Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
FAX 415 969-9131

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 413 2666
Belgium: +32-2-730 38111
Canada: 416 477-6745
Finland: 358-0-502 27 00
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 221-7021
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-831-5568
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000

- Verify the configuration of a terminal port; port 16 in this example.

```
Server>> show port 16 network

- Current Characteristics for Port 16 -- Network Group

-------------------

Protocol:

 Authorized: lat_compatible, slip, tcpip

 Default: tcpip (raw)

 SLIP Compression: disabled

Dedicated Service:                      Characteristics:

  TCP Node Name:        Sybase_SS2     Autoconnect:     disabled

  IP  Address:            129.144.ddd.ddd

  TCP Port Name:              tpca-xact

  TCP Port Number:                 1599

----------------------------------------- DD-MMM-YYYY HH:MM:SS -
```

- To remove dedicated port setting:

```
Server>>change port port dedicated none

Server>>change port port protocol default none
```

- Program the type of flow control that the terminals will use. In this case, use xon/xoff.

```
Server>> change port all flow control xon
```

- Program all ports to disable autobaud.

```
Server>> change port all autobaud disable
```

- Program all terminal ports to use raw type sockets. This is essential for the type of connection dealt with in this paper. Each port must be programmed separately; range values are not accepted. Port 16 is used in this example.

```
Server>> change port all protocol default tcp raw
```

- Program all ports to request service from IP port 1599 on host Sybase_SS2. The port number came from the entry for slave_applc() in /etc/services.

```
Server>> change port 1-32 dedicated tcpip 129.145.ddd.ddd port
1599
```

- Program all ports for host name and host port name. This is optional and can be used to provide information to the user or administrator.

```
Server>> change port 1-32 dedicated host Sybase_SS2 port tpca-
xact
```

- Logout to reset all ports to the newly programmed state. R

```
Server>>logout port all (port reset)
```

This excerpt is used by permission of Emulex Corporation.

```
PERFORMANCE SERIES SERVERS

         TPC-A BENCHMARK APPLICATION

Sample command session on Performance Server:

- Performance XX00 TCP/LAT-Compatible Terminal Server -----------

    Copyright 1988, 1989, 1990 EMULEX Corp.  All Rights Reserved.

                Use subject to Software Agreement.

------------------------------------- DD-MMM-YYYY HH:MM:SS -

Welcome to Performance 4000
```

- An administrator must first log on to the terminal concentrator.

```
Enter username, or HELP> {username}

Server> su

Password> password: system (default)
```

- Program the name of the server, in this case Sybase_SS2, and its IP address.

```
Server>> change server name  Sybase_SS2 ip 129.145.ddd.ddd
```

- Program the baud rates for the terminals connected to this terminal concentrator. All 32 ports will be used; all terminals will run at 9600 baud.

```
Server>> change port all speed 9600
```

Emulex P4000 Terminal Server

Configuration

C

A terminal server provides hardwire connections for a relatively large number of dumb terminals, an Ethernet port, and a mechanism for opening a socket on behalf of a terminal. Each terminal/socket pair can be programmed with a host name and an IP port address on the host. When the terminal is activated, the Emulex P4000 connects with a socket on the target host bound to the programmed IP address, using the services of `inetd`(). `inetd`() of course passes the connection on to the service available at the requested IP port. Thus, a terminal connection is established in much the same way as the `rlogin`() example presented above. The terminal concentrator is responsible for character buffering and seeing that characters are channeled to their originating terminals.

Below is a commented excerpt from an Emulex P4000 terminal concentrator setup session. The setup is presented as though the terminal concentrator were being configured for use with the SPARCstation 2 front end used in the TPC-A test.

sys/types.h

Code Example B-2

```
typedef short   dev_t;
```

`sys/sysmacros.h`

Code Example B-3

```
/* major part of a device */
#define major(x)        ((int)(((unsigned)(x)>>8)&0377))
/* minor part of a device */
#define minor(x)        ((int)((x)&0377))
/* make a device number */
#define makedev(x,y)    ((dev_t)(((x)<<8) | (y)))
```

`os/tty_pty.c`

Code Example B-4

```
ptcopen(dev, flag)
        dev_t dev;
        int flag;
{
        register struct pty *pty;
        register queue_t *q;

        if (minor(dev) >= npty)
                return (ENXIO);
```

Low-level Excerpts

Included below are excerpts from two headers, `sys/types.h` and `sys/sysmacros.h`, that define the data type of the major/minor device number pair and the way the major and minor number are extracted for use in device access. Note that `dev_t` is defined as a 16-bit value and that both the `major`() and `minor`() macros extract 8-bit values. This eight bit size limits the number of devices to 256.

Also included is `ptcopen`() an excerpt from `tty_pty.c`. `ptcopen`() is the part of the device driver that opens a pty. It uses the macro `minor`() to extract the minor device number from the major/minor device pair contained in `dev` of type `dev_t`. `ptcopen`() checks to see if the extracted minor number is greater or equal to npty. npty is a tunable parameter that can be adjusted before a UNIX make. Its default value is 48. Due to the size restriction on the minor number, it can never exceed 256.

sufficient for also finding a tty. Code is present to make sure the associated tty is there.) The for structure arguments limit the number of trys to 256, the maximum number of pty/tty devices allowed.

