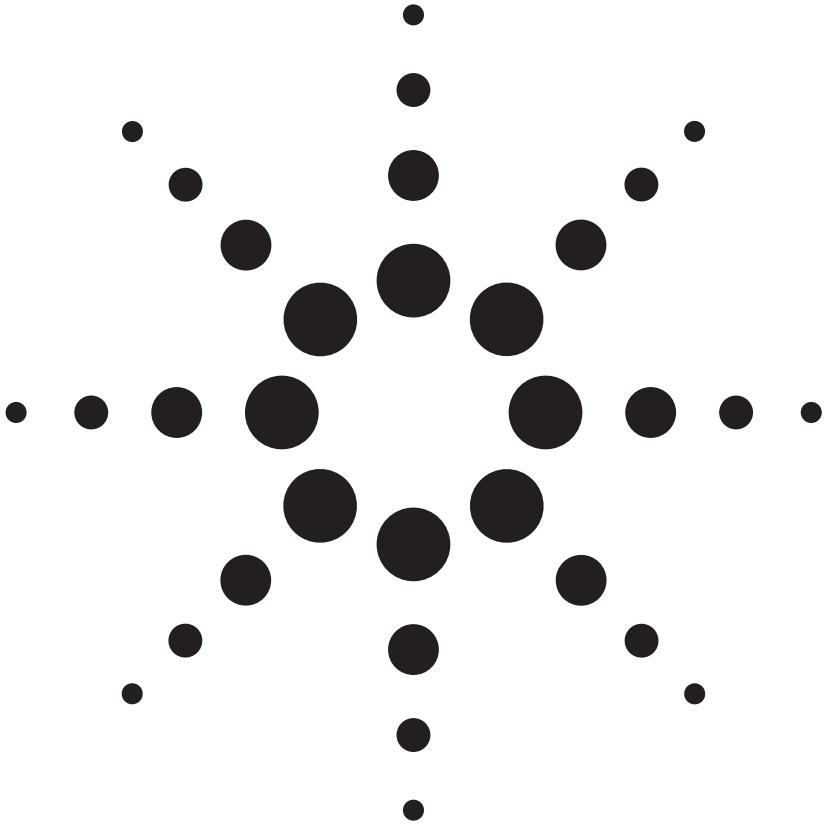
# Linux を使用した LXI 測定器の制御：VXI-11 の使用

Application Note 1465-28



Agilent Open では、I/O インタフェースとして、PC 標準の I/O インタフェースを採用しています。これにより、ハードウェア、I/O、ソフトウェア・ツールを柔軟に組み合わてシステムの構築、拡張、保守が可能になります。たとえば、OS として Linux を使用している場合、LAN や USB インタフェースを有効活用できます。本アプリケーション・ノートでは、Linux 環境でテスト機器を制御する方法を解説しています。サンプル・コードは、http://www.agilent.co.jp/find/linux からダウンロードできます。

## 目次



Agilent Technologies

## LXI 測定器と LAN ベースの測定器

Agilent は長年にわたって、LAN インタフェースを備えた測定器を提供してきました。2004 年の LXI Consortium[1] の発足とともに、LAN ベースの測定器は急速に普及し始め、テスト業界に広く受け入れられるようになりました。

イーサネットには、コストの安さや、分散／リモート・アプリケーションへの適合性などの、いくつかの明白な利点があります。これほど明白ではないにしても、同程度に重要ないくつかの機能もあります。例えば、ギガビット・イーサネットのきわめて高い性能や、マルチキャスト（1 対多）、ピアツーピア、準同時通信によって実現される柔軟性などです。

イーサネットへの移行は、Linux（およびその他の非 Windows）ユーザにとって大きな利点があります。オペレーティング・システムに内蔵された標準 API を使って測定器を制御できるからです。GPIB や MXI（テスト業界専用）、あるいは PCI カードなどのインタフェースには、使用するオペレーティング・システムのフレーバに対応した特殊なドライバ・ソフトウェアが必要であり、そのようなソフトウェアが使用できない場合もあります。

