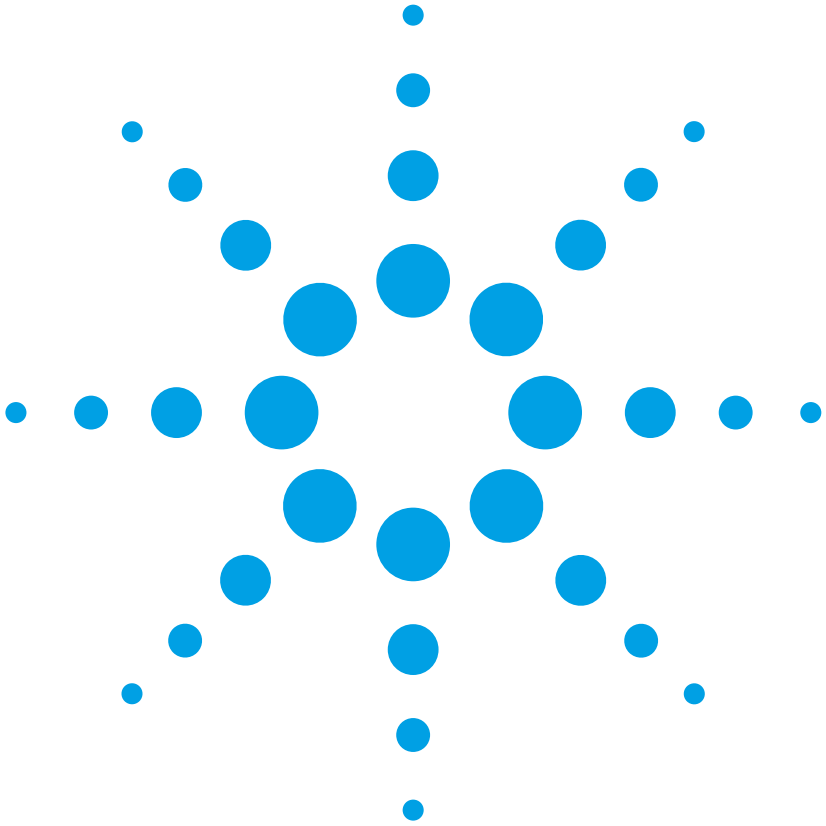
Using Linux to Control LXI Instruments Through VXI-11

Application Note 1465-28



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 VXI-11* 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) users because they can use the operating system's built-in standard API to control 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 request (SRQ), serial poll, device trigger and device clear. It was first used in LAN-to-GPIB gateways, before native LAN-based instruments were available. VXI-11 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 the 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.

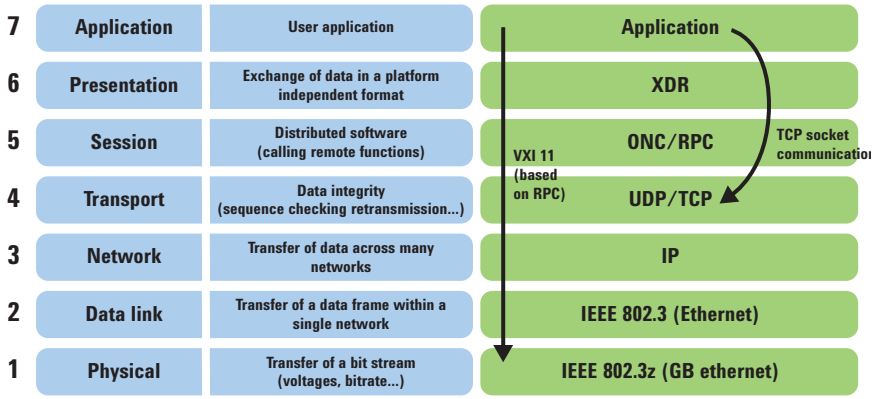
The other method of instrument control is socket communication, where an instrument is controlled through a direct TCP socket connection in a stream-oriented manner, similar to writing to and reading from a disk file. You can learn more about this type of connection in Agilent Application Note 1465-29, *Using Linux to Control LXI Instruments Through TCP.*

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 TCP VXI-11 and 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.

