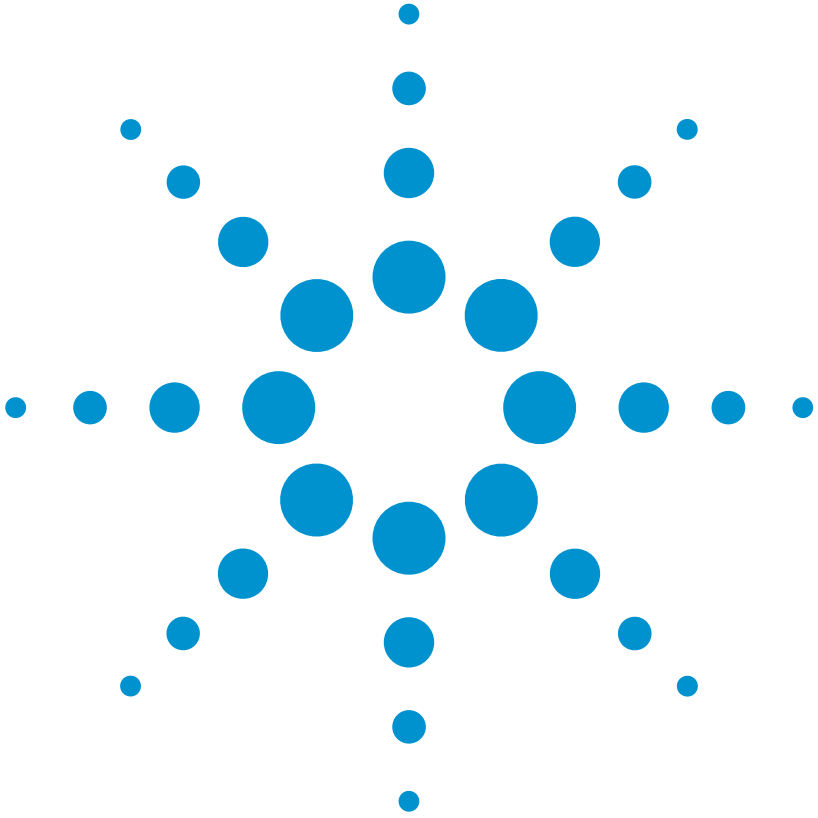
Using Linux to Control USB Instruments

Application Note AN 1465-30



The move to PC standard I/O interfaces is a key element of Agilent Open, which is a versatile combination of hardware, I/O, and software tools that make it easy to create, enhance and maintain systems. You can take advantage of this strategy, especially if you are using Linux as the operating system for your test solution, because support for LAN and USB interfaces is built into the operating system. *Using Linux to Control USB Instruments* is part of a series of application notes designed to explain how to control your test instruments under Linux.

Table of contents

Agilent Technologies

Overview: USBTMC and USB-Based Instruments

With the Agilent Open program, which aims to simplify connectivity to measurement equipment for test system developers, Agilent strongly advocates the move to standard PC interfaces (especially LAN and USB). Many of Agilent's newer instruments support Ethernet, USB and GPIB.

Basic support for USB is built into today's Linux kernels in the form of low-level USB drivers (kernel modules) that control the computer's USB chipset (see Figure 1). However, these drivers do not offer a low-level programming interface to the user (applications running in user space). Instead, they are typically called by other kernel modules (in kernel space) that support certain types of USB devices, such as pointing devices (like mice) or USB disk drives. In most cases, to be able to use a USB device, you need a kernel driver that supports its corresponding device class.

Test and measurement instruments are no exception. A number of leading instrument vendors, among them Agilent, worked with the USB Implementers Forum (USB-IF) to create a vendor-independent standard for USB-based instruments. The resulting USB Test and Measurement Class (USBTMC) specification was published in 2002. Most USB instruments available today adhere to the USBTMC specification—especially those from Agilent.

