/ttyp1**. A process that would normally interact with a physical terminal, such as a shell, uses the slave device in place of a physical device such as **/dev/ttyS0**. A host process such as **sshd** or **telnetd** opens the corresponding master device—in this example, **/dev/ptyp1**. The pseudo-terminal device driver shuttles keystrokes and text output between the two devices, hiding the fact that no physical terminal exists.

Although pseudo-terminals don't need a baud rate or flow control strategy, most of the other terminal attributes and settings covered in this chapter apply to them.

The **expect** scripting language uses a pseudo-terminal to control a process (such as **ftp** or **parted**) that expects to interact with a human user. It is quite useful for automating certain types of sysadmin tasks.

50.8 CONFIGURATION OF TERMINALS

Cheap computers have replaced ASCII terminals. However, even the "terminal" windows on a graphical display (such as **xterm**) use the same drivers and configuration files as real terminals, so system administrators still benefit by understanding how this archaic technology works.

Terminal configuration involves two main tasks: making sure that a process is attached to a terminal to accept logins, and making sure that information about the terminal is available once a user has logged in. Before we dive into the details of these tasks, however, let's look at the entire login process.

The login process

*See page XXX for more information about the **init** daemon.*

The login process involves several different programs, the most important of which is the **init** daemon. One of **init**'s jobs is to spawn a process, known generically as a **getty** (but not on Solaris, which calls it a **ttymon**), on each terminal port that is turned on in the **/etc/ttys** or **/etc/inittab** file. The **getty** sets the port's initial characteristics (such as speed and parity) and prints a login prompt.

The actual name of the **getty** program varies among Linux distributions, and some distributions include multiple implementations. Red Hat and SUSE use a simplified version called **mingetty** to handle logins on virtual consoles. To manage terminals and dial-in modems, they provide Gert Doering's **mgetty** implementation. Ubuntu uses a single **getty** written by Wietse Venema et al.; this version is also available on SUSE systems under the name **agetty**. An older implementation called **uugetty**

has largely been superseded by **mgetty**. Finally, HylaFAX (hylafax.org), a popular open source fax server, has its own version of **getty** called **faxgetty**.

To distinguish among this plenitude of **getty**s, think of them in order of complexity. **mingetty** is the simplest and is essentially just a placeholder for a **getty**. It can only handle logins on Linux virtual consoles. **agetty** is a bit more well-rounded and handles both serial ports and modems. **mgetty** is the current king of the hill. It handles incoming faxes as well as logins and does proper locking and coordination so that the same modem can be used as both a dial-in and a dial-out line.

The sequence of events in a complete login is as follows:

- **getty** prints a login prompt (along with the contents of the **/etc/issue** file on Linux systems).
- A user enters a login name at **getty**'s prompt.
- **getty** runs the **login** program with the specified name as an argument.
- **login** requests a password and validates the account against **/etc/shadow** or an administrative database system such as NIS or LDAP.
- **login** prints the message of the day from **/etc/motd** and runs a shell.
- The shell executes the appropriate startup files.[1]
- The shell prints a prompt and waits for input.

When the user logs out, control returns to **init**, which wakes up and spawns a new **getty** on the terminal port.

Files in **/etc** control the characteristics associated with each terminal port. These characteristics include the presence of a login prompt and **getty** process on the port, the baud rate to expect, and the type of terminal that is assumed to be connected to the port.

Unfortunately, terminal configuration is one area where there is little agreement among vendors. Table 50.4 lists the files used by each system.

