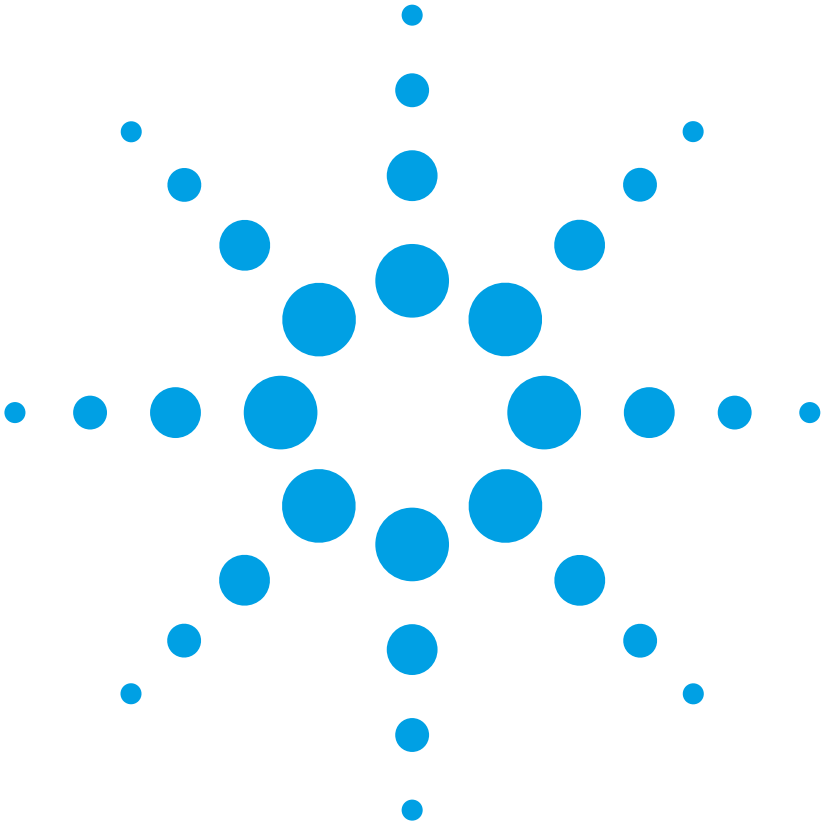
Using Linux to Control LXI Instruments Through TCP

Application Note 1465-29



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 LXI Instruments Through TCP* is part of a series of application notes designed to explain how to control your test instruments under Linux. Example code is available for download at **http://www.agilent.com/find/linux**.

Table of contents

Agilent Technologies

LXI and LAN-based instruments

Agilent has been offering instruments with LAN interfaces for many years. In 2004, with the inception of the LXI Consortium[1], momentum grew and LAN-based instruments became increasingly popular and widely accepted in the test industry.

Some of Ethernet's advantages are obvious, like its low cost and suitability for distributed and remote applications. Other aspects are less obvious but equally important. These include exceptional performance with Gigabit Ethernet and a new level of flexibility enabled by multicast (one-to-many), peer-to-peer and quasi-simultaneous communication.

The move towards Ethernet is great news for Linux (and other non-Windows operating systems) users because they can use the operating system's built-in standard API to control their instruments. Interfaces like GPIB or MXI (specific to the test industry) or PCI cards require special driver software for a given operating system flavor—which may not be available.

TCP/IP protocols used for instrument control

In 2000, the VXI*plug&play* Alliance[2] added support for LAN-based instruments to its VISA specifications. Two popular methods of instrument control via Ethernet were adopted by VISA: VXI-11[3] and "direct" TCP socket communication (see Figure 1).

VXI-11 was originally designed to mimic the capabilities of GPIB, including those based on hardware signals, such as service requests (SRQs), serial polls, device triggers and device clears. It was first used in LAN-to-GPIB gateways, before native LAN-based instruments were available. It is based on remote procedure calls (RPC). A single server like the LAN-to-GPIB gateway can facilitate access to a number of logical devices, such as GPIB instruments behind the gateway. Although VXI-11 was designed for LAN-to-GPIB gateways, it is often supported in native LAN-based instruments as well for compatibility. You can learn more about this type of connection in Agilent Application Note 1465-28, *Using Linux to Control LXI Instruments Through VXI-11*.

The other method of instrument control is socket communication, where an instrument is controlled through a direct TCP socket connection in a stream-oriented manner, which is similar to writing to and reading from a disk file.

