

VXI-11 RPC Programming Guide for the 8065 An Introduction to RPC Programming

INTRODUCTION

ICS's new 8065 Ethernet-to-GPIB Controller is a VXI-11 compatible device. The 8065 can be programmed with calls to a VXI-11 compliant VISA¹, with Agilent's SICL library or with VXI-11 RPC commands. This Application Note describes what RPC commands are and shows how RPC calls can be used with ICS's 8065 to control GPIB instruments.

VXI-11 INTRODUCTION

The VXI-11 Specification specifies a protocol for communication with (test or measurement) devices over a network (LAN) via TCP/IP. This protocol uses the ONC/RPC (Open Network Computing/Remote Procedure Call) standard which is based on TCP/IP and is described in the 'VXI-11: TCP/IP Instrument Protocol Specification'.

VXI-11 is the overall VXI-11 document and describes the network protocol. There are three sub-specifications. VXI-11.1 is for a VXI chassis and is not applicable to the 8065. The VXI-11.2 Specification defines how the interface (8065) controls the GPIB bus and performs bus specific functions. The VXI-11.3 Specification describes instrument specific functions including data transfer to and from an instrument.

ICS's 8065 has both VXI-11.2 and VXI-11.3 capability so it can be used just as you would any 488.2 Controller. Generating IFCs and performing 488.2 protocols such as FindLstn are no problem for the 8065. Competitive units with only VXI-11.3 capability are limited to just device specific functions.

RPC

RPC stands for Remote Procedure Calls. An RPC server, like the 8065, implements a desired set of RPC protocols to carry out specific functions. 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. RPC calls are the requests to the server to execute one of the functions.

The programmer has or obtains a RPC package for his system that includes a protocol converter (rpcgen). Because the VXI-11 Specification contains an RPCL (RPC Language) description of the protocol, it is possible to use the protocol generator (rpcgen) to generate client and server stubs for C language programs which makes application development much easier.

For more information on RPC, the readers may want to look at the references at the end of this Application Note.

CHANNELS

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 Network interface client to the network interface server,
- Abort Channel: abort a previously sent (core channel) command,
- Intr Channel: send an interrupt from the network interface (8065 as a client) to the Application (as a server).

CORE CHANNEL

The core channel is used to establish links to the 8065 and to the devices on the bus. The links can be created and destroyed. The create_link call takes an identifying string as parameter and returns, 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 network interface server itself (no special device). The resulting link can then be used to perform GPIB bus operations or modify the server's operation.

Notes: 1. Some VISA libraries like those from National Instruments do not provide interface level commands and limit the 8065 to just VXI-11.3 functions.

In the special situation of the GPIB server (8065), the identifying string is "gpib0" for the server and "gpib0,<n>" for devices, where <n> is the GPIB address of the device. Some calls like device_docmd only work with the server link, and not with device links. Other commands like device_read and device_write are intended for device links.

Since the network interface 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. Because a one client might not be aware of the existence of another client, it is possible to specify a lock_timeout in the calls so the calls wait a certain number of milliseconds for an existing lock to be released.

FLAGS FIELD

The Flags field is only used during actual Read/Write operations. Of the three fields, one is used for writing and one is used for reading, with the third flag (WaitLock) used for both operations.

The End flag is used when writing to an instrument. In the GPIB world, a write operation normally has the last character sent with the EOI signal asserted to indicate that the current character is the last character of the transmission. It is possible to write multiple blocks of data as a single write, if the End flag is zero on all but the last block. The last block of a write should always have the End flag set. If the device wants a terminating character, such as a linefeed (0x0A), it can be added as the last character in the message.

The TermChrSet is used when reading from an instrument. Most instruments use a terminating character (normally a linefeed, 0x0A) at the end of the data message to terminate the message. (Binary data is terminated by asserting EOI on the last character of the message) If TermChrSet is set to one, then the character will cause the 8065 to terminate the read and return all prior read data to the client. If it is set to zero, then only an EOI will be able to terminate a read. Note that the read length being met will always cause the read to terminate.

The WaitLock tells the 8065 to wait for up to lock_timeout amount of time if another client has the instrument locked. The simply ignore the WaitLock flag while you're learning how to communicate with the 8065.