Table 50.4 Terminal configuration files

System	On/off	Terminal type	Parameters	Monitor
Ubuntu [a]	**/etc/event.d/***tty* [b]	**/etc/ttytype**	**/etc/gettydefs**	**getty**
SUSE	**/etc/inittab**	**/etc/ttytype**	**/etc/gettydefs**	**getty**
Red Hat	**/etc/inittab**	**/etc/ttytype**	**/etc/gettydefs**	**getty**
Solaris [c]	**_sactab**	**_sactab**	**zsmon/_pmtab**	**ttymon**
HP-UX	**/etc/inittab**	**/etc/ttytype**	**/etc/gettydefs**	**getty**
AIX [d]	**/etc/inittab**	**/etc/security/login.cfg**	ODM database	**getty**

a. Ubuntu has moved from **init** to **upstart** for TTY/**getty** management; see page 1456.
b. Virtual consoles are defined in **/etc/default/console-setup**.
c. Solaris configuration files are in **/etc/saf** and should be managed with **sacadm**.
d. To ensure consistency, use SMIT to modify TTY parameters on AIX.

1. **.profile** for **sh** and **ksh**; **.bash_profile** and **.bashrc** for **bash**; **.cshrc** and **.login** for **csh/tcsh**.

The /etc/ttytype file

On many systems, terminal type information is kept in a file called **/etc/ttytype**. The format of an entry in **ttytype** is

```
termtype device
```

where *device* is the short name of the device file representing the port and the *termtype* names an entry in the **termcap** or **terminfo** database. When you log in, the TERM environment variable is set to the value of this field.

Here is a sample **ttytype** file:

```
wyseconsole
dialup      ttyi0
dialup      ttyi1
vt320       ttyi2
h19 ttyi3
dialout     ttyi4
```

The /etc/gettytab file

The **gettytab** file associates symbolic names such as `std.9600` with port configuration profiles that include parameters such as speed, parity, and login prompt. Here is a sample:

```
# The default entry, used to set defaults for other entries, and in cases
# where getty is called with no specific entry name.

default:\
    :ap:lm=\r\n%h login\72 :sp#9600:

# Fixed-speed entries

2|std.9600|9600-baud:\
    :sp#9600:
h|std.38400|38400-baud:\
    :sp#38400:
```

The format is the same as that of **printcap** or **termcap**. The lines with names separated by a vertical bar (|) list the names by which each configuration is known. The other fields in an entry set the options to be used with the serial port.

The /etc/gettydefs file

Like **gettytab**, **gettydefs** defines port configurations used by **getty**. A given system will usually have one or the other, never both. The **gettydefs** file looks like this:

```
console# B9600 HUPCL # B9600 SANE IXANY #login: #console
19200# B19200 HUPCL # B19200 SANE IXANY #login: #9600
9600# B9600 HUPCL # B9600 SANE IXANY HUPCL #login: #4800
4800# B4800 HUPCL # B4800 SANE IXANY HUPCL #login: #2400
2400# B2400 HUPCL # B2400 SANE IXANY HUPCL #login: #1200
```

```
1200# B1200 HUPCL # B1200 SANE IXANY HUPCL #login: #300
300# B300 HUPCL # B300 SANE IXANY TAB3 HUPCL #login: #9600
```

The format of an entry is

```
label# initflags # finalflags # prompt #next
```

getty tries to match its second argument with a *label* entry. If it is called without a second argument, the first entry in the file is used. The *initflags* field lists **ioctl**(2) flags that should be set on a port until **login** is executed. The *finalflags* field sets flags that should be used thereafter.

There must be an entry that sets the speed of the connection in both the *initflags* and the *finalflags*. The flags that are available vary by system; check the **gettydefs** or **mgettydefs** man page for authoritative information.

The *prompt* field defines the login prompt, which may include tabs and newlines in backslash notation. The *next* field gives the label of an **inittab** entry that should be substituted for the current one if a break is received. This was useful decades ago when modems didn't negotiate a speed automatically and you had to match speeds by hand with a series of breaks. Today, it's an anachronism. For a hardwired terminal, *next* should refer to the label of the current entry.

Each time you change the **gettydefs** file, you should run **getty -c gettydefs**, which checks the syntax of the file to make sure that all entries are valid.