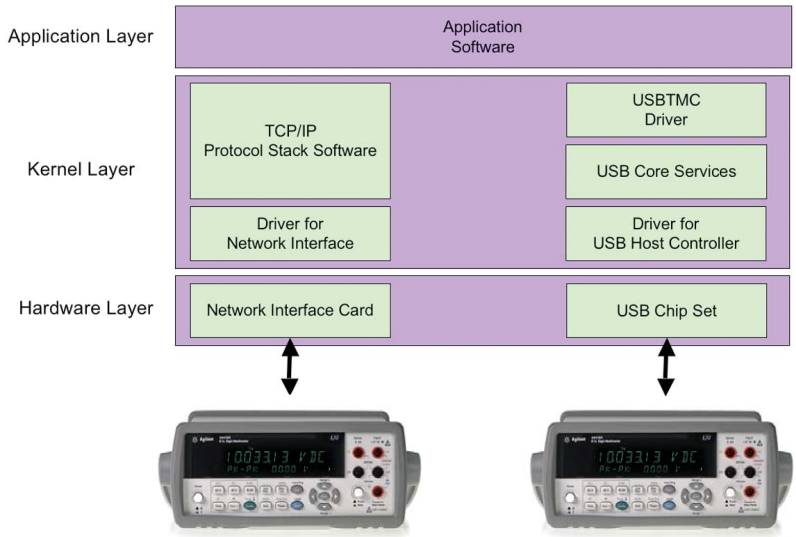
This application note describes how to create a USBTMC kernel module to control USB instruments. Example source code for the driver described below is available from Agilent's Web site at **http://www.agilent.com/find/linux** for compilation on your distribution and kernel versions. It was written and tested under openSUSE 10.2, but it should run nearly unchanged on most current distributions.

Basic USB Terminology

USB has its own way of sharing bandwidth and logically structuring communication on the bus. The most important notion here is that of an *endpoint*. A single physical device typically uses several logical endpoints and each either sends ("in" endpoint) or receives data ("out" endpoint). In addition to their direction, endpoints differ in type.

"Control" endpoints are used for device configuration and initial setup. "Bulk" endpoints are used to transfer larger amounts of data to/from a device. For example, USBTMC devices use "bulk out" endpoints to receive SCPI commands and "bulk in" endpoints to transfer measurement results. "Interrupt" endpoints are used

Figure 1. USBTMC driver structure compared to Ethernet



to accommodate time-sensitive transfers of small amounts of data (for example, the movement of a USB mouse). Similarly, "isochronous" endpoints are used to continuously reserve bandwidth for larger amounts of data (for example, streaming audio/video applications).

An *interface* is a set of endpoints grouped together to offer a certain subfunctionality of the device. For example, USBTMC devices use a control endpoint for setup and bulk in/out endpoints for "regular" communication. These endpoints are all you need to control a USBTMC device, and they are part of a single (logical) USB interface. Some USB devices use several interfaces, such as a USB sound card with separate interfaces for its audio output and input capability.

Although not typically used with USBTMC, an interface can have a number of *alternate settings* (configurations), for example for different physical USB interface speeds or different application types for the same device.

Another important notion about USB is that of a *universal request block* (URB). An URB is a data structure used by a device class driver to instruct the USB core driver to transfer data to/from a USB device. The URB contains the data to be transferred, as well as all the addressing information required to properly process the request.

Communicating with a USBTMC Instrument

Communication with a USBTMC device consists mostly of sending SCPI commands and reading the instrument's responses to query commands. To show you how this works, we'll use an example. We'll send a SCPI command and explore the details of the communication.

For sending instrument commands, the USBTMC standard defines the `DEV_DEP_MSG_OUT` message. This message is sent to the instrument's bulk out endpoint and contains a number of fields in addition to the SCPI command itself (see Table 1).

