ICS ELECTRONICS
division of Systems West Inc.

IEEE 488 APPLICATION BULLETIN

CODING FOR REVERSE CHANNEL SERVICE REQUESTS
From VXI-11.3 Compatible Instruments

INTRODUCTION

Getting a Service Request from a VXI-11.3 compatible instrument is sometimes needed in an application program to signal that the instrument needs help, that it has a problem with a particular command or that it has completed a long term task. Adding the code to handle a Service Request is often difficult since there are so very few examples around and because the Reverse Channel usage is so very different from the normal core channel.

Also, while the VXI-11 Specification discusses the Reverse Interrupt Channel at several places in the Specification, it does not give the reader a good comprehensive description of the Reverse Channel operation. This Application Note is intended to remedy that omission by giving the user the information necessary to implement a Reverse Interrupt Channel and an working C language example.

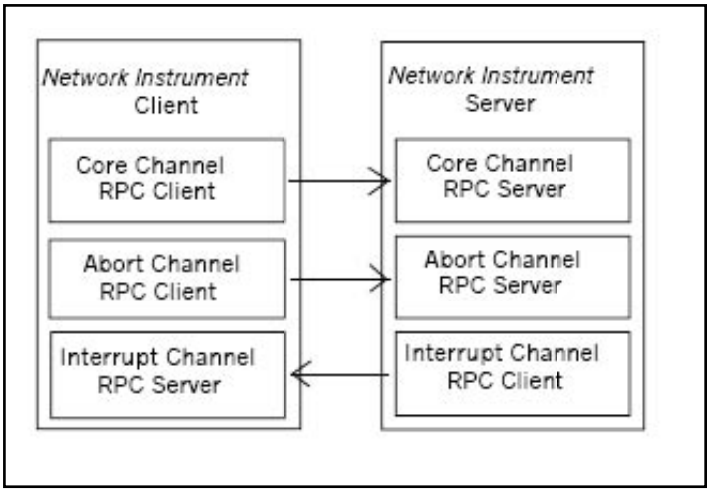

Figure 1 VXI-11 Client-Server Channels

REVERSE CHANNEL CONCEPT

When the client application links to a VXI-11.3 instrument, it typically creates one or two channels. The first channel is a bi-directional Core Channel for transmitting data and commands between the client and the instrument. The second channel is the Abort Channel which is used by the client to abort a command previously sent to the instrument.

Only the Core Channel is required. The Abort and Reverse Interrupt Channel are optional.

This application note deals with the third channel which is an optional Reverse Interrupt Channel that can be created to handle the Service Requests from the instrument. The Interrupt Channel is unique in that the instrument is the client and the user's application program is the server. The three channels are shown in Figure 1.

The concept of RPC communication is that the originator of the message is considered the RPC Client, with the receiver of the message being defined as the RPC Server. Since the SRQ notification must originate from the Instrument (Server), this defines the Instrument (Server) as being the RPC Client when sending the SRQ notification message.

VXI-11 CAPABLE RPC SUPPORT LIBRARY

The method used in the example program requires a VXI-11 capable RPC support library. The example was coded with functions from ICS's internal test library to initiate RPC calls. The user will have to substitute equivalent functions from his selected RPC support library. Companies such as Nebulla LLC (http://www.netbula.com/oncrpc/) or Distinct Corporation (http://www.distinct.com/) provide ONC RPC support libraries for WIN 32 applications. Their libraries will have equivalent functions to those used in the example program.

Note: - ICS has not tested the Nebulla or Distinct ONC libraries and does not endorse them over any other ONC libraries.

One thing to consider when selecting an ONC library or using shareware is the open source disclosure requirements. Do you have to or want to disclose your company's code?

EXAMPLE PROGRAM CONCEPT

The Example Program runs on the client computer and takes advantage of a VXI-11 compatible instrument's ability to generate a Service Request when it receives a bad or incorrect command. All VXI-11.3 instruments must be IEEE-488.2 compliant per the VXI-11 Specification. Figure 2 shows the minimum Status Reporting Structure required in a IEEE-488.2 compatible instrument.

Caution: Be careful with LXI instruments. The LXI Specification does not require LXI instruments to be IEEE-488.2 compliant so check the manufacturer's specifications before running this example program with any LXI instruments.

