VXI-11 and HP E2050A

A Programmer's Guide

B. Franksen
BESSY GmbH
Lentzeallee 100
14195 Berlin
Germany

Table of Contents

1 Introduction

1.1 Scope

This document contains additional information about a standard, specified by the VXIbus Consortium, called

- 'TCP/IP Instrument Protocol and Interface Mapping Specifications'

and how this specification can be used to implement simple access to GPIB devices via the LAN-GPIB gateway manufactured by Hewlett-Packard, called

- HP E2050A.

It is planned to use this gateway in the BESSY II control system wherever GPIB devices need to be accessed. The driver and device support for EPICS will be discussed in another document (‘LanGpib Driver Support for HP E2050A’).

Literature

In no way should the reader expect this document to give complete information about the above mentioned items. It is rather meant as an additional exposure and to gain complete understanding without the lecture of the original VXI documents (see next section for a reference) is not possible. Also, installation of the HP gateway requires reading the ‘HP E2050 LAN/HP-IB Gateway, Installation and Configuration Guide’. A good introduction into RPC programming as well as a reference for the standard RPC libraries is John Blumer’s ‘Power Programming with RPC’, published 1992 by O’Reilly & Associates.

1.2 The VXI-11 Standard

The VXI-11 standard specifies a protocol for communication with (test or measurement) devices over a network (LAN) via a so called *network instrument server* (we will abbreviate ‘network instrument’ with ‘NI’). This protocol uses the ONC/RPC (Open Network Computing/Remote Procedure Call) standard which in turn is based on TCP/IP and is described in a document named ‘VXI-11: TCP/IP Instrument Protocol Specification’.

A *network instrument server* is (in the above document) defined as an interface between the LAN on the one hand, and one or more *network instrument devices* on the other hand. If the connection between the devices and the server is a GPIB (IEEE488) connection, then the behavior of the network instrument servers is defined more specifically. This is done in ‘VXI-11.2: TCP/IP-IEEE488.1 Interface Specification’. No other part of VXI-11 will be of interest here.

1.3 The HP E2050A LAN-GPIB Gateway

The Hewlett-Packard HP E2050A is a gateway between a local area network (LAN) and a GPIB network. On the GPIB side it acts as the bus controller, whereas on the LAN side it acts as an RPC (Remote Procedure Call) server and claims to conform to VXI-11. Besides its GPIB connector it has an interface to RS232. On the LAN side it has a BNC and a 10 Base-T connection. It needs an external power supply.

2 Some Remarks on VXI-11

2.1 General Observations

Because the VXI-11 specification (freely available by the VXIbus Consortium) contains an RPCL (RPC Language) description of the protocol, it is possible to use a *protocol generator* (rpcgen) to generate client and server stubs (in C) which makes application development much easier. Unfortunately parts of the generated code are not reentrant, and so are not usable under VxWorks.

Anyway, I wrote a demo program (vxidemo) that runs under HP-UX and uses the generated code without any changes. The main reason for this was to test the VXI-11 conformance of the HP E2050A. It can now be used as a programming example for people who want to write applications that directly communicate with such an interface.

2.2 Channels

Channels are a feature of ONC/RPC: an RPC server implements a set of RPC protocols (in RPC language they are called programs) and these can exist in different versions. If a client and a server agree on a certain program in a certain version, they as a result establish a so called *channel*. This channel then allows the client to request of the server the execution of channel-specific set of remote procedure calls. Programs and versions are identified by a unique number.

The VXI-11 standard states that a network instrument host has to implement a certain set of RPC functions on three different channels. These are the

- Core Channel: send commands from the NI client to the NI server,
- Abort Channel: abort from a previously sent (core channel) command,
- Intr Channel: send an interrupt from the NI server to the NI client.

All three channels have a registered unique program number.

2.3 Core Channel

The core channels is used to establish so called links to devices on the bus and to give commands to them like read or write.

Links

According to VXI-11, a NI server supplies an abstract view of the connected devices by giving the client the possibility to create and destroy so called *links* to these devices. The *create_link* call takes an identifying string as parameter and returnes, if successful, a handle (of type *Device_Link*), which is to be used in following calls like *device_read* or *device_write*. All addressing of the devices is hidden behind this handle. Additionally one may call *create_link* with an identification string of the NI server itself (no special device). The resulting link can then be used to perform operations that affect all connected devices or change some state inside the server.

In the special situation of the GPIB server HP E2050A, the identifying string is “hpib” for the server and “hpib,<n>” for devices, where <n> is to be the GPIB address of the device. Some calls like *device_docmd* work only with the server link, and not with devices. For others like *device_read* and *device_write* it is the other way around.

Locks

Since the NI server resides in a network it is possible for more than one client to establish a connection to the server.

To avoid concurrent access to devices, links can be locked by the client. The device_lock and device_unlock calls are used for this purpose. It is possible to specify a lock_timeout in most core channel calls to wait a certain number of milliseconds for a lock to be released.