Table 1. Structure of the USBTMC DEV_DEP_MSG_OUT message (example for *RST command)

Offset (bytes)	Field	Size (bytes)	Value	Description
0	MsgID	1	1	DEV_DEP_MSG_OUT message (used to send a command string to the instrument).
1	bTag	1	x	Transfer identifier. This ID is incremented with every transfer and allows the instrument to detect lost messages.
2	bTagInverse	1	~x	One's complement of bTag (transfer identifier).
3	Reserved	1	0x00	Reserved
4...7	TransferSize	4	5	Number of bytes to be transferred (instrument command).
8	bmTransferAttributes	1	0x01	End of message. If bit 0 is set to 1, the instrument message ends with this transfer. Otherwise, the message continues with the next transfer. All other bits are reserved (set to 0).
9...11	Reserved	3	0x000000	Reserved. Set to 0x000000
12...16	Instrument Command	5	"*RST\n"	Instrument command

VISA IO library for Linux

If you are familiar with the IO libraries suite available from Agilent for MS Windows environments, you might also be interested in VISA programming for Linux. VISA is a multi-vendor standard for instrument control. The VISA/SICL library for Red Hat Linux, available from TAMS **(www.tamsinc.com)** supports most common interface types (GPIB, USB, LXI, VXI, GPIO and RS-232). It provides compatible command sets if you want to use VISA in both Windows and Linux environments. This way you have a choice to use either the built-in capabilities of the Linux operating system (as explained in this application note) or to load a VISA library that may provide a more familiar command set. For more information go to **www.tamsinc.com.** The product number is 82091.

Since the VISA implementations available today are not open source, they are not transportable to other Linux distributions. This application note is focused on those situations where a VISA implementation is not the preferred option.

To send a SCPI command to an instrument, a USBTMC driver wraps the command into the message structure as shown and instructs the USB core driver to process the message (i.e. send it to the device's bulk out endpoint). Figure 2 shows an example of the corresponding code.

A couple of things are noteworthy about the code shown. The function `copy_from_user()` is a kernel function that copies data from user space to kernel memory. Unlike `memcpy()`, it takes care of paging issues (pages missing in memory).

The next few lines of code are designed to add alignment bytes if necessary. According to the USBTMC specification, the total number of bytes in the message must be a multiple of four.

Figure 2. Example code: Sending a SCPI command via a DEV_DEP_MSG_OUT message

```
// Setup IO buffer for DEV_DEP_MSG_OUT message
usbtmc_buffer[0]=1; // DEV_DEP_MSG_OUT
usbtmc_buffer[1]=bTag; // Transfer ID (bTag)
usbtmc_buffer[2]=~(bTag); // Inverse of bTag
usbtmc_buffer[3]=0; // Reserved
usbtmc_buffer[4]=command_length&255; // Transfer size (first byte)
usbtmc_buffer[5]=(command_length>>8)&255; // Transfer size (second byte)
usbtmc_buffer[6]=(command_length>>16)&255; // Transfer size (third byte)
usbtmc_buffer[7]=(command_length>>24)&255; // Transfer size (fourth byte)
usbtmc_buffer[8]=1; // Message ends with this transfer
usbtmc_buffer[9]=0; // Reserved
usbtmc_buffer[10]=0; // Reserved
usbtmc_buffer[11]=0; // Reserved

// Append write buffer (instrument command) to USBTMC message
if(copy_from_user(&(usbtmc_buffer[12]),command_buffer,command_length)) {
        // There must have been an addressing problem
        return -EFAULT;
}

// Add zero bytes to achieve 4-byte alignment
n_bytes=12+command_length;
if(command_length%4) {
        n_bytes+=4-command_length%4;
        for(n=12+command_length;n<n_bytes;n++) usbtmc_buffer[n]=0;
}

// Create pipe for bulk out transfer
pipe=usb_sndbulkpipe(usb_dev,bulk_out);

// Send bulk URB
retval=usb_bulk_msg(usb_dev,pipe,usbtmc_buffer,n_bytes,
        &actual,USBTMC_USB_TIMEOUT);
```

The function `usb_sndbulkpipe()` assembles information about the endpoint we are going to use. Finally, `usb_bulk_msg()` asks the kernel to process the message. The latter two functions are part of the services offered by the USB core layer.

Reading data from an instrument works similarly. First, a `DEV_DEP_MSG_IN` message is sent to the bulk out endpoint, asking the instrument to send data in a subsequent read transaction. Then, the data is read from the instrument's bulk in endpoint. For details, see the USBTMC specification and inspect the example code that accompanies this application note.

Registration with the USB Core

The USB core shown in Figure 1 does much more than just process USB messages on behalf of a higher-level driver. It is a facilitator between the USB devices attached and the various higher-layer services installed. It also helps manage hot-plugging of USB devices.