The /etc/inittab file

See page XXX for more information about the role of ***init****.*

init supports various "run levels" that determine which system resources are enabled. There are seven run levels, numbered 0 to 6, with "s" recognized as a synonym for level 1 (single-user operation). When you leave single-user mode, **init** prompts you to enter a run level unless an `initdefault` field exists in **/etc/inittab** as described below. **init** then scans the **inittab** file for all lines that match the specified run level.

Run levels are usually set up so that you have one level in which only the console is enabled and another level in which all **getty**s are enabled. You can define the run levels in whatever way is appropriate for your system; however, we recommend that you not stray too far from the defaults.

Entries in **inittab** are of the form

```
id:run-levels:action:process
```

Here are some simple examples of **inittab** entries:

```
# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
```

In this format, *id* is a one- or two-character string that identifies the entry; it can be null. For terminal entries, it is customary to use the terminal number as the *id*.

run-levels enumerates the run levels to which the entry pertains. If no levels are specified (as in the first line), then the entry is valid for all run levels. *action* tells how to handle the *process* field; Table 50.5 lists some of the commonly used values.

Table 50.5 Common values for the /etc/inittab action field

Value	Wait?	Meaning
	–	Sets the initial run level
`boot`	No	Runs when **inittab** is read for the first time
`bootwait`	Yes	Runs when **inittab** is read for the first time
`ctrlaltdel`	No	Runs in response to a keyboard <Control-Alt-Delete>[a]
`once`	No	Starts the process once
`wait`	Yes	Starts the process once
`respawn`	No	Always keeps the process running
`powerfail`	No	Runs when **init** receives a power-fail signal
`powerwait`	Yes	Runs when **init** receives a power-fail signal
`sysinit`	Yes	Runs before accessing the console
`off`	–	Terminates the process if it is running, on some systems

a. Linux systems only

If one of the *run-levels* matches the current run level and the *action* field indicates that the entry is relevant, **init** uses **sh** to execute (or terminate) the command specified in the *process* field. The Wait? column in Table 50.5 tells whether **init** waits for the command to complete before continuing.

In the example **inittab** lines above, the last two lines spawn **mingetty** processes on the first two virtual consoles (accessed with <Alt-F1> and <Alt-F2>). If you add hardwired terminals or dial-in modems, the appropriate **inittab** lines look similar to these. However, you must use **mgetty** or **getty** (**agetty** on SUSE) with such devices because **mingetty** is not sophisticated enough to handle them correctly. In general, `respawn` is the correct action and 2345 is an appropriate set of levels.

The command **telinit -q** makes **init** reread the **inittab** file.

getty configuration for Linux

Different **getty**s require different configuration procedures. The **getty**/**agetty** version found on SUSE and Ubuntu is generally a bit cleaner than the **mgetty** version because it accepts all of its configuration information on the command line (in **/etc/inittab**).

The general model is

```
/sbin/getty port speed termtype
```

where *port* is the device file of the serial port relative to **/dev**, *speed* is the baud rate (e.g., 38400), and *termtype* identifies the default terminal type for the port. The *termtype* refers to an entry in the **terminfo** database. Most emulators simulate a DEC VT100, denoted **vt100**. Most of the many other minor options relate to the handling of dial-in modems.

mgetty is a bit more sophisticated than **agetty** in its handling of modems and integrates both incoming and outgoing fax capability. Unfortunately, its configuration is a bit more diffuse. In addition to other command-line flags, **mgetty** can accept an optional reference to an entry in **/etc/gettydefs** that specifies configuration details for the serial driver. Unless you're setting up a sophisticated modem configuration, you can usually get away without a **gettydefs** entry.

Use **man mgettydefs** to find the man page for the **gettydefs** file. It's named this way to avoid conflict with an older **gettydefs** man page that no longer exists on any Linux system.

A simple **mgetty** command line for a hardwired terminal looks like this:

```
/sbin/mgetty -rs speed device
```

speed is the baud rate (e.g., 38400), and *device* is the device file for the serial port (use the full pathname).