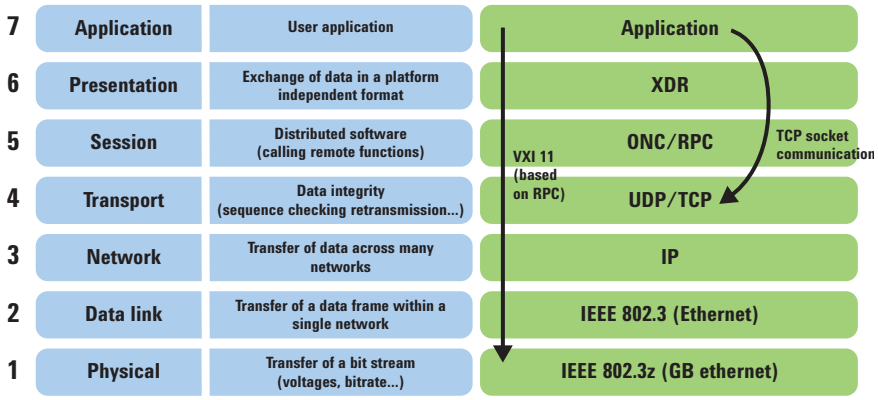
VXI-11 or TCP sockets: Which should you use?

VXI-11 is used exclusively if you are accessing GPIB instruments through a LAN-to-GPIB gateway like the Agilent E5810A or if you are using a PC as a gateway. However, many native LAN instruments support both VXI-11 and TCP socket communication. Which is the better option? Often, it is a matter of preference. However, VXI-11 is the more complex (higher-layer) protocol (as shown in Figure 1). Consequently, direct socket communication will provide better performance in many situations, especially if the actual measurement time is short and you conduct many individual transactions. Furthermore, as the examples in this document will show, sockets are considerably easier to use. Therefore, if they are supported by the instrument, sockets are the recommended approach for native LAN instruments.

Sockets and transport options

A socket is an "endpoint for communication" between two systems, similar to a post box or a telephone number. Communication across sockets usually works both ways (duplex) and, like device drivers and other methods of communication, it is based on a stream model: data is transferred as a stream of bytes (characters), and the underlying API works in much the same way as reading and writing data to a disk file works.

Figure 1. TCP/IP layers and their use for instrument control



The operating system takes care of the "ugly details" for us. For example, it wraps chunks of the byte stream into IP packets and sends them to the destination system. With IP at the network layer, as shown in Figure 1, there are two alternative protocols that are used at the transport layer: TCP and UDP.

TCP is "connection-based:" the operating systems make sure that packets are transmitted successfully (by way of an acknowledgment) and put back in order (by way of sequence numbers). UDP, on the other side, is connectionless. Theoretically, packets could get lost (there is no acknowledgment) and they might arrive at the destination in the wrong order. Given these drawbacks, instrument control is typically done through TCP and we live with TCP's slight increase in overhead compared to UDP.

By convention, Agilent instruments and other vendors' products use TCP and port 5025 for control.

API calls for socket communication

As mentioned above, socket communication is supported by the Linux operating system—and it is straightforward. Table 1 lists basic operating system calls for socket communication.

Figure 2 shows an example program that establishes a TCP connection to an instrument and retrieves the instrument's ID string.

The first step is to create a socket for a particular protocol family (in this case, `PF_INET`, i.e. IPv4) and session type (here, `SOCK_STREAM`, i.e. TCP) through a call to `socket()`. This reserves and initializes appropriate system resources for the TCP link.

The `connect()` call establishes the connection to the server. It accepts a pointer to a data structure that contains the details for the wanted connection. The port number and IP address of the server need to be specified in "network order" (see below).

At this point, we are ready to communicate with the instrument. Commands are sent through a call to `send()`. Note that SCPI strings are terminated with a "\n" (new line) character. The instrument's response is read through a call to `recv()`. It also ends with a new line character—a zero character needs to be appended to turn the response into a C string (at least if you want to use C string functions, such as `printf()`, for post processing).

Finally, the socket is closed through a call to `close()`.

The `recv()` call will only return if data is available (or after data becomes available). In other words, there is no integrated timeout mechanism. It is therefore essential to implement proper timeout handling. One way of doing so is to use the `select()` call described in a later section of this document.

Network order

Ethernet communication is platform-independent by design. This independence is achieved by defining the overall format of Ethernet packets, as well as the byte ordering of individual data fields in the packets, like the IP address and the port number. In this context, the chosen byte order (big-endian[4]) is also known as "network order."

When using an API call like `connect()`, the data passed in as the IP address and the port number is used directly by the system for building the appropriate TCP messages. Consequently, the parameters need to be passed to the API function in network order.

A number of functions are available to convert data types from the native processor format to network order (and back). For example, `htons()` is used to convert the port number (an unsigned short) to network order. Also, `inet_addr()`, a function that converts an IP address string (in dot notation) to a 32-bit unsigned integer, returns the latter in network order.