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



Agilent IO Libraries and the LAN server

You can turn a Windows PC into a LAN-to-GPIB gateway using Agilent's IO Libraries Suite. This software package includes a VXI-11 server which makes the PC's local interfaces accessible from the network (and thereby from your Linux controller).

For more information about the Agilent IO Libraries, please visit **http://www.agilent.com/find/iosuite**.

The basis for VXI-11: Remote procedure calls

As we noted earlier, VXI-11 is based on RPC. With RPC, calling a remote function—one that is executing on another machine on the network—is as easy as calling a local function running on the same computer. The operating system hides most, but not all, of the details involved in communicating with the remote system from the user. It offers system calls like `clnt_call()`, which behave very much like a normal function call—the client waits until the server has executed the function and returns the results (synchronous operation, see Figure 2).

RPC was designed to be independent of a particular programming language and computer platform—the RPC server and client can run on different operating systems and processors. Interoperability is achieved through XDR (the data representation layer, see Figure 1), which defines standard data types, as well as the byte ordering to be used in RPC calls. Consequently, the parameters you are passing to the RPC function need to be translated to XDR format and back to native programming language format for return data.

How do you know which functions an RPC server offers and which parameters they accept? The server typically comes with a description of its interfaces in the form of an RPCL (RPC language) definition file. RPCL is very similar to type definitions in C. For example, the RPCL file for VXI-11 includes definitions like those shown in Figure 3.

How does a client identify the function it wants to call on a given server? It does so by three distinct numbers: the program, version and procedure number. These numbers are included in the server's RPCL definitions. For example, in Figure 3, 0x0607AF is the program number, 1 is the version number and 10 is the procedure number of the `create_link()` function. The server machine itself is identified on the network through its IP address or hostname.

Figure 2. Synchronous execution of an RPC function through clnt_call()

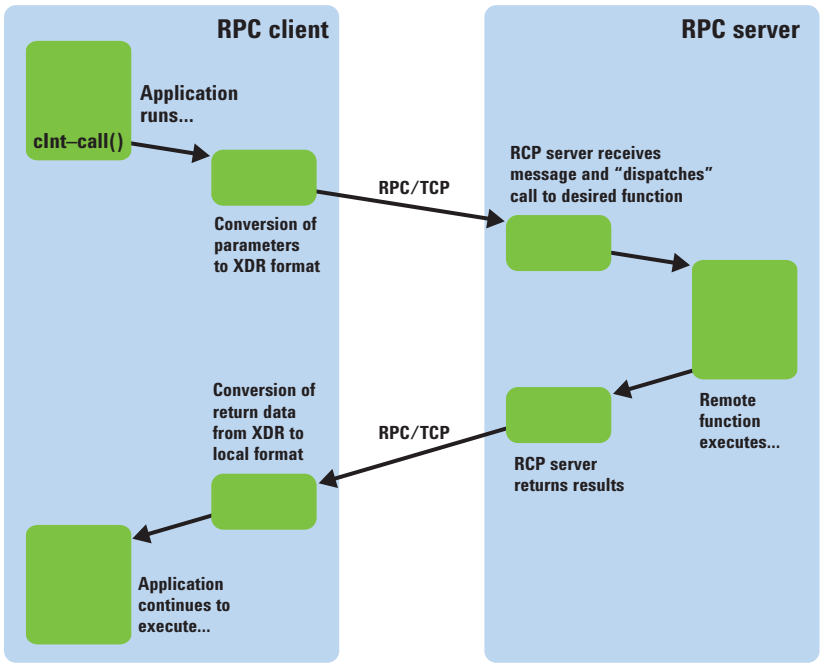


Figure 3. RPCL definition of some key VXI-11 functions

```
program DEVICE_CORE {
    version DEVICE_CORE_VERSION {
        Create_LinkResp create_link (Create_LinkParms) = 10;
        Device_WriteResp device_write (Device_WriteParms) = 11;
        Device_ReadResp device_read (Device_ReadParms) = 12;
        Device_Error destroy_link (Device_Link) = 23;
    } = 1;
} = 0x0607AF;
```