To be able to interact with the USB core—especially for notification that devices are being attached and identifying what they are—higher-level drivers need to register with the USB core.

A key element of this registration process is telling the USB core which devices a higher-level driver would like to service when they become available. Wanted devices can be filtered by various attributes, including the devices' vendor ID, product ID or device class. In the context of USBTMC, it is most appropriate to filter by device class (application-specific) and USBTMC subclass. The USBTMC driver would then get notified whenever a USBTMC-compatible device is being attached, independent of its vendor or product code.

Figure 3 shows how the example driver that accompanies this application note registers with the USB core.

Figure 3: Example code: Registering a USB higher-level driver with the USB core layer

```
// This list defines which devices are serviced by this driver. This driver
// handles USBTMC devices, so we look for the corresponding class (application
// specific) and subclass (USBTMC).
static struct usb_device_id usbtmc_devices[] = {
        {.match_flags=USB_DEVICE_ID_MATCH_INT_CLASS |
                USB_DEVICE_ID_MATCH_INT_SUBCLASS,
                // Device class and sub class need to match to be notified by the system
                .bInterfaceClass=254, // 254 = application specific
                .bInterfaceSubClass=3}, // 3 = test and measurement class (USBTMC)
        { } // Terminating entry
};

// This structure contains registration information for the driver. The
// information is passed to the system through usb_register(), called in the
// driver's init function.
static struct usb_driver usbtmc_driver;
// This structure is used to pass information about this USB driver to the
// USB core (via usb_register)
static struct usb_driver usbtmc_driver = {
        .name="USBTMC", // Driver name
        .id_table=usbtmc_devices, // Devices serviced by the driver
        .probe=usbtmc_probe, // Probe function (called when device is connected)
        .disconnect=usbtmc_disconnect // Disconnect function
};
// Register USB driver with USB core
if((retcode=usb_register(&usbtmc_driver))) {
        printk(KERN_ALERT "USBTMC: Unable to register driver\n");
        goto exit_usb_register;
}
```

Again, a couple of things are noteworthy about the code. The first section assembles a list of structures that will tell the USB core in which types of devices we are interested. In this case, the list has a single entry for USBTMC devices.

Next, the structure of type `usb_driver` is filled. It holds information the USB core layer needs to know in order to register a higher-layer driver. In addition to a pointer to the filter conditions mentioned above, it contains the addresses of a `probe()` and `disconnect()` function.

`probe()` is called by the USB core to notify the higher-layer driver of a newly attached device. It allows the driver to allocate memory, initialize its internal data structures and, in general, get ready for servicing the new device.

`disconnect()` is called to tell the driver that the device is not available anymore. The driver will typically clean up its internal data structures and free any memory or other resources it allocated during the execution of the `probe()` function.

Access to the Driver from User Space

USBTMC-compatible instruments are controlled through text commands, typically—but not necessarily—following the SCPI standard. Likewise, measurement results or other data is usually returned as human-readable text. In other words, communicating with a USB instrument is stream-oriented, very much like reading from and writing to a text file. For such text-based devices, using a character device driver as a window into user space is a frequent (if not obvious) choice.

The beauty of a character device driver is that it behaves like a regular text file. Consequently, you can use standard file I/O system calls to send data to and from a device. Likewise, the output of a console application can be redirected to the device. Character device drivers offer tremendous flexibility.

Character device drivers need to implement a number of entry points that the system calls in order to interact with a device behind the driver. The most basic ones are `open()`, `read()`, `write()` and `release()`, and they correspond to the system calls `open(2)`, `read(2)`, `write(2)` and `close(2)`, respectively.

In the context of USBTMC, the `write()` entry point takes the string to be written and wraps it into a `USBTMC DEV_DEP_MSG_OUT` message. Similarly, the `read()` entry point uses a `DEV_DEP_MSG_IN` message to read data from a device, extract the instrument message part from the return data and copy it to the supplied user buffer.