## 測定器制御に用いられる TCP/IP プロトコル

2000 年に、VXI*plug&play* Alliance[2] は、LAN ベースの測定器のサポートを VISA 仕様に追加しました。イーサネットによる測定器制御の 2 つの方法が VISA に採用されました。1 つは VXI-11[3] で、もう 1 つはダイレクト TCP ソケット通信です（図 1 を参照）。

VXI-11 は、もともと GPIB の機能をシミュレートするために設計されました。これには、サービス・リクエスト（SRQ）、シリアル・ポール、デバイス・トリガ、デバイス・クリアなどの、ハードウェアに基づいた機能も含まれています。ネイティブ LAN ベースの測定器以前には、LAN-GPIB ゲートウェイで使用されていました。VXI-11 は、リモート・プロシージャ・コール（RPC）に基づいたものです。LAN-GPIB ゲートウェイなどのサーバにより、ゲートウェイにつながれた GPIB 測定器など複数の論理デバイスへのアクセスを実現できます。VXI-11 は LAN-GPIB ゲートウェイ用に設計されたものですが、ネイティブ LAN ベースの測定器の多くでも互換性のためにサポートされています。

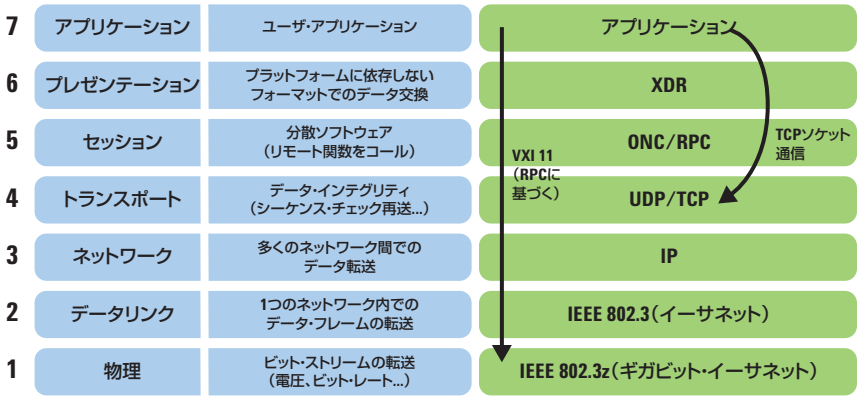
測定器制御のもう 1 つの方法は、ソケット通信です。これは、ダイレクト TCP ソケット接続経由で、ストリーム方式で測定器を制御するものです。ディスク・ファイルの読み書きに似た方法です。このタイプの接続の詳細については、Application Note 1465-29「Linux を使用した LXI 測定器の制御：TCP の使用」で詳細に説明しています。

## VXI-11 と TCP ソケット：どちらを使用すべきか？

Agilent E5810A などの LAN-GPIB ゲートウェイで GPIB 測定器にアクセスする場合、PC をゲートウェイとして使用する場合、使用できるのは VXI-11 だけです。一方、ネイティブ LAN 測定器の多くは、TCP VXI-11 とソケット通信の両方をサポートしています。どちらを使用する方がよいのでしょうか？

多くの場合は、単に好みの問題です。ただし、VXI-11 の方が複雑な（上位レイヤの）プロトコルです（図 1 を参照）。したがって、ダイレクト・ソケット通信の方が多くの場合高い性能が得られます。特に、実際の測定時間が短く、個別のトランザクションを数多く実行する場合にはダイレクト・ソケット通信のほうが適しています。

図 1.　TCP/IP の各層と測定器制御での使用



## Agilent IO ライブラリと LAN サーバ

Agilent IO ライブラリ・スイートを使用すると、Windows PC を LAN-GPIB ゲートウェイとして使用できます。このソフトウェアには VXI-11 サーバが含まれ、これを使って PC のローカル・インタフェースをネットワークから（したがって Linux コントローラから）アクセスできます。