RPCGEN code generator

Using the basic `clnt_call()` operating system call to run RPC functions is a little tedious. Fortunately, the rpcgen tool makes the process easier (see Figure 4). rpcgen turns the server's RPCL definitions into real C declarations. In addition, it generates translator functions (which translate native RPC parameters to XDR), as well as wrapper functions that allow us to call RPC functions using real function names instead of the generic `clnt_call()`.

Figure 5 uses the VXI-11 function `device_write()` as an example and shows how rpcgen works. The original RPCL definition of `device_write()` is shown in Figure 5a. This function is used to send SCPI commands to an instrument (described below in more detail).

One of the files generated by rpcgen is a C header (.h) file. It contains ready-to-use C declarations of the numbers and data types in the RPCL description. Figure 5b shows the definitions generated for `device_write()`, as well as the constant definitions for the program and version number. Note how the generic RPCL "opaque" data type has now been replaced by an equivalent C structure (buffer). Using these C declarations, creating the appropriate structures for the function parameters becomes relatively straightforward. There is no need to worry about XDR.

How do you call the RPC function? The easiest way is to use the wrapper functions generated by rpcgen. Figure 5c shows the corresponding function for `device_write()` in the client implementation (_clnt.c) file. It accepts a pointer to a structure that holds the function's input parameters, as well as a reference to the RPC link. It returns a pointer to a result data structure for access by the calling application. Also, note how the wrapper function passes the addresses of the "xdr" translator functions defined in the xdr.c file to `clnt_call()`.

Figure 4. Files generated by RPCGEN

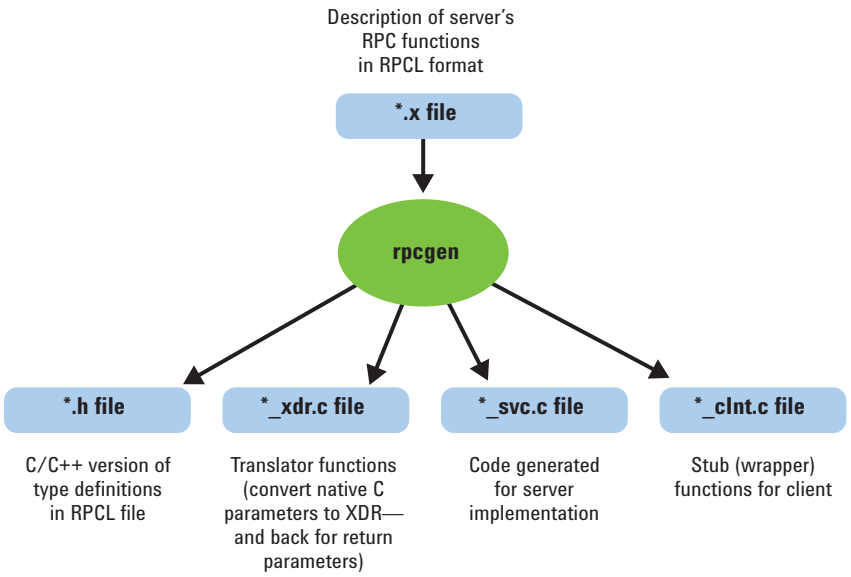


Figure 5a. RPCL definition of the device_write() VXI-11 function

```
struct Device_WriteParms {
   Device_Link lid; /* link id from create_link */
   unsigned long io_timeout; /* time to wait for I/O */
   unsigned long lock_timeout; /* time to wait for lock */
   Device_Flags flags;
   opaque data<>; /* the data length and the data itself */
};
struct Device_WriteResp {
   Device_ErrorCode error;
   unsigned long size; /* Number of bytes written */
};
Device_WriteResp device_write (Device_WriteParms) = 11;
```