Code Example B-1

```
        for (cp = “pqrstuvwxyzPQRST”; *cp; cp++) {
                 struct stat stb;
                 int pgrp;
                  /* make sure this “bank” of ptys exists */
                 line = “/dev/ptyXX”;
                 line[strlen(“/dev/pty”)] = *cp;
                 line[strlen(“/dev/ptyp”)] = ‘0’;
                 if (stat(line, &stb) < 0)
                         break;
                 for (i = 0; i < 16; i++) {
                         line[strlen(“/dev/ptyp”)] =
“0123456789abcdef”[i];
                         line[strlen(“/dev/”)] = ‘p’;
                                                    */
                    if ((p = open(line, O_RDWR | O_NOCTTY)) == -1)
                                 continue;
                                                     */
                         line[strlen(“/dev/”)] = ‘t’;
                         if (chmod (line, 0600) == -1) {
                                  (void) close (p);
                                 continue;
                         }
                        if ((ioctl(p, TIOCGPGRP, &pgrp) == -1) &&
                              (errno == EIO))
                                 goto gotpty;
                         else {
                                  (void) chmod(line, 0666);
                                  (void) close(p);
                         }
                 }
        }
        fatal(f, “Out of ptys”);
```

UNIX System Device Number Limitation B

The UNIX operating system limits the maximum number of pty/tty pairs at both a relatively high level and at a low level. At a high level, `in.rlogind()` is aware that there is an upper limit of 256 pty/tty devices and does not attempt to locate more than that number.

At a low level, a major/minor device number is contained in a single quantity of integer type. The upper 8 bits contains the major number; the lower 8 bits contains the minor number. By data type definition and by assignment convention, the minor number is limited to 256.

Excerpts of UNIX code and header information is provided below to show where these limitations exist.

`in.rlogind.c` *Excerpt*

The portion of `in.rlogind.c` excerpted here deals with locating an available pty/tty pair for use by the `rlogin` client. Within nested for loops, a search is made for the first available pty. (Each pty has a tty slave associated with it so finding a pty is usually

```
    while(dbresults(DB) != NO_MORE_RESULTS)
     while (dbnextrow(DB) != NO_MORE_ROWS)
    ;


    /* Did we deadlock? */
    if (*((DBBOOL *) dbgetuserdata(DB)) == TRUE)
     goto deadlock;
}
```

`stpl_1`() has returned indicating that the transaction completed successfully. The data returned from the transaction is reconverted to a UNIX acceptable format and it is written to the output buffer `out_mesg`. The contents of `out_mesg`, suitable for terminal output, is then written to the socket.

```
        }
    }
    /* close everything and exit */
    dbexit();
    closelog();
    exit(0);
}
```

End of `main()`

```
void
stpl_1()
{
    struct successdata sdata;
    DBFLT8 balance;
    RETCODE sql_stat;


    deadlock:

    sprintf (control.command, “stpl_%d”, (data21.teller %
HISTORY) +1);

    dbrpcinit(DB, control.command, 0);
    dbrpcparam(DB, NULL, 0, SYBINT4, -1, -1, &(data21.account));
    dbrpcparam(DB, NULL, 0, SYBINT4, -1, -1, &(data21.teller));
    dbrpcparam(DB, NULL, 0, SYBINT4, -1, -1, &(data21.branch));
    dbrpcparam(DB, NULL, 0, SYBFLT8, -1, -1, &(data21.delta));
    dbrpcparam(DB, NULL, 0, SYBINT4, -1, -1, &(data21.location));
    dbrpcparam(DB, NULL, (BYTE)DBRPCRETURN, SYBFLT8, -1, -1,
&balance);
    dbrpcsend(DB);
    dbsqlok(DB);


    /* do this whether data is returned or not */
```

```
            syslog(LOG_ERR, “Socket read - %m”);
        } else {
            syslog(LOG_ERR, “Socket incomplete read”);
        }
        break;
    }
    sscanf (in_mesg, “%d %d %d %d %d”,
             &data21.account, &data21.teller, &data21.branch,
             &my_delta, &data21.location);
    /*
     * do conversions and submit transaction
     */
    dbconvert((DBPROCESS *) NULL, SYBINT4, &my_delta, -1,
             SYBFLT8, &data21.delta, -1);
```

The input data for a transaction is read from the socket and loaded into the buffer `in_mesg`. The contents of `in_mesg` are then moved into the relevant fields of `data21`. `data21` contents are next converted to a format that Sybase is sure to understand. `stpl_1`() is called to submit the data and initiate the transaction.

```
    stpl_1();
    dbconvert((DBPROCESS *) NULL, SYBFLT8, &data21.delta, -1,
             SYBINT4, &my_delta, -1);
    /*
     * put output
     */
    bzero(out_mesg, sizeof(out_mesg));
    sprintf (out_mesg, “%d %d %d %d %d”,
             data21.account, data21.teller, data21.branch,
             my_delta, data21.location);
    blank_mesg(out_mesg, sizeof(out_mesg));

    err = write(1, (char *)out_mesg, OUTPUT_SZ);
    if (err < OUTPUT_SZ) {
        if (err < 0) {
            syslog(LOG_ERR, “Socket write - %m”);
        } else {
            syslog(LOG_ERR, “Socket incomplete write”);
        }
        break;
```

Before data transmission can begin, error handling precautions must be taken.

```
    bzero(db_server, sizeof(db_server));
    err = read(0, (char *)db_server, sizeof(db_server) - 1);
    if (err < (sizeof(db_server) - 1)) {
    if (err <= 0) {
        syslog(LOG_ERR, “Socket read db_server - %m”);
    } else {
        syslog(LOG_ERR, “Socket incomplete read db_server “);
    }
    closelog();
    exit (1);
    }
    if (putenv(strdup(db_server)) != 0) {
    syslog(LOG_ERR, “putenv failed”);
    }
    /*init()*/
    control.program = “Slave”;
    dberrhandle(err_handler);/* install error handler */
    dbmsghandle(msg_handler);/* install message handler */


    /*get_ready()*/
db_connect() opens a connection with the Sybase backend engine.
Once a connceciton is successfully established, the data exchange
can begin.

    db_connect(DB, USERID, PASSWD, SERVER, APPLICATION);
    if (dbuse(DB, DATABASE) != SUCCEED) {
    syslog(LOG_ERR, “dbuse did not return SUCCEED”);
    }

    /*
     * Run transactions
     */
    for ( ;; ) {
    /*
     * get input
     */
    bzero(in_mesg, sizeof(in_mesg));
    err = read(0, (char *)in_mesg, INPUT_SZ);
    if (err < INPUT_SZ) {
        if (err <= 0) {
```

```
#include <sys/types.h>
#include <syslog.h>
#include “sybfront.h”
#include “sybdb.h”
#include “tp_rte.h”

#define RESPONSE_TIME930000

CONTROLcontrol;
XACT21data21;
voiddb_connect();


main()
{
    chardb_server[2 * OBJ_NAME + 1];
    charin_mesg[INPUT_SZ + 1];
    charout_mesg[OUTPUT_SZ + 1];
    interr;
    inti;
    DBINT my_delta;


    openlog(“slave_appl”, LOG_PID | LOG_CONS, LOG_DAEMON);

    /*
     * get server name to put in environment
     */
```