If you want to specify a default terminal type for a port when using **mgetty**, you must specify it in a separate file, **/etc/ttytype**, and not on the **mgetty** command line. The format of an entry in **ttytype** is described on page 1454.

Ubuntu Upstart

Ubuntu has replaced its **init** with a rearchitected version called Upstart that starts and stops services in response to events. The executable file for Upstart is still known as **/sbin/init**, however.

Upstart uses one file for each active terminal in **/etc/event.d**. For example, if we wanted a **getty** to run on ttyS0, **/etc/event.d/ttyS0** might look like this:

```
# ttyS0 - getty

# This service maintains a getty on ttyS0 from the point when
# the system is started until it is shut down again.

start on runlevel 2
start on runlevel 3
start on runlevel 4
start on runlevel 5

stop on runlevel 0
stop on runlevel 1
stop on runlevel 6 respawn
```

```
exec /sbin/getty 38400 ttyS0
```

See page XXX for some additional comments on Upstart.

Solaris and sacadm

Rather than traditional UNIX **getty**s that watch each port for activity and provide a login prompt, Solaris has a convoluted hierarchy called the Service Access Facility that controls TTY monitors, port monitors, and many other things that provide a lot of complexity but little added functionality.

To set up a serial port to provide a login prompt, you must first configure a "monitor" that watches the status of the port (**ttymon**). You then configure a port monitor that watches the TTY monitor. For example, to set up a 9,600 baud monitor on **ttyb** to print a login prompt with terminal type VT100, you would use the following commands.

```
solaris$ sudo sacadm -a -p myttymon -t ttymon -c /usr/lib/saf/ttymon -v 1
solaris$ sudo pmadm -a -p myttymon -s b -i root -fu -v 1 -m "`ttyadm -d /
    dev/term/b -l 9600 -T vt100 -s /usr/bin/login`"
```

The **/etc/ttydefs** file is used much like **gettydefs** on other systems to set speed and parity parameters.

See the manual pages for **saf**, **sacadm**, **pmadm**, **ttyadm**, and **ttymon** as well as the terminals chapter in the Solaris AnswerBook for more information about setting up these monitors. Have fun.

50.9 Special characters and the terminal driver

The terminal driver supports several special functions that you access by typing particular keys (usually control keys) on the keyboard. The exact binding of functions to keys can be set with the **tset** and **stty** commands. Table 50.6 lists some of these functions, along with their default key bindings.

Table 50.6 Special characters for the terminal driver

Name	Default	Function
erase	<Control-?>	Erases one character of input
werase	<Control-W>	Erases one word of input
kill	<Control-U>	Erases the entire line of input
eof	<Control-D>	Sends an "end of file" indication
intr	<Control-C>	Interrupts the currently running process
quit	<Control-\>	Kills the current process with a core dump
stop	<Control-S>	Stops output to the screen
start	<Control-Q>	Restarts output to the screen
susp	<Control-Z>	Suspends the current process
lnext	<Control-V>	Interprets the next character literally

Depending on what a vendor's keyboards look like, the default for ERASE might be either <Control-H> or the delete character. (The actual keyboard key may be labeled "backspace" or "delete," or it may show only a backarrow graphic.) Unfortunately, the existence of two different standards for this function creates a multitude of problems.

You can use **stty erase** (see the next section) to tell the terminal driver which key code your setup is actually generating. However, some programs (such as text editors and shells with command-editing features) have their own idea of what the backspace character should be, and they don't always pay attention to the terminal driver's setting. In a helpful but confusing twist, some programs obey both the backspace and delete characters. You may also find that systems you log in to through the network make different assumptions from those of your local system.