Figure 5b. C declarations in the header file generated by rpcgen

```
struct Device_WriteParms {
   Device_Link lid;
   u_long io_timeout;
   u_long lock_timeout;
   Device_Flags flags;
   struct {
      u_int data_len;
      char *data_val;
   } data;
};
typedef struct Device_WriteParms Device_WriteParms;
struct Device_WriteResp {
   Device_ErrorCode error;
   u_long size;
};
typedef struct Device_WriteResp Device_WriteResp;

#define DEVICE_CORE 0x0607AF
#define DEVICE_CORE_VERSION 1
#define device_write 11
```

Figure 5c. Wrapper function for device_write(), generated by rpcgen

```
Device_WriteResp *device_write_1(Device_WriteParms *argp, CLIENT *clnt)
{
   static Device_WriteResp clnt_res;
   memset((char *)&clnt_res, 0, sizeof(clnt_res));
   if (clnt_call (clnt, device_write,
      (xdrproc_t) xdr_Device_WriteParms, (caddr_t) argp,
      (xdrproc_t) xdr_Device_WriteResp, (caddr_t) &clnt_res,
      TIMEOUT) != RPC_SUCCESS) {
      return (NULL);
      }
   return (&clnt_res);
}
```

API calls for RPC

Table 1 lists a number of basic API calls for RPC. There are many more functions and flavors (see rpc(5) man page for more information), but these are sufficient for most applications.

Using basic VXI-11 functions

After creating the link to the RPC server using `clnt_create()`, you can call VXI-11 functions either through the rpcgen wrappers or through direct calls to `clnt_call()`. Table 2 summarizes basic VXI-11 functions.

Table 1. Basic operating system calls for access to an RPC server

API function	Description
`clnt_create()`	Creates an RPC client for access to a particular host and RPC program on that host. This is always the first step (required both for direct use of clnt_call() and when using the wrapper functions generated by rpcgen) See clnt_create(3) man page for details
`clnt_call()`	Calls a remote function offered by the RPC server. This function is used by the wrappers generated by rpcgen See clnt_call(3) man page for details.
`clnt_destroy()`	Destroys the RPC client and releases any resources allocated for the link See clnt_destroy(3) man page for details

Table 2. Basic VXI-11 functions

API function	Description
`create_link`	Establishes a link to a logical device in the VXI-11 server
`device_write`	Sends a message (typically, a SCPI command) to the instrument
`device_read`	Reads data (e.g. measurement results) from the instrument
`destroy_link`	Releases the link established by create_link and frees any resources allocated

Figure 6 shows a basic example. It reads the instrument's ID string through the *IDN? query. First, the header and C files generated by rpcgen are included.

Next, the RPC link to the instrument's VXI-11 server is created through a call to `clnt_create()`. Note that you can specify which transport protocol you would like to use (TCP or UDP). `clnt_create()` returns a pointer which is used to reference the RPC link in subsequent calls.

From here on, the wrapper functions generated by rpcgen are used to call individual VXI-11 functions. `create_link_1()` establishes a VXI-11 link to a given logical device in the VXI-11 server. The return data structure includes a link ID number (lid) that is stored in a local variable—it is needed for subsequent VXI-11 calls to this same logical device. Note that "inst0" is the logical device name used by most native LAN instruments. When using a LAN-to-GPIB gateway, the logical device name refers to a particular GPIB address behind the server, for example "gpib0,9".

Next, `device_write_1()` is used to send the *IDN? command to the instrument. Note that SCPI commands are terminated with a newline ('\n') character.