A key concept about device drivers is that of *major* and *minor numbers*. Device files are created using the mknod(1) command, and the major number specified refers to a character driver behind the (arbitrary) device file name. The minor number is typically used to specify which device the driver will control if several devices are being serviced by the same driver.

When a character device driver is loaded into the kernel, it first needs to register its major and minor numbers with the kernel and publish its entry points. Figure 4 shows the corresponding lines from the example driver available with this application note.

The first section of the code shown dynamically allocates a free major number and a range of minor numbers to use with the driver.

The structure of type `file_operations` is initialized to hold the addresses of the various entry points the driver is going to publish for file I/O. The code then allocates and fills the cdev structure that describes the new character driver and finally uses the `cdev_add()` function to activate the new driver. From this point on, the driver should be ready for calls to its previously published entry points.

Figure 4: Example code: Registering a character device driver

```
// Dynamically allocate char driver major/minor numbers
if((retcode=alloc_chrdev_region(&dev, // First major/minor number to use
         0, // First minor number
         USBTMC_MINOR_NUMBERS, // Number of minor numbers to reserve
         "USBTMCCHR" // Char device driver name
         ))) {
         printk(KERN_ALERT "USBTMC: Unable to allocate major/minor numbers\n");
         goto exit_alloc_chrdev_region;
}

// This structure is used to publish the char device driver functions
static struct file_operations fops = {
         .owner=THIS_MODULE,
         .read=usbtmc_read,
         .write=usbtmc_write,
         .open=usbtmc_open,
         .release=usbtmc_release,
         .ioctl=usbtmc_ioctl,
         .llseek=usbtmc_llseek,
};

// Initialize cdev structure for this character device
cdev_init(&cdev,&fops);
cdev.owner=THIS_MODULE;
cdev.ops=&fops;

// Combine major and minor numbers
printk(KERN_NOTICE "USBTMC: MKDEV\n");
devno=MKDEV(MAJOR(dev),n);

// Add character device to kernel list
printk(KERN_NOTICE "USBTMC: CDEV_ADD\n");
if((retcode=cdev_add(&cdev,devno,1))) {
         printk(KERN_ALERT "USBTMC: Unable to add character device\n");
         goto exit_cdev_add;
}
```

Compiling and Installing the USBTMC Driver

The example driver that comes with this application note is available at **http://www.agilent.com/find/linux** in the form of a TAR archive. Copy the archive to a suitable (empty) directory and extract it using the command tar -x an1465-30.tar.

The extracted files include the source files, as well as a makefile. Compile the driver using the make(1) command. This will create a fresh usbtmc.ko file (kernel object file). Note that, to compile the driver, you will need a kernel sources tree installed on your system. It is typically available on your distribution's media but often not installed by default (look for a package named "kernel-source").

You can now install the driver module in the running kernel using the command `insmod ./usbtmc.ko`. Similarly, the module can be unloaded from the kernel using `rmmod usbtmc`. You need root privileges to run these commands.

To use the driver, first create the proper device files. To do that, you need to know which major number the driver uses. (It is allocated dynamically in the initialization routine when you install the driver, i.e. when running insmod.) The easiest way to get that information is by reading `/proc/devices` using the command: `cat /proc/devices|grep USBTMCCHR`.

With the major number returned by the above command, you can now create the device files using `mknod/dev/usbtmc0 c 253 0` (and similar)—where 253 would be the major number allocated by the driver. Finally, use `chmod(1)` to set the read/write bits appropriately.

The driver comes with a shell script named `usbtmc_load` that automates the above steps. It is shown in Figure 5.