Slave_applc() A

The code segment presented below is a piece of the TPC-A benchmark code. It is the front end process that takes input data, in a well defined format, packages it for use by the backend Sybase database engine, and sends it to the backend.

It then receives results from the backend, unpackages them, and sends them to the originating terminal.

This is the service routine, analogous to `in.rlogind`(), that is exec’d by `inetd`(). There is one of these processes for each connected terminal.

```
#ifndef lint
staticchar sccsid[] = “%Z%%M% %I% %E% SMI”;
#endif

/*
 * Master/Slave Driver software
 * Written by Hal Spitz at Sybase and Jim Skeen at Sun
Microsystems October, 1990
 *  modified Ravi Chhabria at Sun Microsystems March, 1992
 * NOTE: PROPRIETARY SOFTWARE OF SYBASE AND SUN MICROSYSTEMS
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
```

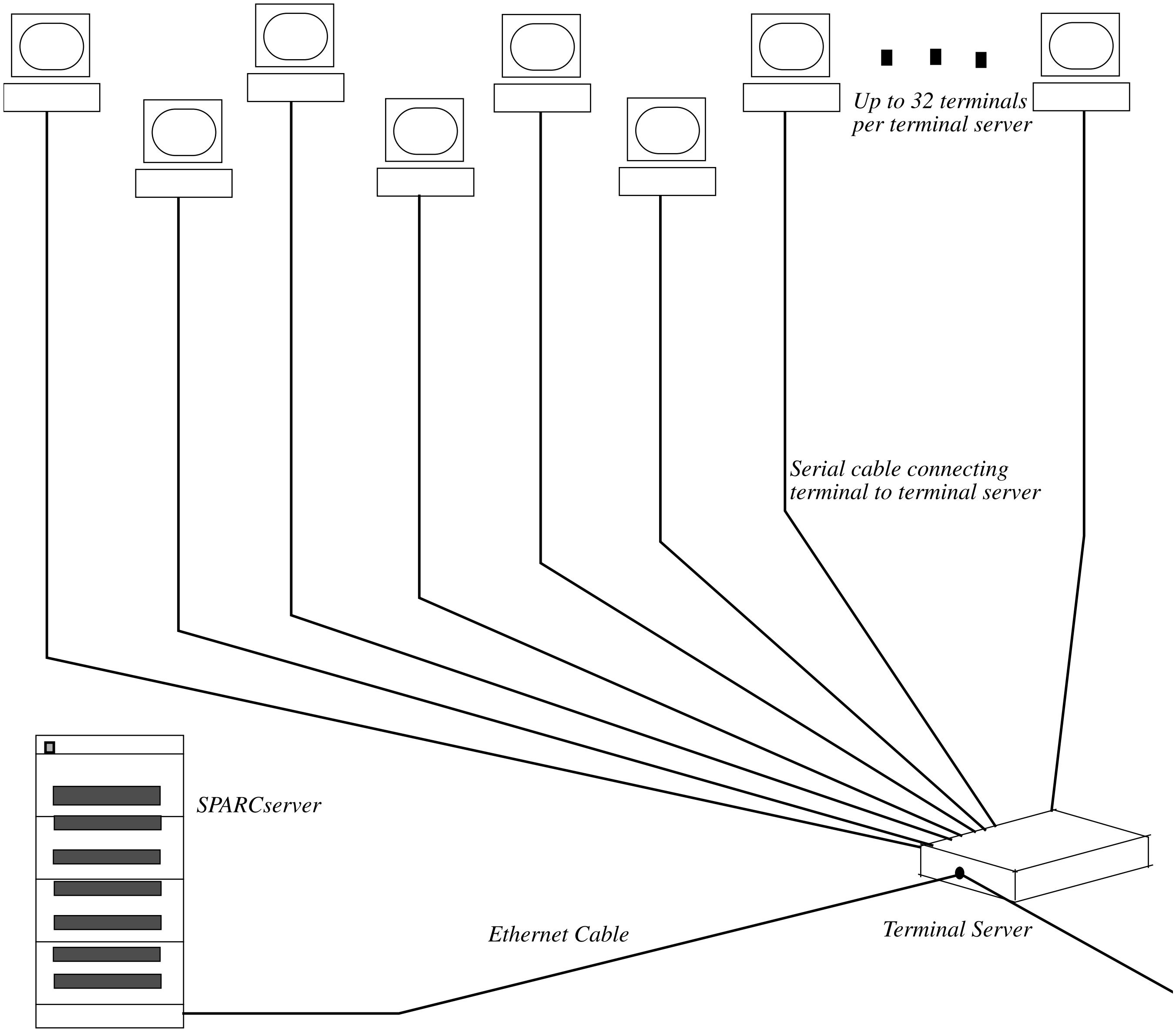


Figure 3-1 Generic configuration showing SPARCserver, terminal server, and terminals connected to the terminal server.

Multiple references to a given device driver are established by making a second (or further) entry in `cdevsw`(). The position that an entry occupies in `cdevsw`() determines the major number through which the device driver will be referred. To enable this modification, a UNIX `make` must be performed.