The instrument's response is read using the VXI-11 `device_read_1()` function. Note the termChar parameter: It defines which character indicates the end of the instrument's response. The response string is copied to a local buffer and a zero character is appended in order to turn the buffer into a standard C string, which is required to be able to use standard C string functions such as `printf()`.

Finally, `destroy_link_1()` releases the link to the logical device. This allows the VXI-11 server to free any resources allocated for the connection. Similarly, `clnt_destroy()` releases the basic RPC connection.

Figure 6. Using VXI-11 to retrieve an instrument's ID string

```
#include "vxi11.h"
#include "vxi11_xdr.c"
#include "vxi11_clnt.c"

CLIENT *VXI11Client;

if((VXI11Client=clnt_create("169.254.9.80",
   DEVICE_CORE,DEVICE_CORE_VERSION,"tcp"))==NULL) {
   /* Do error handling here */
}

Create_LinkParms MyCreate_LinkParms;
MyCreate_LinkParms.clientId = 0; // Not used
MyCreate_LinkParms.lockDevice = 0; // No exclusive access
MyCreate_LinkParms.lock_timeout = 0;
MyCreate_LinkParms.device = "inst0"; // Logical device name
Create_LinkResp *MyCreate_LinkResp;
if((MyCreate_LinkResp=create_link_1(&MyCreate_LinkParms,VXI11Client))==NULL) {
   /* Do error handling here */
}
Device_Link MyLink;
MyLink = MyCreate_LinkResp->lid; // Save link ID for further use
Device_WriteParms MyDevice_WriteParms;
MyDevice_WriteParms.lid = MyLink;
MyDevice_WriteParms.io_timeout = 10000; // in ms
MyDevice_WriteParms.lock_timeout = 10000; // in ms
MyDevice_WriteParms.flags = 0;
MyDevice_WriteParms.data.data_val = "*IDN?\n";
MyDevice_WriteParms.data.data_len = 6;
Device_WriteResp *MyDevice_WriteResp;
if((MyDevice_WriteResp=device_write_1(&MyDevice_WriteParms,VXI11Client))
   ==NULL) {
   /* Do error handling here */
}

Device_ReadParms MyDevice_ReadParms;
MyDevice_ReadParms.lid = MyLink;
MyDevice_ReadParms.requestSize = 200;
MyDevice_ReadParms.io_timeout = 10000;
MyDevice_ReadParms.lock_timeout = 10000;
MyDevice_ReadParms.flags = 0;
MyDevice_ReadParms.termChar = '\n';
Device_ReadResp *MyDevice_ReadResp;
if((MyDevice_ReadResp=device_read_1(&MyDevice_ReadParms,VXI11Client))==NULL) {
   /* Do error handling here */
}
char DataRead[200];
strncpy(DataRead,MyDevice_ReadResp->data.data_val,
   MyDevice_ReadResp->data.data_len);
DataRead[MyDevice_ReadResp->data.data_len]=0;
printf("Instrument ID string: %s\n",DataRead);

if(destroy_link_1(&MyLink,VXI11Client)==NULL)
{
   /* Do error handling here */
}

clnt_destroy(VXI11Client);
```

Additional VXI-11 functions

VXI-11 features a number of additional functions that are straightforward to use. They are listed in Table 3.

Note: Not all of these operations might be supported by a given logical device behind a VXI-11 server (depending on the physical interface between the VXI-11 server and the device). If the server recognizes that a requested operation is not available for a particular device (which may occur), the call will return error code 8, "operation not supported."

Calling the above functions is straightforward. The example in Figure 7 triggers an instrument through a call to `device_trigger_1()`.