Solving these annoying little conflicts can be a Sunday project in itself. In general, there is no simple, universal solution. Each piece of software must be individually beaten into submission. Two useful resources to help with this task are the *Linux Backspace/Delete mini-HOWTO* from tldp.org and a nifty article by Anne Baretta at ibb.net/~anne/keyboard.html. These notes are both written from a Linux perspective, but the problem (and solutions) are not limited to Linux.

50.10 STTY: SET TERMINAL OPTIONS

stty lets you directly change and query the various settings of the terminal driver. There are about a zillion options, but most can be safely ignored. **stty** generally uses the same names for driver options as the **termios** man page does, but occasional discrepancies pop up.

A good combination of options to use for a plain-vanilla terminal is

```
solaris$ stty intr ^C kill ^U erase ^H -tabs
```

Here, **-tabs** prevents the terminal driver from taking advantage of the terminal's built-in tabulation mechanism, a useful practice because many emulators are not very smart about tabs. The other options set the interrupt, kill, and erase characters to <Control-C>, <Control-U>, and <Control-H> (backspace), respectively.

You can use **stty** to examine the current modes of the terminal driver as well as to set them. **stty** with no arguments produces output like this:

```
solaris$ stty
speed 38400 baud;
erase = ^H; eol = M-^?; eol2 = M-^?; swtch = <undef>;
ixany
tab3
```

For a more verbose status report, use the **-a** option:

```
solaris$ stty -a
speed 38400 baud; rows 24; columns 80;
intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = M-^?; eol2
    = M-^?;
swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; dsusp = ^Y; rprnt =
    ^R;
werase = ^W; lnext = ^V; flush = ^O;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon
    -ixoff
-iuclc ixany imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab3 bs0
    vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop
    -echoprt
echoctl echoke
```

The format of the output is similar but lists more information. The meaning of the output should be intuitively obvious if you've written a terminal driver recently.

stty operates on the file descriptor of its standard input, so you can set and query the modes of a terminal other than the current one by using the shell's input redirection character (<). You must be the superuser to change the modes on someone else's terminal.

50.11 TSET: SET OPTIONS AUTOMATICALLY

tset initializes the terminal driver to a mode appropriate for a given terminal type. The type can be specified on the command line; if the type is omitted, **tset** uses the value of the TERM environment variable.

tset supports a syntax for mapping certain values of the TERM environment variable into other values. This feature is useful if you often log in through a modem or data switch and would like to have the terminal driver configured correctly for

the terminal you are really using on the other end of the connection rather than something generic and unhelpful such as "dialup."

For example, suppose that you use **xterm** at home and that the system you are dialing in to is configured to think that the terminal type of a modem is "dialup." Putting the command

```
tset -m dialup:xterm
```

in your **.login** or **.profile** file sets the terminal driver appropriately for **xterm** whenever you dial in.

Unfortunately, the **tset** command is not really as simple as it pretends to be. To have **tset** adjust your environment variables in addition to setting your terminal modes, you need lines something like this:

```
set noglob
eval `tset -s -Q -m dialup:xterm`
unset noglob
```

This incantation suppresses the messages that **tset** normally prints (the **-Q** flag), and asks that shell commands to set the environment be output instead (the **-s** flag). The shell commands printed by **tset** are captured by the backquotes and fed to the shell as input with the built-in command **eval**, causing the commands to have the same effect as if they had been typed by the user.

set noglob prevents the shell from expanding any metacharacters such as * and ? that are included in **tset**'s output. This command is not needed by **sh**/**ksh** users (nor is the **unset noglob** to undo it), since these shells do not normally expand special characters within backquotes. The **tset** command itself is the same no matter what shell you use; **tset** looks at the environment variable SHELL to determine what flavor of commands to print.

50.12 TERMINAL UNWEDGING

Some programs (e.g., **vi**) make drastic changes to the state of the terminal driver while they are running. This meddling is normally invisible to the user, since the terminal state is restored when the program exits or is suspended. However, a program can crash or be killed without performing this housekeeping step. When this happens, the terminal may behave very strangely: it might fail to handle newlines correctly, to echo typed characters, or to execute commands properly.