ABORT AND INTERRUPT 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 that is still in progress. Some GPIB operations are completed immediately and the device_abort call will have no affect. An useful example is cancelling a read from a device that has no data.

The intr channel is used to implement GPIB service requests (SRQ). The intr channel effectively reverses the rolls of the network interface server and client. Here the 8065 becomes a client and the application becomes the server. To setup the intr channel the application supplies

an SRQ handler or RPC server, and uses the create_intr_chan call to ask the 8065 to create an intr channel.

To generate the interrupt, the client enables the device to generate the SRQ and gives it an id key. This also sets the 8065 to Auto Serial Poll. When the enabled instrument generates a SRQ, the 8065 serial polls the enabled device(s) and passes a message with the id key of the SRQ generator to the RPC service function in the PC (Application). The RPC service function receives but does not reply to the interrupt message. Instead a flag or other signal should be set so that the main routine in the client application will read the status byte from the instrument who generated the SRQ and service the instrument.

For more details on creating and using the intr channel, see Application Note AB80-4 'SRQ Handling with a VXI-11 Interrupt Channel on ICS's Model 8065 Ethernet-to-GPIB Controller'

SIMPLE PROGRAM STEPS

The following are the basic steps for writing a simple program that reads the IDN message from an instrument at address 28.

1. Initialize the RPC layer.
2. Using RPC, send the following VXI-11 commands.
 - a) create_link to "gpib0,28"
 - b) device_write "**IDN?" with the End flag set to one
 - c) device_read with TermChrSet set to zero
 - d) destroy_link
3. Close the RPC layer

USING THE GENERATED PROTOCOL

The following examples are for a UNIX operating system but apply to any operating system with an RPC package. While the examples are believed to be correct, the user assumes all responsibility for their operation in his system.

The utility program rpcgen creates a number of C language files from the RPCL protocol description. The files include:

- 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: vx11core.rpcl (containing core and abort channel) and vx11intr.rpcl (containing the intr channel). Not all of the generated files are needed unless you want to implement the complete protocol. The abort and intr channel are optional in a network interface client (Application). Note that the 8065 includes a complete core, abort and intr channel implementation for a network server.

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

EXAMPLE RPC -> 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 sure, 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 server routine automatically calls simple_func_1 as soon as the RPC request arrives from a client.

UNDER UNIX

The protocol generator, rpcgen, generates source code that is directly usable under the operating system. While this example is for UNIX, the example applies to any operating system with an RPC package. The basic steps are:

Include Files

The following header files must be included:

```
#include <rpc/rpc.h>
#include "vxllcore.h"
#include "vxll.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 device link. Here the device is at a GPIB primary address of 28. Use 'gpib0,28' as the device.

```
/* fill Create_LinkParms structure */
crlp.clientId = rpcClient;
crlp.lockDevice = 0;
crlp.lock_timeout = 10000;
crlp.device = "gpib0,28";

/* 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 (cl) 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 */.
```

Note that most GPIB devices use the linefeed terminating character to terminate a ASCII string. Newer IEEE-488.2 devices can also sense the EOI line when it is asserted on the last character of the string. Use the End flag when writing. Set the End Flag to 1. Use TermChrSet to set the terminating character. Specify the TermChr within the device_read command.

Destroy Device Link

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

```
Device_Error *derr;

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

Close RPC Connection

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

```
clnt_destroy(rpcClient);
```

PORTING TO OTHER SYSTEMS

The source code generated by rpcgen uses some static variables (e.g. for the returned structures) and is therefore not reentrant. The user has to take this into account when writing his application. Of interest are the following generated files:

- vxil1core_clnt.c: client side functions for the core and abort channel
- vxil1intr_svc.c: server side functions for the intr channel

Be especially aware of this limitation when the RPC code is called by a higher level library or application with multi-threaded capability. One such example is VxWorks.