Agilent IO ライブラリの詳細については、**http://www.agilent.co.jp/find/iosuite** をご覧ください。

## VXI-11 の基礎：リモート・プロシージャ・コール

前述のように、VXI-11 は RPC に基づいたものです。RPC を使うと、リモート関数（ネットワーク上の他のマシンで実行するもの）を、同じコンピュータ上のローカル関数と同様にコールできます。リモート・システムとの通信に関する詳細はほとんど（全部ではない）オペレーティング・システムが行い、ユーザには隠されています。提供されるシステム・コールの例として、`clnt_call()` があります。これは通常の関数コールとよく似た動作をします。クライアントはサーバが関数の実行を終えて結果を返すまで待ちます（同期動作、図 2 を参照）。

RPC は特定のプログラミング言語やコンピュータ・プラットフォームに依存しないように設計されています。RPC サーバ／クライアントは異なるオペレーティング・システムやプロセッサで動作することができます。相互運用性を実現しているのは XDR（データ表現層、図 1 を参照）です。これは、標準のデータ型と、RPC コールで用いられるバイト順序を定義しています。したがって、RPC 関数に渡すパラメータは XDR フォーマットに変換し、戻り値はネイティブ・プログラミング言語のフォーマットに戻す必要があります。

RPC サーバが提供している関数とそのパラメータを知るにはどうしたらいいのでしょうか？サーバには通常、インタフェースの説明を記述した RPCL（RPC 言語）定義ファイルがあります。RPCL は C の型定義によく似ています。例えば、VXI-11 の RPCL ファイルには、図 3 に示すような定義が記載されています。

クライアントは相手のサーバ上でコールしたい関数をどのように識別するのでしょうか？このために、プログラム番号、バージョン番号、プロシージャ番号の 3 つの数値が用いられます。これらの数値は、サーバの RPCL 定義に含まれています。図 3 の例では、0x0607AF がプログラム番号、1 がバージョン番号、10 が `create_link()` 関数のプロシージャ番号です。サーバ・マシン自体をネットワーク上で識別するには、IP アドレスまたはホスト名を使用します。