In addition, a corresponding list of pty and tty entries must be entered in /dev. Each entry must refer to the new `cdevsw`() major number and minor numbers must not be duplicated within the scope of the major number.

Using this technique, it should be possible to create additional sets of 256 pty/tty device pairs. Further work needs to be done to prove that this proposal works.

This simplicity is a consequence of the limited nature of the application. The TPC-A benchmark executes a small repetoire of activities. All input and output is well behaved. Output formatting is not required. Nevertheless, connectivity has been established. The approach presented here is a starting point for anyone wanting to make similar connections. It is up to the application to use the connection correctly.

As it enters an active state, `slave_applc`() has received from `inetd`() a reference to the connecting socket in descriptor 0. After performing some environment setup and connecting to the database server, it enters a loop. In the loop it reads from `stdin` (fd0) `INPUT_SZ` bytes and loads them in a buffer.The routine blocks on `read`(). The buffered input is parsed into Sybase specific structures. The structure contents are converted to a form presentable to the database, and the structure is submitted as a transaction.

When the transaction completes, the returned data is reconverted and copied from the structure format into a character buffer. The contents of the buffer are then written to `stdout` (fd1). This sequence of read input, submit transaction, receive results, write output continues indefinitely. See Appendix A for details on `slave_applc`()

Loose Ends

The solution proposed here answers the question of connectivity. It still leaves some questions of utility unanswered. When a user is entering data at a terminal errors are bound to occur. Errors are anticipated and provided for in the pty/tty situation with canonical processing. This allows for such things as back-space-delete and line input rather than character input.

In those cases where it is necessary to do character input processing such as that provided by canonical processing, one possible alternative would be to duplicate the device pathways to the pty/tty driver.

In one sense, the driver is a template. The driver itself is unaware of the devices on which it operates. The device specific information is passed to the driver when the device is opened. The driver imposes a 256 device limitation - the minor device number (See Appendix B for more detail), but as a consequence of the template-like design, it makes no assumptions about the major device number. Therefore, a driver can be referred to through more than one major number. Each major number then has a separate set of 256 minor numbers associated with it.

This is the same IP address that is programmed in the terminal concentrator port. The Sybase application entries for the inetd.conf and services tables are shown below in Tables 4-4 and 4-5.

`inetd`() listens for a request as in the native UNIX case. When it hears one, it extracts information such as socket characteristics, type of connection, host name and address. `inetd`() accepts the socket connect request and exec's `slave_applc`. Once running, `slave_applc`() is responsible for all terminal I/O, independent of pty/tty mechanisms.

Table 3-1 Sybase application inetd.conf entry

Entry in inetd.conf used by inetd() to start the Sybase front end process slave_applc. (See Table 1)

```
# @(#)inetd.conf 1.23 90/01/03 SMI
#
# Configuration file for inetd(8). See inetd.conf(5).
#
# To re-configure the running inetd process, edit this file, then
# send the inetd process a SIGHUP.
#
#
# Internet services syntax:
# <service_name> <socket_type> <proto> <flags> <user> <server_paththname> <args>

tpca-xact stream tcp nowait dbbench /export/db/dbbench/vendors/sybase/core_queries/c/slave_appl slave_applc
#
```

Table 3-2 Sybase application services entry

Entry in services used by inetd() to extract the IP socket address for slave_applc.See Table 2)

```
tpca-xact 1599/tcp # DBE tpc benchmarks
```

Sybase Front End Process

slave_applc(), the Sybase front end process, is really very simple. It's job is to accept input data in a pre-determined format, package the data for the back end database engine, and dispatch the data. It then waits for results from the database engine, again in a pre-determined format. It repackages the received data and passes it along to the source. It repeats this indefinitely.

Connecting to a Sybase Application 3

Sybase connectivity

The mechanism for establishing a terminal connection to a Sybase application is very similar to that used in the rlogin sequence. There are elements analogous to `rlogin`() and `in.rlogind().` `inetd()` is used in exactly the same way as in the UNIX example. Instead of using `rlogin`(), the Sybase application depends on the use of Emulex P4000 32 port terminal concentrators. Each port in each terminal concentrator can be programmed to connect to an IP port on a given front end server. Each port can also be set up for terminal characteristics. The configuration of the Emulex terminal concentrator is detailed in Appendix B.

Code for a process was created and installed on the front end ELC and SPARCstation 2 system. This process was used to send terminal data to and received terminal data from the terminals; and to send transaction requests to and receive transaction results from the database engine. This process, called `slave_applc`(), provided a link in the connection sequence similar to that provided by `in.rlogind()`.

To open a connection to `slave_applc`(), a user at a terminal connected to a terminal concentrator might enter a carriage return. This would alert the terminal concentrator to connect to a socket bound to a specific address on one of the front end systems. The socket address, associated with `slave_applc`(), must be entered in `inetd.conf` and in services prior to use and must follow the same format used by other IP services. The address should be greater than 1024 and must not conflict with any other address; addresses in the range 1 to 1024 are reserved for privileged processes such as root.

the argument to `execlp`(). Ordinarily, the process is a shell, thus giving the user access to system resources and applications. However, `login`() imposes no restrictions on what process must be executed here. It can be any procedure to which the user has legitimate access. Once operating in the context of the shell or other process, the user has finally established connection.

`getty`() first opens the tty device for which it is responsible and dups `stdin`, `stdout`, and `stderr` onto it. It then moves into a loop in which it sets up terminal characteristics and calls `getname`(). `getname`() reads from the tty. If there is nothing there it returns to the `getty`() loop. Otherwise, it reads the terminal input. When it encounters a NL character, it returns to the `getty`() loop having saved the input in a buffer.

If on return, `getname`() indicates no activity, the loop continues to turn. If getname() returns indicating activity, `getty`() completes the environment setup for the device and then makes the following exec call:

```
execle(LO, "login:", "-p", name, (char*)0, env)
```

`login`() is the process to be exec'd with the -p flag set. Name is the name of the user requesting a connection. Env provides reference to environment information to be used by `login`().