SuSE EXAMPLE

The following is a complete Linux example for SuSE 10.0. The example creates links to an ICS 8065 then to a HP437B Power Meter at GPIB address 12.

```
#include "vxil1core.h"
#include <stdio.h>

#define WAITLOCK_FLAG 1
#define WRITE_END_CHAR 8
#define READ_END_CHAR 0x80
#define VXI_ENDW (WAITLOCK_FLAG | WRITE_END_CHAR)

CLIENT*rpcClient;
Create_LinkParms crlp;
Create_LinkResp *crlr;

Device_WriteParms dwrp;
Device_WriteResp *dwrr;

Device_ReadParms drdp;
Device_ReadResp *drdr;

Device_Error *derr;

static char *svName = "192.168.46.105";

int main(){
    char sendMessage[1024];
    char responseMessage[2048];
    rpcClient = clnt_create(svName, DEVICE_CORE, DEVICE_
CORE_VERSION, "tcp");
    if (rpcClient == NULL){
        printf("Error creating client\n");
        clnt_pcreateerror(svName);
        return 1;
    }
    printf("Hello %s\n", svName);
    crlp.clientId = rpcClient;
    crlp.lockDevice = 0;
    crlp.lock_timeout = 10000;
    crlp.device = "gpib0,12";
```

```

crlr = create_link_1(&crlr, rpcClient);
if (crlr == NULL){
    clnt_perror(rpcClient, svName);
    return 1;
}

printf("Link created to %s\n", crlr.device);

dwrp.lid = crlr->lid;
dwrp.io_timeout = 1000;
dwrp.lock_timeout = 10000;
dwrp.flags = VXI_ENDW;
sprintf(sendMessage, "**IDN?\n");
dwrp.data.data_len = strlen(sendMessage);
dwrp.data.data_val = sendMessage;

dwrr = device_write_1(&dwrp, rpcClient);
if (dwrr == NULL){
    clnt_perror(rpcClient, svName);
    return 1;
}
printf("device_write returns error code %d\n", dwrr->error);

drdp.lid = crlr->lid;
drdp.io_timeout = 1000;
drdp.lock_timeout = 10000;
drdp.flags = VXI_ENDW;
drdp.termChar = '\n';
drdp.requestSize = 1024;

drdr = device_read_1(&drdp, rpcClient);
if (drdr == NULL){
    clnt_perror(rpcClient, svName);
    return 1;
}
printf("device read returns error %d, reason %d.\n", drdr->error,
drdr->reason);
printf("response = %s\n", drdr->data.data_val);

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

clnt_destroy(rpcClient);
}

```

DEBUGGING HINTS

Lost device LIDs, lost data or overwritten LIDs may be caused by not saving the LID or data before calling the next RPC function. Some RPC implementations return a pointer to a LID or data buffer when executing a create_link or read call. The user must read and save the values in his own variables to keep them from being overwritten by a subsequent RPC call.

SUMMARY

This application note has described RPC calls and shown how the calls are generated by the rpcgen utility. A brief examples are provided so the reader can better understand how RPC calls are used with the 8065 to control GPIB devices.

This application note is only intended to introduce the reader to the RPC concept and guidelines for programming with the 8065. The following references are recommended reading before attempting RPC coding:

1. "Cisco IOS for S/390 RPC/XDR Programmer's Reference".
<http://www.cisco.com/univercd/cc/td/doc/product/software/ioss390/ios390rp/>
2. Halfway down, "Managing the Server"
http://publib16.boulder.ibm.com/pseries/en_US/aixprgpd/programc/rpc_ref.htm
3. Builds and registers a simple RPC service.
http://publib16.boulder.ibm.com/pseries/en_US/aixprgpd/programc/rpc_lowest_ex.htm#a287x92028
4. Similar to the second URL, but with more examples.
http://www.unet.univie.ac.at/aix/aixprgpd/programc/rpc_ref.htm

The reader is urged to use Google to search for an RPC example or programming information for his system since RPC implementations vary from system to system and an example that works on one system may not work on another system.

DISCLAIMER

While the examples herein are correct to the best of our knowledge, the user assumes all responsibility for their operation in his system. The authors are not responsible for the examples' operation.

ACKNOWLEDGEMENTS

The author wishes to thank B. Franksen at Bessy GmbH in Germany for his paper 'VXI-11 and HP E2050' Many of his ideas are presented in this Application Note.

Additional thanks to Bob Clemmons at Aeroflex for his patience and SuSE example.