図 2.　clnt_call() による RPC 関数の同期実行

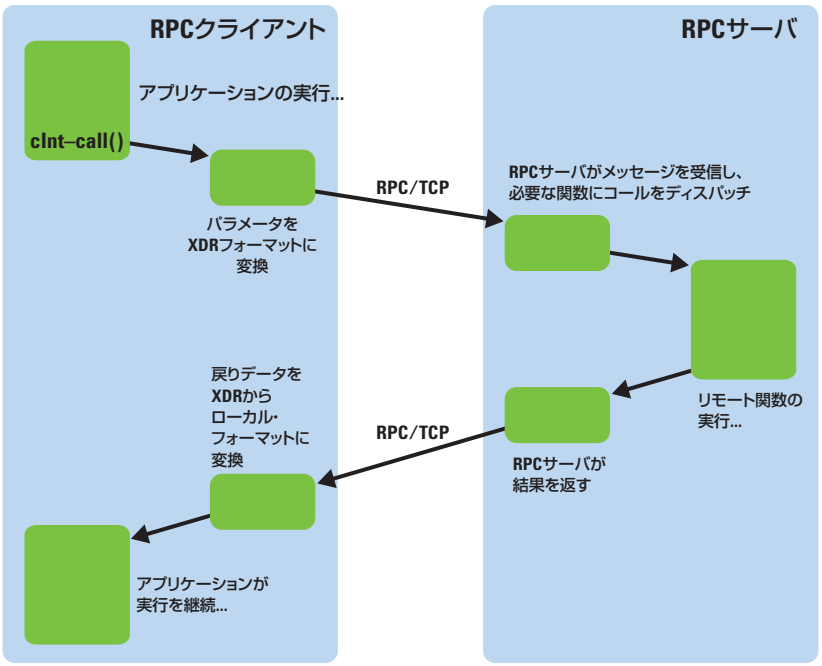



図 3.　主な VXI-11 関数の RPCL 定義

```
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 コード・ジェネレータ

基本的な clnt_call() システム・コールを使って RPC 関数を実行するには、多少手間がかかります。rpcgen ツールを使うと、このプロセスを簡単にできます（図 4 を参照）。rpcgen は、サーバの RPCL 定義を実際の C の宣言に変換します。また、トランスレータ関数（ネイティブ RPC パラメータを XDR に変換するもの）と、ジェネリックな clnt_call() の代わりに実際の関数名で RPC 関数をコールするためのラッパ関数を生成してくれます。

図 5 は、例として VXI-11 の関数 device_write() を使用して、rpcgen の動作を説明したものです。device_write() の元の RPCL 定義は図 5a に示されています。この関数は、SCPI コマンドを測定器に送信するために用いられます（詳細は後述）。

rpcgen で生成されるファイルの中には、C ヘッダ（.h）ファイルがあります。これは、RPCL に記述された番号とデータ型を C で簡単に使用できるように宣言したものです。図 5b は、device_write() 用に生成された定義と、プログラム番号とバージョン番号の定義です。ジェネリックでわかりにくい RPCL のデータ型が、等価な C の構造体（バッファ）に置き換えられています。これらの C 宣言を使うと、関数パラメータのための構造体の作成が比較的容易になり、XDR について考慮する必要がなくなります。

RPC 関数をコールするにはどうすればよいのでしょうか？最も簡単な方法は、rpcgen が生成したラッパ関数を使用する方法です。図 5c に示すのは、クライアント・インプリメンテーション（_clnt.c）ファイル内にある device_write() に対応する関数です。これは、関数の入力パラメータを格納した構造体へのポインタと、RPC リンクへの参照を受け取ります。戻り値は、コール元アプリケーションでアクセスできるデータの構造体へのポインタです。また、ラッパ関数が、xdr.c ファイルで定義された"xdr"トランスレータ関数のアドレスを clnt_call() に渡す方法に注目してください。

**図 4.　rpcgen が生成するファイル**

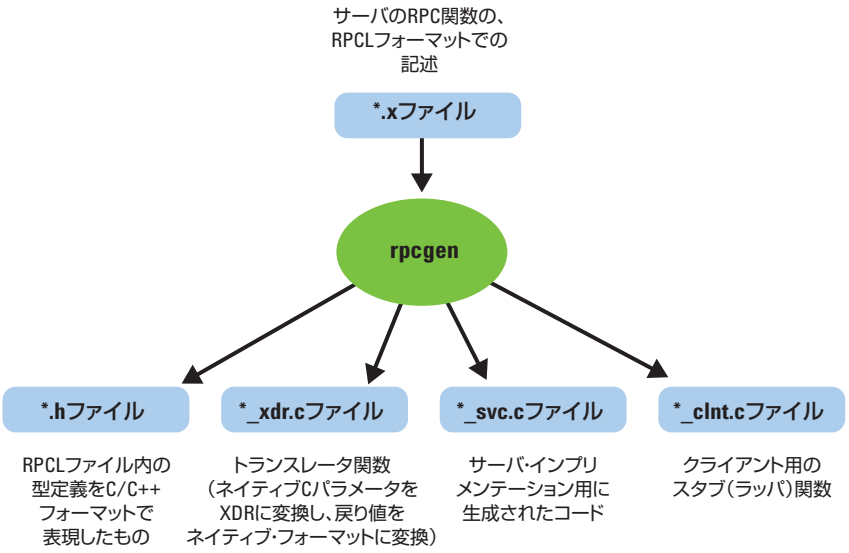


図 5a.　device_write() VXI-11 関数の RPCL 定義

```
struct Device_WriteParms {
   Device_Link lid; /* create_link からのリンク ID */
   unsigned long io_timeout; /* I/O を待つ時間 */
   unsigned long lock_timeout; /* ロックを待つ時間 */
   Device_Flags flags;
   opaque data<>; /* データ長とデータ */
};
struct Device_WriteResp {
   Device_ErrorCode error;
   unsigned long size; /* 書き込まれたバイト数 */
};
Device_WriteResp device_write (Device_WriteParms) = 11;
```

図 5b.　rpcgen が生成したヘッダ・ファイル内の C 宣言