2.4 Abort and Intr Channel

The intr and abort channels both support only one call (*device_intr_srq* and *device_abort*). With the *device_abort* call the client can abort a previously given (core channel) call. The intr channel is used to implement GPIB service requests (SRQ). The intr channel effectively reverses the rolls of NI server (becoming an RPC client)

and the NI client (becoming an RPC server). If the client supplies an SRQ server, it must ask the NI server to create an intr channel by calling the (core channel) call *create_intr_chan*, giving his IP address, the port number, and the RPC program number and version.

SRQ Handling

If the intr channel server (on the NI client host) is called by the NI server it is not known which device has asserted the SRQ line. This information must be retrieved by the client program using the *device_readstb* (read status byte) call. Every device on the bus must be sent this call (a process called 'serial polling' in GPIB terminology). The returned status byte gives information about the state of the devices including the information 'has this device asserted the SRQ line?' (yes if bit 6 is set).

3 Using the Generated Protocol

3.1 General

The UNIX utility program rpcgen creates a number of files if given a protocol description written in RPCL (RPC Language, based on C). For every file xxx.rpcl it generates (see also corresponding manual page)

- a header file xxx.h, containing the type definitions, the function numbers and the prototypes for the generated functions
- a C source file xxx_clnt.c containing client side functions
- a C source file xxx_svc.c containing server side code and
- a C source file xxx_xdr.c containing data type conversion functions for the data types declared in xxx.rpcl.

The VXI-11 RPCL protocol description consists of two files: vxi11core.rpcl (containing core and abort channel) and vxi11intr.rpcl (containing the intr channel). Of course, not all of the generated files are needed to implement the protocol completely, bu only the client side of the core/abort channel and the server side of the intr channel. Furthermore, the abort and intr channel are optional in a NI client.

To illustrate how rpcgen transforms an RPCL declaration into C source code, consider the following simple example:

Example RPCL -> C

This is the RPCL declaration of a function taking two integers as parameter and returning two floats together with the type declaration of its parameters and return values (file simple.rpcl):

```
struct Parms
{
    int p1;
    int p2;
};
struct Resp
{
    float a;
    float b;
};
program SIMPLE
{
    version SIMPLE_VERSION
    {
        Resp simple_func (Parms) = 1;
    } = 1;
} = 0x0607AF;
```

This is transformed by rpcgen to the C declarations (simple.h):

```
#include <rpc/types.h>

struct Parms {
    int p1;
    int p2;
};
typedef struct Parms Parms;
bool_t xdr_Parms();

struct Resp {
    float a;
    float b;
};
typedef struct Resp Resp;
bool_t xdr_Resp();

#define SIMPLE ((u_long)0x444)
#define SIMPLE_VERSION ((u_long)1)
#define simple_func ((u_long)1)
extern Resp *simple_func_1();
```

The last declared function is the one the user is actually going to call. NOTE: the generated function prototypes do not contain parameter declarations. If you want to be shure, read the corresponding `xxx_clnt.c` file. The general rule is: first argument is a pointer to the (user allocated) parameter; second argument is the CLIENT handle as obtained from `clnt_create` (declared in rpc/rpc.h).

Client Code

The client side call of the remote procedure could look like this:

```
    Parms  parms;
    Resp   *resp;

    /* Fill parms structure */
    parms.p1 = 1;
    parms.p2 = 2;

    /* Do the call; rpcClient is the RPC CLIENT handle */
    resp = simple_func_1(&parms, rpcClient);

    /* Evaluate result */
    if (resp == NULL)
    {
        clnt_perror(rpcClient, “<name of the server>”);
        return;
    }
    /* resp now points to a (static) structure holding the
       result values
    */
```

Server Code

The server side only has to define the function:

```
Resp *simple_func_1(Parms *parms)
{
    ...
}
```

The file `simple_svc.c` contains a main function that starts a server routine automatically which int turn calls `simple_func_1` as soon as a request arrives from a client.

3.2 Under UNIX

The protocol generator rpcgen is a UNIX tool and one can expect that the generated source code is directly usable under UNIX. We present the main steps here. Many of the things presented here remain the same under VxWorks. The differences are stated in the next section.

Include Files

The following header files must be included:

```
#include <rpc/rpc.h>

#include “vxi11core.h”
#include “vxi11.h”
```

Open RPC Connection

To start, we must open an RPC connection. The client handle `rpcClient` is used in all subsequent RPCs.

```
CLIENT       *rpcClient;
static char  *svName = “box10.acc”;

/* open rpc connection */
rpcClient = clnt_create(svName,DEVICE_CORE,
    DEVICE_CORE_VERSION, “tcp”);
if (rpcClient == NULL)
{
    clnt_pcreateerror(svName);
    return 1;
}
```

Create Device Link

Before sending any data to our device we must create a link:

```
/* fill Create_LinkParms structure */
crlp.clientId = rpcClient;
crlp.lockDevice = 0;
crlp.lock_timeout = 10000;
crlp.device = “hpib,28”;

/* call create_link on instrument server */
crlr = create_link_1(&crlp, rpcClient);
if (crlr == NULL)
{
    clnt_perror(rpcClient, svName);
    return 1;
};
```

Example: send string

To send a string (e.g. a command) to our device we do the following:

```
Device_WriteParms  dwrp;
Device_WriteResp   *dwrr;

dwrp.lid = crlr->lid;
dwrp.io_timeout = IO_TIMEOUT;
dwrp.lock_timeout = LOCK_TIMEOUT;
dwrp.flags = VXI_ENDW;
dwrp.data.data_len = strlen(dval);
dwrp.data.data_val = dval;

dwrr = device_write_1(&dwrp, cl);
...
/* error test here like above */
```

The principle should be clear now.