Table 1. Basic Linux system calls for socket communication

Operating system call	Description
`socket()`	Creates a socket on the client (controller) for a particular protocol family (for example, IPv4 or IPv6) and protocol type (connectionless/UDP or connection-based/TCP). See socket(2) man page for details.
`connect()`	Initiates a socket connection to a given server (referenced by an IP address and port number). See connect(2) man page for details.
`send()`	Sends a message (for example, a SCPI command) to the instrument. See send(2) man page for details.
`recv()`	Reads data (for example, measurement results) from the instrument. See recv(2) man page for details.
`close()`	Closes the connection initiated by connect(). See close(2) man page for details.

Figure 2. Basic SCPI communication through TCP

```
int MySocket;
if((MySocket=socket(PF_INET,SOCK_STREAM,0))==-1) exit(1);

struct in_addr {
   unsigned long s_addr;
};
struct sockaddr_in {
   short int sin_family; // Address family
   unsigned short int sin_port; // Port number
   struct in_addr sin_addr; // Internet address
   unsigned char sin_zero[8]; // Padding
};
struct sockaddr_in MyAddress;

// Initialize the whole structure to zero
memset(&MyAddress,0,sizeof(struct sockaddr_in));
// Then set the individual fields
MyAddress.sin_family=PF_INET; // IPv4
MyAddress.sin_port=htons(5025); // Port number used by most instruments
MyAddress.sin_addr.s_addr=inet_addr("169.254.9.80"); // IP Address

// Establish TCP connection
if(connect(MySocket,(struct sockaddr *)&MyAddress,
   sizeof(struct sockaddr_in))==-1) exit(1);

// Send SCPI command
if(send(MySocket,"*IDN?\n",6,0)==-1) exit(1);

// Read response
char buffer[200];
int actual;
if((actual=recv(MySocket,&buffer[0],200,0))==-1) exit(1);
buffer[actual]=0; // Add zero character (C string)
printf("Instrument ID: %s\n",buffer);

// Close socket
if(close(MySocket)==-1) exit(1);
```

Nagle's algorithm/ TCP_NODELAY

Most operating systems, including Linux, use Nagle's algorithm (named after its inventor, John Nagle, see RFC896[5]) to make TCP communication more effective. With this algorithm, the sending of small packets is delayed for short periods of time.

The idea is that sending several small pieces in one combined packet is more effective than sending individual packets (given the overhead of TCP and Ethernet).

This works great for many applications. However, using this algorithm often is not desirable in measurement applications due to the increase in latency. The `setsockopt()` system call can be used to set the `TCP_NODELAY` option for a given socket. This will instruct the system not to use Nagle's algorithm (and send even small messages immediately). The code fragment shown in Figure 3 sets the `TCP_NODELAY` option.

`setsockopt()` can also be used to modify various other parameters, such as the send/receive buffer size allocated for a socket connection. See the `setsockopt(2) man` page for further information.

Figure 3. The TCP_NODELAY option minimizes latency by deactivating Nagle's algorithm.

```
#include <netinet/tcp.h>
#include <netinet/in.h>

int StateNODELAY = 1; // Turn NODELAY on

setsockopt(MySocket,IPPROTO_TCP,TCP_NODELAY,
    (void *)&StateNODELAY,sizeof StateNODELAY);
```

Control port/Device clear

In addition to the "regular" socket connection used above, most Agilent instruments support a control connection. This separate TCP link is used for messages that require immediate delivery, namely device clear messages and SRQs (service requests).

The port number to be used for the control connection is not standardized. However, a special query command, `SYST:COMM:TCPIP:CONTROL?` (sent through the regular connection), is available to query the port number to use for the control connection for a given instrument.

Note: Not all instruments support a control connection. Therefore, it is important to verify the instrument's response. A valid port number (greater then zero) indicates that the control connection is supported by the instrument.

One of the important functionalities enabled through the control connection is a "device clear" that clears the instrument's communication buffers. It often can be used to regain control over the instrument when the communication gets "stuck" due to communication problems.

A device clear is initiated by sending the string "DCL\n" through the control connection. Note that the instrument echoes the command back as an acknowledgment.

The example code shown in Figure 4 sets up a link to the control port and uses it to clear the device.