The Status Reporting Structure works in the following manner. When the instrument receives a bad or illegal command, it sets bit 5 in the Standard Event Status Register (ESR Register). Setting the corresponding bit in the Event Status Enable Register allows the bit status cascade down and set bit 5 in the Status Byte Register. Setting the corresponding bit in the Status Byte Enable Register will let a set bit generate a Service Request (SRQ) by setting bit 6.

The example program has three parts: a Callback Routine, an IDN Query Routine and the Main Section. The Callback Routine sets the callback flag when called. The IDN Query Routine reads the instrument's IDN message when called.

The Main.c gets the target's (instrument's) IP address and creates a link to it. It then performs and IDN query to be sure it has communication with the correct instrument. The Status Byte is cleared and the Reverse Channel is created. Next an '*ESE 32' command is sent to enable bit 5 in the ESR Register and an '*SRE 32' command is sent to enable bit 5 in the Status Byte Register. After the setup is complete, the thread goes into a long loop.

A bad command is then written out to the instrument to create the service request. The callback function in a separate thread sets the callback flag when it recognizes the key word "Hello".

When the Main program sees that the Callback Flag is set, it reads the Status Byte to knock down the SRQ bit (bit 6) and then reads the ESR Register to clear the bit 5. The second read of the Status Byte confirms that the Status Byte is now 0. Any other steps needed to service the Service Request could be inserted at this point.

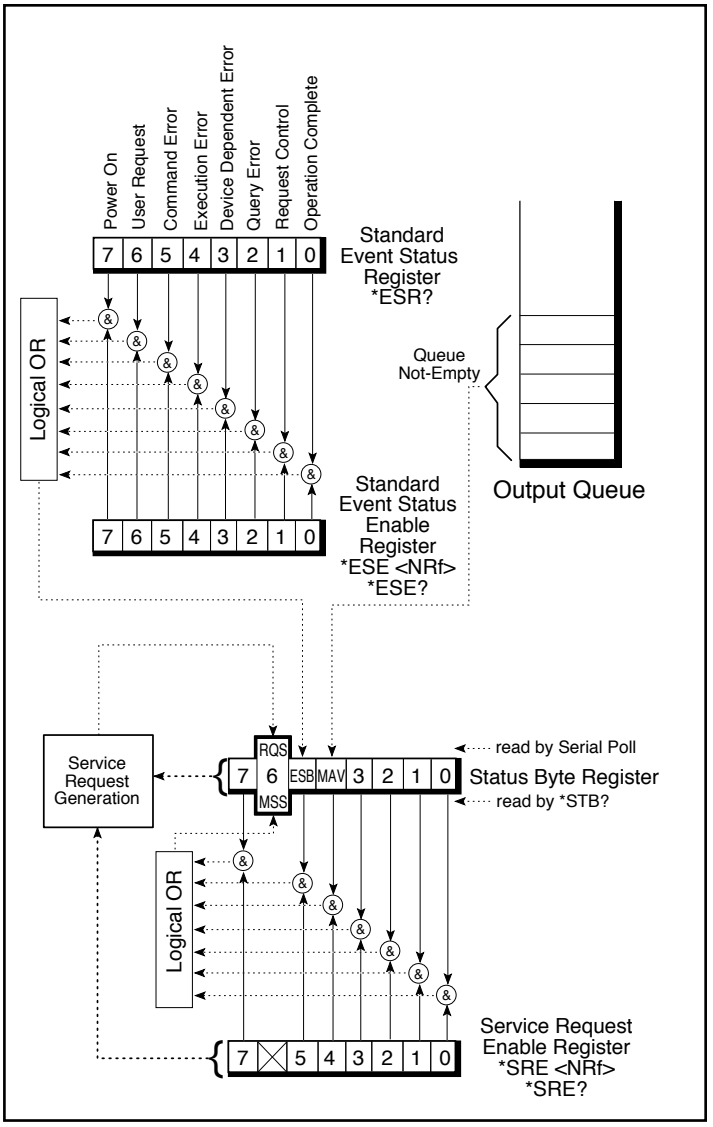

Figure 2 IEEE-488.2 Required Status Reporting Structure

ADAPTING THE EXAMPLE TO YOUR PROGRAM

First find and adopt a RPC Support Library. Then determine what will be the cause of the Service Request and what actions you need to take as a result of the Service Request. Examine the instrument's Status reporting Structure and decide what bits need to be enabled to generate the Service Request. Figure 2 shows the simple minimal required Status Structure. While many instruments have expanded Status Reporting Structures with many potential ways to generate Service Requests (SRQs), the concepts in the example program still apply. Be sure to replace the calls to ICS's internal library (light gray boxes in the example listing) with calls to your library.