Another common way to confuse a terminal is to accidentally run **cat** or **more** on a binary file. Most binaries contain a mix of 8-bit characters that is guaranteed to send some of the less-robust emulators into outer space.

To fix this situation, use **reset** or **stty sane**. **reset** is actually just a link to **tset** on many systems, and it can accept most of **tset**'s arguments. However, it is usually run without arguments. Both **reset** and **stty sane** restore the default state of the

terminal driver and send out an appropriate reset code from **termcap**/**terminfo** if one is available.

In many cases for which a **reset** is appropriate, the terminal has been left in a mode in which no processing is done on the characters you type. Most terminals generate carriage returns rather than newlines when the Return or Enter key is pressed. Without input processing, this key generates <Control-M> characters instead of sending off the current command to be executed. To enter newlines directly, use <Control-J> or the line feed key (if there is one) instead of Return.

50.13 Debugging a serial line

Debugging serial lines is not difficult. Here are some typical errors:

- Forgetting to tell **init** to reread its configuration files
- Forgetting to set soft carrier when using three-wire cables
- Using a cable with the wrong nullness
- Soldering or crimping connectors upside down
- Connecting to the wrong wire because of bad or nonexistent wire maps
- Setting the terminal options (including speed) incorrectly

A breakout box is an indispensable tool for debugging serial cabling problems. It is patched into the serial line and shows the signals on each pin as they pass through the cable. The better breakout boxes have both male and female connectors on each side and so are flexible in their positioning. LEDs associated with each "interesting" pin show when the pin is active.

Some breakout boxes are read-only and just let you monitor the signals; others let you rewire the connection and assert a voltage on a particular pin. For example, if you suspect that a cable needs to be nulled (crossed), you can use the breakout box to override the actual cable wiring.

50.14 Connecting to serial device consoles

Perhaps the most common and useful application of RS-232 today is to connect to the serial "console" of another device. The device could be anything from a manageable UPS or network switch to an embedded Linux system such as the TiVo box under your TV. For example, you might connect a serial line to the UPS that powers your equipment rack in a remote data center so that you can shut off power remotely in an emergency.

The basic steps for connecting to a serial console are as follows:

- Attach a cable between the serial port on your UNIX system and the device you want to talk to. See the discussion earlier in this chapter about the various connector types and pinouts that might be necessary. You'll

most likely need a null modem cable. These are available at your nearest computer store.

- Install or identify the terminal communication software you will use on your UNIX or Linux system. Decades ago, the standard command for this was **cu** or **tip**. You can still use these in a pinch, but modern-day alternatives such as **minicom** and **picocom** are better. Linux distributions normally include one of these; on other systems, you may need to install the software yourself (see freshmeat.net/projects/minicom or freshmeat.net/projects/picocom, respectively).
- Configure your communication software to open the correct device file (see the discussion earlier in this chapter). Usually, names like **/dev/ttya**, **/dev/tty1**, **/dev/ttyS0**, or **/dev/S0** are good first guesses.
- Set the baud rate, stop bits, and flow control to match the defaults used on the target device. These parameters are usually outlined in the manual for the device, but you can also try all possible combinations. If you don't know the correct baud rate, an "old dog" trick is to connect and type a few characters. If you have to type multiple characters to get a single character of garbage, you've set the baud rate too high. If typing one or two characters produces many characters of garbage, you've set the baud rate too low. Shhhh… don't tell anyone!
- Once you've successfully connected, you should be able to enter commands on the remote console. If you find that the device suddenly hangs on long output, you have probably misconfigured the flow control; typing <Control-Q> will sometimes get you by.

If you have trouble connecting, the first debugging step should be to remove the crossover in the cable, or to add one if you didn't start with one. Don't forget that if you're connecting to a remote UNIX box, you'll need to set up a **getty** on the far end to listen for your connection and present a login prompt.