Figure 4. Using the control port to clear an instrument

```
void send_string(int MySocket,char string[])
{
    if(send(MySocket,string,strlen(string),0)==-1) {
        /* Do error handling here */
    }
    return;
}

void read_string(int MySocket,char *buffer)
{
    int actual;
    if((actual=recv(MySocket,buffer,200,0))==-1) {
        /* Do error handling here */
    }
    else buffer[actual]=0;
    return;
}

send_string(MySocket,"SYST:COMM:TCPIP:CONTROL?\n");
char buffer[200];
read_string(MySocket,buffer);
unsigned int ControlPort;
sscanf(buffer,"%u",&ControlPort);
printf("Control Port: %u\n",ControlPort);

int MyControlSocket;
if((MyControlSocket=socket(PF_INET,SOCK_STREAM,0))==-1) {
    /* Do error handling here */
}

struct sockaddr_in MyControlAddress;
memset(&MyControlAddress,0,sizeof(struct sockaddr_in));
MyControlAddress.sin_family=PF_INET; /* IPv4 */
MyControlAddress.sin_port=htons((unsigned short)ControlPort);
MyControlAddress.sin_addr.s_addr=inet_addr("169.254.9.80");
if(connect(MyControlSocket,(struct sockaddr
*)&MyControlAddress,
    sizeof(struct sockaddr_in))==-1) {
    /* Do error handling here */
}

send_string(MyControlSocket,"DCL\n");
read_string(MyControlSocket,buffer);
if(strcmp(buffer,"DCL\n")==0)
    printf("DCL\\n received back from instrument...\n");
else printf("Response: %s\n",buffer);

if(close(MyControlSocket)==-1) {
    /* Do error handling here */
}
```

SRQs (Service Requests)

As mentioned above, the control port is also used for SRQs. Instruments use an SRQ to signal the system controller when they need attention (for example, when an error occurs or when measurement results are available in the instrument's output buffer).

The instrument raises an SRQ by sending the string "SRQ" followed by the current value of the instrument's status byte (for example, "SRQ +128\n"). The test application can then react accordingly.

The mechanism described above usually requires some form of asynchronous programming since the application does not know when the instrument will generate an SRQ. This is often implemented using an "SRQ handler" which is set up and then goes to sleep (is suspended) until data becomes available on the control connection (meaning, the instrument has likely raised an SRQ).

One way to achieve this is through the `select()`[6] function. `select()` sleeps until one or several file descriptors (in this case, the handle to the control socket connection) change status. More exactly, as used below, `select()` will return (wake up) when data becomes available (or if the timeout expires).

The example code shown in Figure 5 uses `select()` to wait for SRQs.

First, the set of file descriptors to be monitored by `select()` is set up. `FD_ZERO` and `FD_SET` are macros that are available to manipulate the `fd_set` structure. In this case, only the control port handle is added to the structure.

Next, the instrument is set up to generate an SRQ. In this example, which is based on the 34410A multimeter, the operation complete bit is used to trigger an SRQ.

At this point, `select()` is used to wait for data to become available on the control socket connection.

Note that the `select()` function suspends the current thread until data becomes available (or the timeout expires). Normally, monitoring the control port would be done in a separate thread.

Summary

If sockets are available with your instrument, you may want to choose to use them instead of the VXI-11 protocol. They offer the same functionality at higher performance and they are straightforward to use.

1 For details about LXI (LAN Extensions for Instrumentation) and the LXI Consortium, see **http://www.lxistandard.org**

2 For details about VISA and the VXI*plug&play* Alliance, see **http://www.vxipnp.org**

3 For details about VXI-11, see **http://www.vxibus.org/freepdfdownloads/vxi-11.pdf**

4 In a big-endian system, the most significant byte is stored first (at the lower address). Intel processors use little-endian (least-significant byte first).

5 For details about RFC896, visit **http://www.ietf.org/rfc**

6 See select(2) man page for details.

Figure 5. Using select() to wait for an SRQ

```
fd_set MyFDSet;
struct timeval tv;
int retval;

tv.tv_sec=10; tv.tv_usec=0; // Timeout

FD_ZERO(&MyFDSet); // Clear set
FD_SET(MyControlPort,&MyFDSet); // Add control port

// Cause an SRQ
send_string(MySocket,"*ESE 1\n"); // OPC sets standard event bit
send_string(MySocket,"*SRE 32\n"); // Standard event will cause
SRQ
send_string(MySocket,"CONF:FREQ\n"); // Do something...
send_string(MySocket,"*OPC\n"); // Set OPC when done

retval=select(MyControlPort+1,&MyFDSet,NULL,NULL,&tv);
if(retval==-1) {
   // Do error handling here
}
if(retval==1)
{
    // One connection changed status... Must be control port
    printf("Data available\n");
    read_string(MyControlPort,buffer);
    printf("Data read: %s\n",buffer);
}
```

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. All of the individual notes listed below are also available in a compilation:

- *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**

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	81 426 56 7832
Korea	080 769 0800
Malaysia	1 800 888 848
Singapore	1 800 375 8100
Taiwan	0800 047 866
Thailand	1 800 226 008

Europe

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
Germany	01805 24 6333*
	*0.14€/minute
Ireland	1890 924 204
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: May 7, 2007

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

Printed in USA, June 13, 2007
5989-6717EN

Agilent Technologies