```
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
```

図 5c.　rpcgen が生成した device_write() のラッパ関数

```
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);
}
```

## RPC 用の API コール

表 1 に、RPC 用の基本的な API コールを示します。このほかにもさまざまな関数やフレーバがありますが（詳細については rpc(5) man ページを参照）、ほとんどのアプリケーションに対してはこれで十分です。

## 基本的な VXI-11 関数の使用

`clnt_create()` を使って RPC サーバへのリンクを作成したら、VXI-11 関数へのコールを、rpcgen ラッパまたは `clnt_call()` の直接コールにより実行できます。基本的な VXI-11 関数を表 2 に示します。

**表 1. RPC サーバにアクセスするための基本的なシステム・コール**

| API 関数 | 概要 |
|---|---|
| `clnt_create()` | 特定のホストと、そのホスト上の RPC プログラムにアクセスするための RPC クライアントを作成します。これは常に最初のステップです（clnt_call() を直接使用する場合にも、rpcgen が生成したラッパ関数を使用する場合にも必要）。<br><br>詳細については clnt_create(3) man ページを参照してください。 |
| `clnt_call()` | RPC サーバが提供するリモート関数をコールします。この関数は、rpcgen が生成するラッパで用いられます。<br><br>詳細については clnt_call(3) man ページを参照してください。 |
| `clnt_destroy()` | RPC クライアントを破棄し、リンク用に割り当てられたリソースを解放します。<br><br>詳細については clnt_destroy(3) man ページを参照してください。 |

**表 2. 基本的な VXI-11 関数**

| API 関数 | 概要 |
|---|---|
| `create_link` | VXI-11 サーバ上の論理デバイスに対するリンクを作成します。 |
| `device_write` | 測定器にメッセージ（通常は SCPI コマンド）を送信します。 |
| `device_read` | 測定器からデータ（測定結果など）を読み取ります。 |
| `destroy_link` | create_link で作成したリンクを解放し、割り当てられたリソースを解放します。 |

図 6 に基本的な例を示します。これは、*IDN? 問合せを使って測定器の ID 文字列を読み取ります。最初に、rpcgen が生成したヘッダ・ファイルと C ファイルをインクルードします。

次に、測定器の VXI-11 サーバへの RPC リンクが `clnt_create()` のコールにより作成されます。使用するトランスポート・プロトコル（TCP または UDP）を指定できます。`clnt_create()` は、以降のコールで RPC リンクを参照するためのポインタを返します。

これ以降は、rpcgen が生成したラッパ関数を使って、個々の VXI-11 関数をコールします。`create_link_1()` は、VXI-11 サーバ上の特定の論理デバイスに対する VXI-11 リンクを作成します。返されたデータ構造体には、リンク ID 番号（lid）が記録されています。これをローカル変数に格納します。これはこの論理デバイスに対する以降の VXI-11 コールに必要だからです。”inst0” は、ほとんどのネイティブ LAN 測定器が使用する論理デバイス名です。LAN-GPIB ゲートウェイを使用する場合は、論理デバイス名はサーバの背後にある特定の GPIB アドレス、例えば”gpib0,9” を表します。

次に、`device_write_1()` を使って、*IDN? コマンドを測定器に送信します。SCPI コマンドの末尾には改行文字（‘\n’）が付くことに注意してください。

測定器の応答は、VXI-11 の `device_read_1()` 関数を使って読み取られます。termChar パラメータに注目してください。これは測定器の応答の終了を示す文字を定義します。応答文字列はローカル・バッファにコピーされ、末尾に 0 がアペンドされます。これにより、バッファは標準の C 文字列となり、`printf()` などの標準の C 文字列関数で使用できます。

最後に、`destroy_link_1()` で論理デバイスとのリンクを解放します。これにより VXI-11 サーバは、接続に割り当てられたリソースを解放します。同様に、`clnt_destroy()` は基本的な RPC 接続を解放します。

図 6. VXI-11 を使用した測定器の ID 文字列の読み取り