Table 3. Additional VXI-11 functions using the core channel

VXI-11 function	Description
`device_readstb()`	Reads the device's status byte. This corresponds to the IEEE488 serial poll operation.
`device_trigger()`	Triggers the device.
`device_clear()`	Clears (resets) the device.
`device_remote()`	Sets the device to remote mode (front panel controls are disabled).
`device_local()`	Sets the device to local mode (front panel controls are enabled).
`device_lock()`	Attempts to lock (get exclusive access to) the device.
`device_unlock()`	Releases the lock for the device.
`device_docmd()`	Executes a device-specific command.

Figure 7. Triggering a device through device_trigger

```
void vxi-11_trigger()
{
    Device_GenericParms MyDevice_GenericParms;
    MyDevice_GenericParms.lid = MyLink;
    MyDevice_GenericParms.flags = 0;
    MyDevice_GenericParms.lock_timeout = 10000;
    MyDevice_GenericParms.io_timeout = 10000;
    if(device_trigger_1(&MyDevice_GenericParms,VXI11Client)==NULL) {
        /* Do error handling here */
    }
}
```

Abort channel

Some VXI-11 operations, for example, `device_abort`, use a separate RPC link, in this case the abort channel. Using a separate channel makes it easier for the instrument to execute this call with due priority (depending on the architecture of the instrument, incoming RPC calls might be serialized).

Figure 8 shows the RPCL definition of the abort channel.

Use of the abort channel follows mostly the same rules and mechanics as the core channel described above. However, there are some differences in the way the RPC link is set up. When setting up the core channel, the port mapper service in the instrument is used to establish the link through the appropriate TCP port number. Consequently, there is no need to specify the port number when establishing a connection through a call to `clnt_create()`.

The abort channel server, in contrast, does not register itself with the port mapper service. In this case, the TCP port number used by the RPC server in the instrument needs to be specified explicitly. The `clnttcp_create()` system call does just that–it accepts a pointer to a structure of type `sockaddr_in` which holds both the IP address and the port number of the destination server. How do we know which port number is being used in the instrument for the abort channel? This information is part of the parameters returned when establishing the core channel link through a call to `clnt_create()`.

As shown in Figure 8, the abort channel RPC server is used for a single RPC function, `device_abort`, only. This function instructs the instrument to abort any RPC operations pending on the core channel. Due to the synchronous nature of most instrument programming, it is seldom used.

SRQs (Service Requests)

VXI-11 includes provisions for service requests. 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).

SRQs use a separate RPC link, the interrupt channel. Due to the nature of SRQs (they originate in the instrument, not the controller), the roles of server and client are reversed: the instrument signals SRQs by calling an RPC function, `device_intr_srq()`, to be executed on the controller.

Figure 9 shows the RPCL description of the interrupt channel.

Figure 8. RPCL description of the abort channel

```
program DEVICE_ASYNC{
    version DEVICE_ASYNC_VERSION {
        Device_Error device_abort (Device_Link) = 1;
    } = 1;
} = 0x0607B0;
```

Figure 9. RPCL description of the interrupt channel

```
struct Device_SrqParms {
    opaque handle<>;
};
program DEVICE_INTR {
    version DEVICE_INTR_VERSION {
        void device_intr_srq (Device_SrqParms) = 30;
    }=1;
}= 0x0607B1;
```

On the PC that controls the instrument, you need to set up an RPC server that makes this function available. How do you create the RPC server program? Again, rpcgen will do most of the work for you. It generates the initialization and dispatch routine required (see the _svc.c file). All you need to do is implement the actual `device_intr_srq()` function, which is called by the RPC dispatcher. Figure 10 shows a key fragment of the code generated by rpcgen.

Note that in the main function, the RPC server is registered for both TCP and UDP transport protocols. Support for UDP is optional in this case, but it won't hurt to register the dispatch routine for both protocols.