EXAMPLE REVERSE CHANNEL PROGRAM

INCLUDES

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>
#include <conio.h>
#include <time.h>

#include "gpib.h"

int    volatile    callbackFlag;
```

CALLBACK SUBROUTINE

```
void  __stdcall  callback (char key[], int keyLen)
   {
   char    buf[1024];

   memcpy (buf, key, keyLen);
   buf[keyLen] = '\0';

   printf ("Key: [%s]\r", buf);

   callbackFlag = 1;
   }
```

IDN QUERY SUBROUTINE

```
idnQuery (int link)
   {
   int     j;
   char    buf[1024];

   if (device_write (link, 0x04, "*IDN?\n"))
      {
      printf ("Error performing *IDN query write\n");
      exit (1);
      }

   j = sizeof(buf);
   if (device_read (link, 0x08, buf, &j, 0x00))
      {
      printf ("Error performing *IDN query read\n");
      exit (1);
      }
   }
```

MAIN.C

```
int  main (int argc, char *argv[])

   {
   int     i, link, loop;
   short   s;

   if (argc != 2)
      {
      printf ("Please provide target IP\n");
      exit (1);
      }

   if (!initEth488 (argv[1]))
      {
        printf ("Error: Unable to perform initEth488 at IP %s\n",
argv[2]);
      exit (1);
      }

   // Create a link to the device. P=0 & S=0 means inst0
   i = create_link (argv[2], 0, 0, 0, &link);

   if (i)
      {
      printf ("Error %d on create_link for device\n", i);
      exit (1);
      }

   idnQuery (link);

   // Flush the status byte to be certain it is empty.
   device_readstb (link, &s);

   create_intr_chan ();
   device_enable_srq (link, "Hello", 5, callback);

   // Initialize the unit to detect an Command Error and generate
   // a reverse channel (SRQ) message.
   device_write (link, 0x04, "*ESE 32\n");
   device_write (link, 0x04, "*SRE 32\n");

   for (loop= 0; loop < 100000; ++loop)
      {
      printf ("Loop: %d ... ", loop);
      callbackFlag = 0;


      // Now generate a Command Error by sending the instrument
an obviously illegal command (*EEE).
       // This should cause a reverse channel message to be gener-
ated.
      device_write (link, 0x04, "*EEE\n");
```

Note: Gray boxes indicate calls to ICS's internal library. Replace them with calls to your RPC Support Library

```
    // Now wait for up to 10 seconds for a reverse channel mes-
sage to be seen.
    for (i= 0; i < 200; ++i)
      {
      if (callbackFlag)
        {
        if (device_readstb (link, &s))
          {
          printf ("Error executing device_readstb\n");
          exit (1);
          }
        else
          {
          if (device_write (link, 0x04, "*ESR?\n"))
            {
            printf ("Error writing the *ESR query\n");
            exit (1);
            }
          else
            {
            char    buf[1024];
            int     j= sizeof(buf);

            if (device_read (link, 0x08, buf, &j, 0x00))
              {
              printf ("Error reading *ESR query response\n");
              exit (1);
              }

            if (device_readstb (link, &s))
              {
               printf ("Error executing second device_readstb\
n");
              exit (1);
              }
            }
          }

        idnQuery (link);

        // Terminate the loop and exit
        break;
        }
      else
        {
        // Pause 100ms
        Sleep (100);
        }
      }

    if (!callbackFlag)
      {
      printf ("No background interrupt detected\n");
      exit (1);
      }
    }
```

```
  disconnectEth488 ();

  return (0);
  }
```

SUMMARY

This Application Note has described the major characteristics of a VXI-11 Reverse Interrupt Channel and how an example program works.

The Application Note also includes an C language example that sets up a reverse channel, enables the instrument to generate a Service Request and then handles the Service Request.

References:

The following references were used in the course of developing this program:

* VXI-11 RPC Programming Guide for the 8065: An Introduction to RPC Programming, Application Note AB80-3, ICS Electronics.

* VMEbus Extensions for Instrumentation: TCP/IP Instrument Protocol Specification VXI-11, Revision 1.0, July 17, 1995.

* Power Programming with RPC, John Bloomer, O'Reilly & Associates, Inc., 1992.