Destroy Device Link

If the link to the device is no longer used destroy it:

```
Device_Error *derr;
```

```
derr = destroy_link_1(&(crlr->lid), rpcClient);
```

Close RPC Connection

At the and of the program the client handle must be destroyed.

```
clnt_destroy(rpcClient);
```

3.3 Under VxWorks

As mentioned in the introduction, the source code generated by rpcgen partly uses static variables (e.g. for the returned structures) and is therefore not reentrant. Of interest are the following generated files:

- vxi11core_clnt.c: client side functions for the core and abort channel
- vxi11intr_svc.c: server side functions for the intr channel

The functions provided in these files cannot be used under VxWorks, at least not as library functions (which may be used by different tasks). They may be used on a UNIX workstation but only when statically linked to the program. This is the way in which vxidemo uses these functions.

Under VxWorks, we must use the rpc library functions directly. On the client side this looks like:

Include Files

```
#include <rpcLib.h>
#include <rpc/rpc.h>
```

RPC Task Init

```
/* every task that wants to use RPC must call rpcTaskInit */
if(rpcTaskInit() != OK)
{
    logMsg("Can't init RPC for this task\n");
    return ERROR;
}
...
struct timeval rpcTimeout = {10, 0};
/* timeout for rpc calls (10 sec) */

CLIENT *rpcClient;
/* link creation/destruction is identical to UNIX */
...
    Create_LinkParms crLinkP;
    Create_LinkResp  crLinkR;
    enum clnt_stat   clntStat;
```

Example: Create a Link

```
    crLinkP.clientId = rpcClient;
    crLinkP.lockDevice = 0;
    crLinkP.lock_timeout = 10000;
    crLinkP.device = "hpib,28";

    /* initialize crLinkR */
    bzero((char*)&crLinkR, sizeof(Create_LinkResp));

    /* call create_link on instrument server */
    clntStat = clnt_call(rpcClient, create_link,
        xdr_Create_LinkParms, &crLinkP,
        xdr_Create_LinkResp, &crLinkR, rpcTimeout);
    if(clntStat != RPC_SUCCESS)
    {
        printf("RPC error");
        return ERROR;
    }
    if(crLinkR.error != 0)
    {
```

```
        printf("VXI error");
        return ERROR;
    }
```

Remarks:

- The generated XDR functions can and should be used without change.
- Following changes must/may be made in the generated header files:
  - change the included file `<rpc/types.h>` to `<rpc/rpctypes.h>` and
  - delete the `xxx_1` function prototypes.
- Error messages in the above examples should be more informative.

So far we have considered only client code. To write a server under VxWorks is more difficult.

4 The Example Program *vxidemo*

Vxidemo is a very simple UNIX command line program for accessing GPIB devices via the HP E2050A. It accepts exactly one parameter which is the so called *device name* resp. *interface name*. The syntax is:

Syntax

vxidemo <interfacename>[,<address>]

where <interfacename> is any name you have programmed into the gateway, by default 'hpib'. If the optional ',<address>' is omitted you are directly connected with the interface. This enables you to send commands like, for example, IFC (InterFace Clear) and even to control single lines (like ATN or REM).

If you supply the ',<address>' (no whitespace between the arguments) then you are connected with the device on the bus that has the GPIB address <address>. Every command you are sending now goes to the specified device.

After startup the program shows you a few lines like this:

```
create_link result  = 0 (no error)
device link         = 537382552
abort port          = 897
max receive size    = 32768
```

This tells you if a link to the interface or device was successfully created or not. (The other three values are returned by the create_link RPC and not very interesting).

Main Menu

Thereafter (and every time a command is executed) you see the following 'menu':

```
s   send string
a   send string and receive answer
r   receive only
c   do special command
v   clear device
b   read status byte
t   trigger device
le  go to local
ld  local lockout
dl  device lock
du  device unlock
qe  enable srq's
qd  disable srq's
pe  serial poll enable
pd  serial poll disable
x   exit
```

Some of these commands need arguments. You are prompted for them after you have typed in the command. When you have given the arguments, the command is executed and you get a message like

Response

```
device_local result = 8 (operation not supported)
```

or

```
device_docmd result = 0 (no error)
```

which tells you

1. which RPC has been called and
2. the error status of the command.

If the command was a query for some information from the device or interface, the result is shown in the next line.

Remarks

It should be noted that vxidemo only uses the core channel (and not the abort and intr channel) of the VXI-11 protocol.

Again, without knowledge of the VXI papers some things about this program may seem very mysterious.