Login and Beyond

Whether through a login or `rlogin` request, the user has arrived at a common point within the system, namely at the `login`() process. All activity leading up to this point has been preparatory to making a connection to the system. In `login`() the connection is finally established.

rlogin() case - `in.rlogind`() has exec'd `login` with the -r flag set and passed the clients name as a parameter. `login`() first looks in the file /`etc/hosts.equiv` or ~.`rhosts` to see if the client is allowed access. If access is allowed, then the terminal type is propagated over the network. `login`() must know who the user is. `login`() either uses the clients name or an alternate name specified by the client in the `rlogin` command line.

login() case - `getty`() has exec'd `login`() with the p flag set. The p flag indicates that the environment (set up by `getty`()) should be preserved. `login`() writes the login: prompt to the tty and waits for the user to enter a name.

With a user name in hand and machine access verified, `login` and `rlogin` follow much the same path. The /`etc/passwd` file is searched for an entry corresponding to the login name. If a password is indicated in the entry, `login`() prompts for it.

If all goes well, and the `login`() process has found no irregularities, it performs an `execlp`() call. The exec'd process is specified in the final field of the requestor's entry in /`etc/passwd`. `login` indexes through the passwd file until it finds an entry. that matches the login name. It extracts the path name of the entry and provides it as

`init()`

`init()` is a very involved process responsible for setting up a user session. Once the session is set up a user has access to such system resources as system services, peripheral devices, and applications. As part of it's peripheral initialization services, `init()` sets up tty devices.

The tty setup is managed by a routine called `merge()`, called from within the body of `init()` at boot time. It is also invoked by a signal handler when a connection to a tty ends. Thus, when a user ends a work session, say from a shell, the hang-up signal sent makes it's way to `init()` and a new tty setup is started.

`merge()` refers to the file `/etc/ttytab` to determine active tty candidates and to extract information used when they are made active. The format of ttytab is shown in Table 3.

Table 2-3 ttytab Format

Field Name	**Field Description**	**Example**
name	The terminals entry in the device directory	console
getty	The command to be executed by the line (typically getty)	/usr/etc/getty
terminal	The type of terminal expected to be connected to the terminal device, as found in `/etc/termcap`	sun
status_fields	Deals with security issues, whether to run command in getty field, whether to ignore modem signals	on local secure

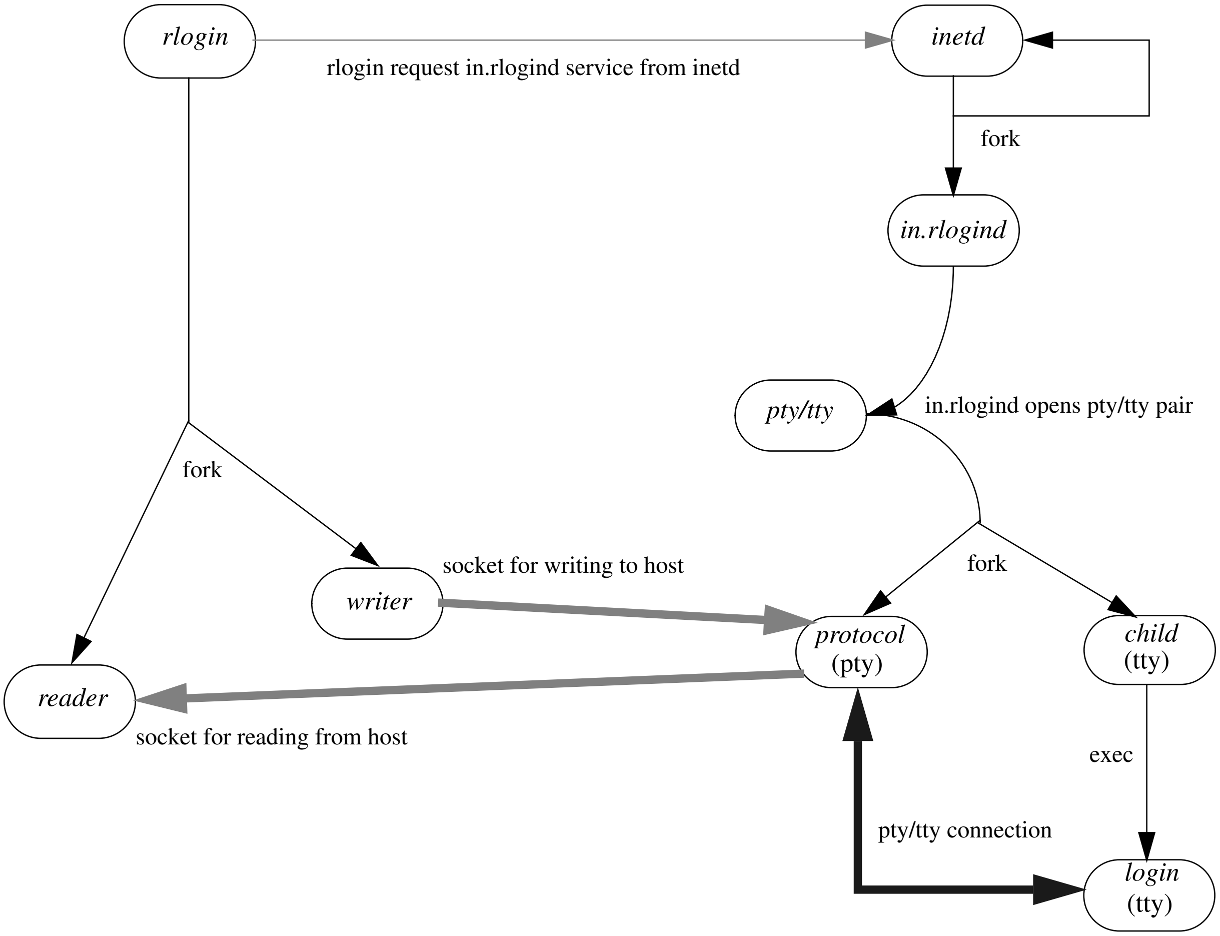
In ttytab, there is one entry per terminal device. `merge()` reads down through each entry. Configuration information is extracted from `ttytab` per active tty. For each active entry, `merge()` first kills any existing processes running on the tty and then forks a process. The child process execs the process indicated in the getty field of ttytab and associates it with the device of the current line. Thus, when the merge section of `init()` completes, a `getty()` process will be running for each active terminal.

getty()

A getty() process runs for each idle tty device. It cycles through a loop and looks at it's tty device for input activity. If there is activity on a line, it reads the input and execs `login()`.