Figure 5: Module load script

```
#!/bin/sh

module="usbtmc"

# Remove module from kernel (just in case it is still running)
/sbin/rmmod $module

# Install module
/sbin/insmod ./$module.ko

# Find major number used
major=$(cat /proc/devices | grep USBTMCCHR | awk '{print $1}')
echo Using major number $major

# Remove old device files
rm -f /dev/${module}[0-9]

# Ceate new device files
mknod /dev/${module}0 c $major 0
mknod /dev/${module}1 c $major 1
mknod /dev/${module}2 c $major 2
mknod /dev/${module}3 c $major 3
mknod /dev/${module}4 c $major 4
mknod /dev/${module}5 c $major 5
mknod /dev/${module}6 c $major 6
mknod /dev/${module}7 c $major 7
mknod /dev/${module}8 c $major 8
mknod /dev/${module}9 c $major 9

# Change access mode (RW access for everybody)
chmod 666 /dev/${module}0
chmod 666 /dev/${module}1
chmod 666 /dev/${module}2
chmod 666 /dev/${module}3
chmod 666 /dev/${module}4
chmod 666 /dev/${module}5
chmod 666 /dev/${module}6
chmod 666 /dev/${module}7
chmod 666 /dev/${module}8
chmod 666 /dev/${module}9
```

Using the USBTMC Driver

The example USBTMC driver dynamically issues the next free (unused) minor number to each USBTMC device attached—in the order the USB core notifies the driver of the existence of the new USB devices. To communicate with an instrument, you need to know which minor number that device is using. In the example USBTMC driver, that information is available by reading from minor number 0. In other words, minor number 0 is reserved for communication with the USBTMC driver itself.

After attaching your USB devices (or after booting the system with the instruments already attached), you can read a list of the devices using `cat/dev/usbtmc0`. This will return the product number, manufacturer ID, serial number and minor number of each device.

You can send SCPI commands to a device by redirecting the command string to its device file. For example, you can reset the first USBTMC device using `echo *RST>/dev/usbtmc1`.

Likewise, you can read from a USBTMC device using cat. For example, `echo *IDN?>/dev/usbtmc1`, followed by `cat/dev/usbtmc1` would print the device's ID string (see Figure 6).

Figure 6: Interactive instrument control using echo and cat

```
skopp@A0071584:~/Projects/usbtmc/src> make
make    -C /lib/modules/2.6.18.2-34-default/build       SUBDIRS=/home/skopp/Projects/usbtmc/
src modules
make[1]: Entering directory `/usr/src/linux-2.6.18.2-34-obj/i386/default'
make -C ../../../linux-2.6.18.2-34 O=../linux-2.6.18.2-34-obj/i386/default modules
  CC [M]  /home/skopp/Projects/usbtmc/src/usbtmc.o
  Building modules, stage 2.
  MODPOST
  LD [M]  /home/skopp/Projects/usbtmc/src/usbtmc.ko
make[1]: Leaving directory `/usr/src/linux-2.6.18.2-34-obj/i386/default'
skopp@A0071584:~/Projects/usbtmc/src> su
Password:
A0071584:/home/skopp/Projects/usbtmc/src # ./usbtmc_load
ERROR: Module usbtmc does not exist in /proc/modules
Using major number 253
A0071584:/home/skopp/Projects/usbtmc/src # cat /dev/usbtmc0
Minor Number    Manufacturer    Product Serial Number
001     Agilent Technologies    34980A Switch Measure Unit      MY44003719
A0071584:/home/skopp/Projects/usbtmc/src # echo *RST>/dev/usbtmc1
A0071584:/home/skopp/Projects/usbtmc/src # echo *IDN?>/dev/usbtmc1
A0071584:/home/skopp/Projects/usbtmc/src # cat /dev/usbtmc1
Agilent Technologies,34980A,MY44003719,2.19-2.19-2.07-1.05
```

In your (automated) test application, simply use file IO system calls to send the appropriate SCPI command strings to your devices (or read back their responses). Figure 7 shows a basic example.

Figure 7: Programmatic instrument control using file IO system calls

```
#include <stdio.h>
#include <fcntl.h>

main()
{
        int myfile;
        char buffer[4000];
        int actual;
        myfile=open("/dev/usbtmc1",O_RDWR);
        if(myfile>0)
        {
                write(myfile,"*IDN?\n",6);
                actual=read(myfile,buffer,4000);
                buffer[actual]=0;
                printf("Response:\n%s\n",buffer);
                close(myfile);
        }
}
```

Summary

The majority of USB measurement devices available today adhere to the USBTMC specification. To use these devices, you need a USBTMC device driver. The techniques described in this application note demonstrate how to create a generic driver for use with all current Linux distributions and versions. The driver is implemented as a character device driver and, as a result, instrument access is possible through output redirection and simple system calls for file I/O.