Figure 10. Interrupt channel server code generated by rpcgen

```
static void
device_intr_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        Device_SrqParms device_intr_srq_1_arg;
    } argument;
    char *result;
    xdrproc_t _xdr_argument, _xdr_result;
    char *(*local)(char *, struct svc_req *);
    switch (rqstp->rq_proc) {
    case NULLPROC:
        (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
        return;
    case device_intr_srq:
        _xdr_argument = (xdrproc_t) xdr_Device_SrqParms;
        _xdr_result = (xdrproc_t) xdr_void;
        local = (char *(*)(char *, struct svc_req *))
            device_intr_srq_1_svc;
        break;
    default:
        svcerr_noproc (transp);
        return;
    }
    memset ((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs (transp, (xdrproc_t) _xdr_argument,
        (caddr_t) &argument)) {
        svcerr_decode (transp);
        return;
    }
    result = (*local)((char *)&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp,
        (xdrproc_t) _xdr_result, result)) {
        svcerr_systemerr (transp);
    }
    if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument,
        (caddr_t) &argument)) {
        fprintf (stderr, "%s", "unable to free arguments");
        exit (1);
    }
    return;
}
```

Figure 10, continued.

```
int
main (int argc, char **argv)
{
    register SVCXPRT *transp;
    pmap_unset (DEVICE_INTR, DEVICE_INTR_VERSION);
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create udp service.");
        exit(1);
    }
    printf("UDP Socket for VXI-11 interrupt channel: %d\n",transp->xp_port);
    if (!svc_register(transp, DEVICE_INTR, DEVICE_INTR_VERSION,
        device_intr_1, IPPROTO_UDP)) {
        fprintf (stderr, "%s",
        "unable to register (DEVICE_INTR, DEVICE_INTR_VERSION, udp).");
        exit(1);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create tcp service.");
        exit(1);
    }
    printf("TCP Socket for VXI-11 interrupt channel: %d\n",transp->xp_port);
    if(!svc_register(transp, DEVICE_INTR, DEVICE_INTR_VERSION,
        device_intr_1, IPPROTO_TCP)) {
        fprintf (stderr, "%s",
        "unable to register (DEVICE_INTR, DEVICE_INTR_VERSION, tcp).");
        exit(1);
    }
    svc_run ();
    fprintf (stderr, "%s", "svc_run returned");
    exit (1);
    /* NOTREACHED */
}
```

The `printf()` statements were added to the code generated by rpcgen. They demonstrate how to get the port numbers allocated to the RPC server. You will need this information below.

`svc_run()` starts the RPC server. This call will not return, normally. Therefore, it is typically run as a separate process or thread.

The dispatch routine, `device_intr_1()`, dispatches incoming RPC calls to the local function implementations. In this case, there is only a single function to implement, `device_intr_srq_1_svc()`. A very basic example is shown in Figure 11. Typically, signals or other means of inter-process communication would be used to pass the event on to the main application program.

Once the RPC server has been put in place, SRQs can be activated in the instrument using the VXI-11 calls that are part of the regular core channel. See Table 4.

Summary

The VXI-11 protocol is used by LAN-to-GPIB gateways and by many native LAN-based instruments. Because it is based on RPC, a standard TCP/IP protocol, VXI-11 is supported by all flavors and versions of Linux. Although RPC is more complex than a simple TCP link, programming is still relatively straightforward with the help of rpcgen.

Figure 11. A trivial implementation of a device_intr_srq service routine

```
#include "vxi11intr_xdr.c"
void * device_intr_srq_1_svc(Device_SrqParms *MyDevice_SrqParms, struct
svc_req *Mysvc_req)
{
   printf("SRQ received...\n");
   return(NULL);
}
```

Table 4: VXI-11 functions for SRQs

VXI-11 function	Description
`create_intr_chan()`	Tells the instrument about the RPC server available for SRQs. Requests the instrument to establish an interrupt channel Parameters include the IP address and the port number of the RPC server put in place by the controller.
`device_enable_srq()`	Enables or disables the use of the interrupt channel.
`destroy_intr_chan()`	Requests the instrument to release the interrupt channel.

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**

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**

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 8, 2007
5989-6716EN

Agilent Technologies