At this point, assuming no error conditions have been encountered, the connection is established.

Figure 2-1 Diagram of `rlogin` session



Establishing Connectivity Using the getty- login Sequence

Users working at terminals and connecting to a local host follow a different path from that used by `rlogin` although both eventually end up at `login`. The gross pathway to `login` for local users is `getty`() to login(). Before these steps can be taken, `getty`() processes must be spawned, one per active tty. It is the responsibility of the `init`() process to do this.

`inetd`() has set the stage for the `rlogin` daemon in.`rlogind`() by accepting a request from a client and resolving environment inconsistencies. When `inetd`() execs `in.rlogind`() it passes a reference to the established socket connection in descriptor 0.

Since `in.rlogind`() will eventually exec a call to `login`() which requires the name of the requesting user, it fetches this name from the client in a call to `gethostbyaddr`(). It then checks to see that the connecting socket is authorized to request a login.

The next task `in.rlogind`() undertakes is to find a pty/tty pair that is available for use. The list of ptys (in /`dev`) is visited one by one. The first available pty is opened and the slave tty associated with it is locked to prevent others from using it. In a similar operation, the tty is then opened and terminal environments are set up.

Having secured a pty/tty pair, `in.rlogind`() forks a child process. The parent process calls `protocol`() where control remains until the session ends. `protocol`() moves through a loop looking for activity on the socket and activity on the pty. Using `select`(), it determines where the activity is and also which of the descriptors, `stdin` or `stdout`, can be written to. If input is detected on the socket it is read into a buffer. If the pty can be written to, the contents of the buffer are written to it. Conversely, if input is detected on the pty, it is read into a second buffer. If the socket can be written on, the second buffer contents are sent to it.

In addition `protocol`() constantly monitors for a signal from the client to make changes in the window attributes. Any indicated window modifications are made with an appropriate `ioctl`() call.

The child process closes the socket and pty descriptors and dups the tty descriptor on `stdin`, `stdout`, and `stderr`. It then closes the tty descriptor and performs the following command:

```
execl("/bin/login", "login", hostname, 0).
```

inetd() Main Loop

Once inetd() has set up it's services table, it enters a loop out of which it never breaks. At the top of the loop a select() call is performed. select() is passed a list of sockets references corresponding to the active services. When it returns is has flagged those sockets with requests pending.

In an inner loop each socket reference is visited and if any activity is indicated it is performed. If the requestor has asked for a STREAM socket as `rlogin`() does, an accept() is made to the socket. The effect of accept() is to open a second socket and connect it to the first requestor on the listen queue. This new socket, with a new file descriptor, inherits the properties of the original socket. Providing a second socket to the requestor establishes the service related communications without interfering with other requestors for the same service. The original socket continues to listen for incoming requests.

Having set up a private communications pathway to the client, inetd() (for STREAM socket connections like `rlogin`()) forks a child process. The child closes all open file descriptors except for the socket descriptor, stdin, stdout, and stderr, setting itself up as session leader. It then dups the socket descriptor on stdin, stdout, and stderr.

The effective group ID and the user ID are set to that of the user as specified in inetd.conf. Similarly, supplementary group IDs are set. With the child environment firmly in the hands of the user, an execv() call is made to the service routine indicated in inetd.conf. For `rlogin`, the service routine is in.`rlogind`().

The parent process closes the socket descriptor opened by the accept() call and returns to the top of the main loop. The client is connected to the service routine, in.`rlogind`(), and inetd() is once more listening for activity on the service ports.

In.rlogind()

When `inetd`() execs `in.rlogind`(), it passes a reference to the connecting socket. in.`rlogind`() opens a pty/tty pair and then forks a copy of itself. The child of this fork inherits the tty from the pty/tty pair. Finally the child execs `login`() itself, passing along references to the tty in `stdin, stdout, and stderr.` The parent maintains communication with the client `rlogin`() process through the socket and communication with `login`() (which has been exec'd by the child process) through the pty/tty pair.

a queue where incoming requests can be buffered and then triggers the socket to queue requests as they come into the associated port number. Once the table of queues is established and the sockets set up, `inetd`() then moves through it looking continuously for sockets with pending requests.

Table 2-1 The format of /etc/inetd.conf is:

Field Name	**Field Description**	**Example**
service_name	name of valid service listed in the file etc/services.	login
socket_type	stream for a stream socket	stream
	dgram for a datagram socket	N/A
	raw for a raw socket	N/A
	rdm for a “reliably delivered socket”	N/A
	seqpacket for a sequenced packet socket	N/A
protocol	a recognized protocol listed in /etc/protocol e.g., udp, tcp, icmp.	tcp
flags	wait for a “single threaded” socket which does not allow other requestors access to the service	N/A
	nowait for a “multi threaded” socket which allow multiple instances of the service to coexist	nowait
user	user id under which the server should run	root
server_pathname	pathname of the service to be run by inetd	/usr/etc/in.rlogind
args	server command line arguments	in.rlogind

Table 2-2 The format of /etc/services is:

Field Name	**Field Description**	**Example**
service_name	official Internet service name	login
port/protocol	port and protocol through which a service is provided	513/tcp
aliases	alternate name(a) for a service	N/A

`inetd`(): Servicing Internet Requests

`inetd`() manages requests for services in the Internet domain. `inetd`() runs on every machine that is configured to accept IP service requests. The service requests to which it will respond are defined externally. Knowledge of them is imported when `inetd`() initializes; it is possible to dynamically reconfigure this list of services.