---

[1] Linux Device Drivers, Jonathan Corbet/Alessandro Rubini/ Greg Kroah-Hartman, O'REILLY

[2] USB Test and Measurement Class Specifications, USB Implementers Forum, **http://www.usb.org/developers/devclass_docs#approved**

Related Agilent literature

The 1465 series of application notes provides a wealth of information about the creation of test systems, the successful use of LAN, WLAN and USB in those systems, and the optimization and enhancement of RF/microwave test systems:

- *Test-System Development Guide: A Comprehensive Handbook for Test Engineers*
  (pub no. 5989-5367EN)
  http://cp.literature.agilent.com/litweb/pdf/5989-5367EN.pdf

Test System Development

- *Test System Development Guide:*
  Application Notes 1465-1 through 1465-8 (pub no. 5989-2178EN)
  http://cp.literature.agilent.com/litweb/pdf/5989-2178EN.pdf
- *Using LAN in Test Systems: The Basics*
  AN 1465-9 (pub no. 5989-1412EN)
  http://cp.literature.agilent.com/litweb/pdf/5989-1412EN.pdf
- *Using LAN in Test Systems: Network Configuration*
  AN 1465-10 (pub no. 5989-1413EN)
  http://cp.literature.agilent.com/litweb/pdf/5989-1413EN.pdf
- *Using LAN in Test Systems: PC Configuration*
  AN 1465-11 (pub no. 5989-1415EN)
  http://cp.literature.agilent.com/litweb/pdf/5989-1415EN.pdf
- *Using USB in the Test and Measurement Environment*
  AN 1465-12 (pub no. 5989-1417EN)
  http://cp.literature.agilent.com/litweb/pdf/5989-1417EN.pdf
- *Using SCPI and Direct I/O vs. Drivers*
  AN 1465-13 (pub no. 5989-1414EN)
  http://cp.literature.agilent.com/litweb/pdf/5989-1414EN.pdf
- *Using LAN in Test Systems: Applications*
  AN 1465-14 (pub no. 5989-1416EN)
  http://cp.literature.agilent.com/litweb/pdf/5989-1416EN.pdf
- *Using LAN in Test Systems: Setting Up System I/O*
  AN 1465-15 (pub no. 5989-2409)
  http://cp.literature.agilent.com/litweb/pdf/5989-2409EN.pdf
- *Next-Generation Test Systems: Advancing the Vision with LXI*
  AN 1465-16 (pub no. 5989-2802)
  http://cp.literature.agilent.com/litweb/pdf/5989-2802EN.pdf

RF and Microwave Test Systems

- *Optimizing the Elements of an RF/Microwave Test System*
  AN 1465-17 (pub no. 5989-3321)
  http://cp.literature.agilent.com/litweb/pdf/5989-3321EN.pdf
- *6 Hints for Enhancing Measurement Integrity in RF/Microwave Test Systems*
  AN 1465-18 (pub no. 5989-3322)
  http://cp.literature.agilent.com/litweb/pdf/5989-3322EN.pdf
- *Calibrating Signal Paths in RF/Microwave Test Systems*
  AN 1465-19 (pub no. 5989-3323)
  http://cp.literature.agilent.com/litweb/pdf/5989-3323EN.pdf

LAN eXtensions for Instrumentation(LXI)

- *LXI: Going Beyond GPIB, PXI and VXI*
  AN 1465-20 (pub no. 5989-4371)
  http://cp.literature.agilent.com/litweb/pdf/5989-4371EN.pdf
- *10 Good Reasons to Switch to LXI*
  AN 1465-21 (pub no. 5989-4372)
  http://cp.literature.agilent.com/litweb/pdf/5989-4372EN.pdf
- *Transitioning from GPIB to LXI*
  AN 1465-22 (pub no. 5989-4373)
  http://cp.literature.agilent.com/litweb/pdf/5989-4373EN.pdf
- *Creating Hybrid Systems with PXI, VXI and LXI*
  AN 1465-23 (pub no. 5989-4374)
  http://cp.literature.agilent.com/litweb/pdf/5989-4374EN.pdf
- *Using Synthetic Instruments in Your Test System*
  AN 1465-24 (pub no. 5989-4375)
  http://cp.literature.agilent.com/litweb/pdf/5989-4375EN.pdf
- *Migrating System Software from GPIB to LAN/LXI*
  AN 1465-25 (pub no. 5989-4376)
  http://cp.literature.agilent.com/litweb/pdf/5989-4376EN.pdf
- *Modifying a GPIB System to Include LAN/LXI*
  AN 1465-26 (pub no. 5989-6824)
  http://cp.literature.agilent.com/litweb/pdf/5989-6824EN.pdf