`inetd`() operates in two phases.

1. In the initialization phase it runs through a list of offered services and enables them one by one.
2. After the initialization phase, it sits on a server and listens for requests for services from remote clients. When it discovers a request, it accepts it and passes the connection to the responsible server routine. Having accomplished this, inetd() returns to listening for requests from Internet clients.

inetd() Initialization

`inetd`() is invoked by `init`() as it runs through `/etc/rc.` `inetd`() expects `portmap`() to be running before it starts. Signal handling is set up and then an internal routine, `config`(), is called.

`config`() builds an internal table of services data based on information it extracts from two files, `/etc/inetd.conf` and `/etc/services.` `inetd.conf` provides IP protocol information and service routine path information; `services` provides the IP port numbers. (See tables 2-1 and 2-2 for details.) As each service type is encountered in `inetd.conf`, the corresponding entry in `services` is located and the port number of the service is retrieved. Port numbers are assigned by convention; a given service is requested at the same port number on all machines under a common network administration. In a Network Information Service (NIS) environment, services resides on the NIS server; in a non-NIS environment services resides on the local machine.

The initialization phase is run only once at boot time.However, it is possible to add or delete services during the run phase. Appropriate additions or deletions are made to both `inetd.conf` and services Sending a hang-up signal (`SIGHUP`) to inetd() will cause it to adjust the service table accordingly.

For each entry in the internal services table, a socket is opened and bound to the port number extracted from services. For STREAM type sockets, such as the one to which `rlogin`() will connect, a listen() call is made on behalf of the socket. listen() creates to the service. Once `rlogin`() is connected to `login`(), `inetd`() goes back to looking for connection requests and `login`() takes over the task of opening a shell for `rlogin`().

`rlogin()`

The shell parses the `rlogin` command line and passes parameters along to `rlogin`(), which extracts the host name and the user login if specified. It also sets flags according to command line options and sets up the environment for terminal characteristics, window size and signal catching.

To connect with the server, `rlogin`() calls `rcmd`(), which takes as arguments the host name, the port to which the client is to connect on the server, the local user's name and the remote user's name. The port on the server in this case is `IPPORT_LOGINSERVER`. It is defined in /`usr/include/netinet/in.h` and is one of 1024 well defined Internet addresses reserved for privileged processes such as root. `IPPORT_LOGINSERVER` calls for a TCP or STREAM type socket.

`rcmd`() retrieves particulars about the host with a call to `gethostbyname`(). It then opens a socket and binds it to an address in the range reserved for privileged processes. Next, rcmd() opens a connection with the remote port, `IPPORT_LOGINSERVER`. `rcmd`() passes arguments to the server and then returns to `rlogin`() with a socket descriptor.

After `rcmd`() returns, `rlogin`() calls `doit`(), from which it never returns. In `doit`(), full duplex communication is set up by forking a child process. The child calls reader() to read from the socket into a buffer and write from the buffer to stdout. `reader`() uses an 8K byte buffer and reads and writes blocks up to the size of the buffer. `reader`() continues to read and write until the socket connection is broken.

The parent process calls `writer`() to read from `stdin` and write to the socket. `writer`() reads and writes in character increments. `writer`() tracks beginnings of lines so that it can catch command characters, but leaves canonical processing to the host. `writer`() continues to read and write until an end-of-file is reached or an error condition is encountered.

Native UNIX Connectivity

2

To establish a connection with a UNIX host supporting multiple users at so called dumb terminals, a user will typically enter a user name and password at a login prompt. Similarly, a user working at the shell level, whether on a dumb terminal or at a workstation, may establish a second (or further) shell connection through use of the `rlogin`() command. To the user these two procedures are identical in intent and closely related in execution. However, the underlying mechanisms are quite different.

At a physical level, a dumb terminal is connected to a UART by a serial line, commonly following RS-232 specifications. The UART plugs into the bus of the host machine and mediates character reception and transmission between the host terminal handler and the terminal itself. An `rlogin`() connection is established by means of sockets and server resident daemons. If the session is to take place on a server physically removed from the client, ethernet controllers, ethernet transmission media, and TCP/IP are involved as well.

Establishing Remote Connections Using `rlogin`

At a shell prompt, a user enters `rlogin` and specifies a host. If the session is to run under a user name different from that of the current session, that *username* is provided as an optional parameter. Other optional parameters can be specified but do not bear on this discussion.

`rlogin`() establishes a connection with a shell on the specified host. It does this by requesting a `login`() process, on the host, from the IP daemon `inetd`(). `inetd`() monitors activity on a set of IP ports active on the host. When a request for a connection to a host service such as `login`() is made, `inetd`() sets up the connection

The need for the type of character filtering provided by canonical processing still exists in some applications. This is true in those cases where a user enters data from a keyboard or pad and can potentially make errors requiring deletions, substitutions, etc. For those applications where data input is automated or modular such as UPC tagged items and barcode readers, the need is not so critical.

The rationale for configuring an absolute minimum cost system in this example was to contribute to the favorable results of a benchmark. Because of the nature of the test, canonical processing was not an issue. Input data was generated automatically and accuracy was guaranteed. For applications that fit the model, this approach is worth considering.

There is an alternative approach that works within the bounds of the UNIX device constraint. Using the Sun™ client/server configuration of a backend database engine with multiple front end systems acting as connectivity mediators makes it easy to configure large numbers (greater than 256) of users. One front end is needed plus one front end for each additional group of 256 users. Especially for large systems, adding front end connectivity servers only increases the cost per user a small amount. The decision that must be made is whether or not the expense of additional front end systems justifies having to add some form of canonical processing which is otherwise available only through use of pty/tty devices.

Working with the UNIX Environment

It is obvious that because the direct terminal connection setup used in this TPC-A test ran in a UNIX environment, it had to use and conform to the UNIX conventions. Even though the device driver for pty/tty devices was bypassed, a great deal of the native UNIX pathway for establishing a user connection was used. The next chapter will look closely at this pathway and lead to a direct understanding of the method used in the TPC-A test.

Each Emulex terminal concentrator supported up to 32 terminals connections over an ethernet network. A terminal concentrator was assigned an IP address; each terminal port was then configured to address an IP address on the network. The Emulex terminal concentrator gathered input from it's terminals and packetized it to be sent to assigned front end hosts. Similarly, it accepted replies form the host and passed individual replies to their designated terminals. Using this mechanism, it was possible to bypass the pty/tty intermediary.

Transaction Processing in Brief

If improved UNIX connectivity is gained, then it is appropriate to ask at what cost. Two things are lost by bypassing the pty/tty mechanism: access to UNIX system resources (as provided by a shell process) and canonical character processing. To address the first point, it is worthwhile to look at the way transaction processing systems are used.

Transaction processing systems find great utility in the commercial world. Typical applications include ATMs, point-of-sale terminals, inventory control systems, order entry systems, etc. While there is no functional reason that these systems could not be configured to serve only a limited number of users, they frequently serve large numbers in a high use environment. They benefit from economies of scale and may become commercially expedient only when serving large numbers of users. ATM networks are a good example of this. Hundreds of ATMs spread over a metropolitan area are used by thousands of banking customers. The cost of building the network is justified only if there is a large customer base who will use the facility. Naturally, the lower the cost of the system, the greater the return on investment.

Two transaction processing system requirements—cost minimization and throughput rate—are facilitated by bypassing both the pty/tty mechanism and the shell. Cost is minimized by reducing the number of machines needed to support large numbers of terminals. Since code paths are shortened, system overhead is reduced and throughput is promoted. A third consideration is the fact that these systems experience high use in a limited context. For instance, at a point-of-sale terminal, hundred of items may pass through in an hour but each item is processed in the same way. There is no variation in use, so access to secondary system resources is not really an issue. On the contrary, by reducing the options of operation, reliability and security is enhanced.

Transaction Processing Issues

1

The TPC-A Test System

Sun Microsystems Database Engineering reported a TPC-A Benchmark result in February 1992 of 95.41 tpsA at $8972 per tpsA, an industry leading figure. In order to minimize the dollar/tpsA figure, it was essential to configure the least costly platform consistent with the anticipated results. One of the strategies used to achieve this was to configure a system supporting 960 dumb terminals through only two SPARCstation front ends. While effective at reducing the system cost, this seems to violate a UNIX limitation of 256 terminals per kernel.

Specifically, the UNIX limitation is the number of pty/tty pairs that can be supported and not the number of terminals. If terminal communications can be established without the use of a pty/tty device then the restriction is defeated. This is what the Database Engineering team responsible for TPC configuration did.

The TPC test platform included a SPARCserver™ 690MP system, a SPARCstation™ ELC, a SPARCstation 2, and 30 Emulex® P4000 terminal concentrators. The 690MP was host to the Sybase database server. The SPARCstation 2 and the ELC were hosts to Sybase front end processes.

A Sybase front end process was created for each connected user. It’s purpose was to accept input command and data, to render it into a form suitable for use by the database manager, and to send it to the manager. It then waited for the results of the operation to be returned and output them to the connected user. The SPARCstation 2 supported 548 of these processes; the ELC supported 412.

Three appendices are included. Respectively, they are:

- Test system code examples for those interested in creating extended configurations under similar circumstances.
- Extracts from UNIX showing why there is a 256 terminal device limitation.
- Details on configuring the Emulex P4000 32 port Ethernet Terminal Server.

Abstract

This paper provides a starting point for those interested in configuring UNIX systems in a client/server environment with more than the 256 terminal connections potentially available in a standard UNIX system. It is based on work done by SMCC Database Engineering. It is also serves as a reference document describing the UNIX mechanism used to connect a (pseudo) terminal with an interactive process.

In February 1992, SMCC Database Engineering published a report of TPC-A Benchmark performance. The report cited a Sun configuration supporting 960 terminals using 30 Emulex terminal concentrators, two SPARCstation front ends, and a backend 690MP database server.

It is widely understood that current SunOS and other UNIX releases will support no more than 256 terminal sessions per kernel. In order to configure the 960 terminals as described in the TPC-A report, it was necessary to bypass SunOS imposed limitations without introducing systemic problems. The result was a system that supported up to 960 users of a dedicated application, without direct access to operating system facilities.

The paper begins with a brief description of the TPC-A test system and the transaction processing environment. Transaction processing is an area where this type of connectivity may reasonably be proposed.

The method used to establish connections between the terminals and the SPARCstation front ends was very similar to that used to establish a login session in UNIX. Using login() and rlogin() as examples, the mechanism for establishing a (pseudo) terminal connection is traced. Next, following the framework of the UNIX examples, the method used to configure these 960 terminals is presented.

Extended UNIX Connectivity
Contents

© 1992 Sun Microsystems, Inc.—Printed in the United States of America.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation is protected by copyright and distributed under licenses restricting its use, copying, distribution and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc. and the University of California, respectively. Third party font software in this product is protected by copyright and licensed from Sun's Font Suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a trademark and product of the Massachusetts Institute of Technology.

Code examples for©1990, 1991 Sybase, Inc. and Sun Microsystems, Inc.

Please
Recycle

Extended UNIX Connectivity

Technical White Paper

Technical Product Marketing

Sun Microsystems
Computer Corporation
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.