Using Linux in Your Test Systems

Example code is available for download at **http://www.agilent.com/find/linux**

- *Using Linux in Your Test Systems: Linux Basics*
  AN 1465-27 (pub no. 5989-6715)
  http://cp.literature.agilent.com/litweb/pdf/5989-6715EN.pdf
- *Using Linux to Control LXI Instruments Through VXI-11*
  AN 1465-28 (pub no. 5989-6716)
  http://cp.literature.agilent.com/litweb/pdf/5989-6716EN.pdf
- *Using Linux to Control LXI Instruments Through TCP*
  AN 1465-29 (pub no. 5989-6717)
  http://cp.literature.agilent.com/litweb/pdf/5989-6717EN.pdf

www.agilent.com/find/open

Agilent Email Updates

www.agilent.com/find/emailupdates
Get the latest information on the products and applications you select.

Agilent Direct

www.agilent.com/find/agilentdirect
Quickly choose and use your test equipment solutions with confidence.

Agilent Open

www.agilent.com/find/open
Agilent Open simplifies the process of connecting and programming test systems to help engineers design, validate and manufacture electronic products. Agilent offers open connectivity for a broad range of system-ready instruments, open industry software, PC-standard I/O and global support, which are combined to more easily integrate test system development.

LXI

www.lxistandard.org
LXI is the LAN-based successor to GPIB, providing faster, more efficient connectivity. Agilent is a founding member of the LXI consortium.

Remove all doubt

Our repair and calibration services will get your equipment back to you, performing like new, when promised. You will get full value out of your Agilent equipment throughout its lifetime. Your equipment will be serviced by Agilent-trained technicians using the latest factory calibration procedures, automated repair diagnostics and genuine parts. You will always have the utmost confidence in your measurements.

Agilent offers a wide range of additional expert test and measurement services for your equipment, including initial start-up assistance onsite education and training, as well as design, system integration, and project management.

For more information on repair and calibration services, go to:

www.agilent.com/find/removealldoubt

www.agilent.com

For more information on Agilent Technologies' products, applications or services, please contact your local Agilent office. The complete list is available at:

www.agilent.com/find/contactus

Americas

Canada	(877) 894-4414
Latin America	305 269 7500
United States	(800) 829-4444

Asia Pacific

Australia	1 800 629 485
China	800 810 0189
Hong Kong	800 938 693
India	1 800 112 929
Japan	0120 (421) 345
Korea	080 769 0800
Malaysia	1 800 888 848
Singapore	1 800 375 8100
Taiwan	0800 047 866
Thailand	1 800 226 008

Europe & Middle East

Austria	0820 87 44 11
Belgium	32 (0) 2 404 93 40
Denmark	45 70 13 15 15
Finland	358 (0) 10 855 2100
France	0825 010 700* *0.125 € fixed network rates
Germany	01805 24 6333** **0.14 €/minute
Ireland	1890 924 204
Israel	972-3-9288-504/544
Italy	39 02 92 60 8484
Netherlands	31 (0) 20 547 2111
Spain	34 (91) 631 3300
Sweden	0200-88 22 55
Switzerland (French)	41 (21) 8113811(Opt 2)
Switzerland (German)	0800 80 53 53 (Opt 1)
United Kingdom	44 (0) 118 9276201

Other European Countries:
www.agilent.com/find/contactus

Revised: October 24, 2007

Product specifications and descriptions in this document subject to change without notice.

Printed in USA, November 7, 2007
5989-6718EN

Agilent Technologies