```
#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) {
   /* エラー処理をここで実行 */
}

Create_LinkParms MyCreate_LinkParms;
MyCreate_LinkParms.clientId = 0; // 未使用
MyCreate_LinkParms.lockDevice = 0; // 排他アクセスなし
MyCreate_LinkParms.lock_timeout = 0;
MyCreate_LinkParms.device = “inst0”; // 論理デバイス名
Create_LinkResp *MyCreate_LinkResp;
if((MyCreate_LinkResp=create_link_1(&MyCreate_LinkParms,VXI11Client))==NULL) {
   /* エラー処理をここで実行 */
}
Device_Link MyLink;
MyLink = MyCreate_LinkResp->lid; // リンク ID を後で使用するために保存
Device_WriteParms MyDevice_WriteParms;
MyDevice_WriteParms.lid = MyLink;
MyDevice_WriteParms.io_timeout = 10000; // ms 単位
MyDevice_WriteParms.lock_timeout = 10000; // ms 単位
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) {
   /* エラー処理をここで実行 */
}

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) {
   /* エラー処理をここで実行 */
}
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)
{
   /* エラー処理をここで実行 */
}
clnt_destroy(VXI11Client);
```

## その他の VXI-11 関数

VXI-11 には、簡単に使用できるその他の関数もいくつか用意されています。これらを表 3 に示します。

注記：これらの動作は、VXI-11 サーバの背後にある論理デバイスによってはサポートされない場合があります（VXI-11 サーバとデバイスとの間の物理インタフェースに依存）。要求された動作が特定のデバイスに対して使用できないこと（これが起こる可能性があります）をサーバが認識すると、コールはエラー・コード 8「サポートされない動作」を返します。

これらの関数のコール方法は簡単です。図 7 の例では、`device_trigger_1()` のコールにより測定器をトリガしています。

表 3. コア・チャネルを使用するその他の VXI-11 関数

| VXI-11 関数 | 概要 |
|---|---|
| `device_readstb()` | デバイスのステータス･バイトを読み取ります。IEEE488 のシリアル･ポール動作に対応します。 |
| `device_trigger()` | デバイスをトリガします。 |
| `device_clear()` | デバイスをクリア（リセット）します。 |
| `device_remote()` | デバイスをリモート・モード（フロント・パネル・コントロール使用不可）にします。 |
| `device_local()` | デバイスをローカル・モード（フロント・パネル・コントロール使用可）にします。 |
| `device_lock()` | デバイスのロック（排他的アクセス権の取得）を試みます。 |
| `device_unlock()` | デバイスのロックを解放します。 |
| `device_docmd()` | デバイス固有コマンドを実行します。 |

図 7. device_trigger によるデバイスのトリガ

```
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) {
      /* エラー処理をここで実行 */
   }
}
```

## アボート・チャネル

一部の VXI-11 動作、例えば device_abort は、独立した RPC リンク（この場合はアボート・チャネル）を使用します。独立したチャネルを使用することにより、測定器がこのコールを適切な優先度で実行するのが容易になります（測定器のアーキテクチャによっては、受信した RPC コールがシリアル化される場合があります）。

図 8 に、アボート・チャネルの RPCL 定義を示します。

アボート・チャネルの使用法は、すでに説明したコア・チャネルの場合とほとんど同じルール／仕組みです。ただし、RPC リンクのセットアップ方法にいくつか違いがあります。コア・チャネルをセットアップする場合は、測定器のポート・マッパ・サービスを使って、適切な TCP ポート番号経由のリンクを作成します。したがって、clnt_create() のコールによって接続を確立する際にポート番号を指定する必要はありません。

これに対して、アボート・チャネル・サーバは、ポート・マッパ・サービスに自分自身を登録していません。この場合は、測定器の RPC サーバが使用する TCP ポート番号を明示的に指定する必要があります。このためには、clnttcp_create() システム・コールを使用します。このシステム・コールは、宛先サーバの IP アドレスとポート番号の両方を保持する sockaddr_in 型の構造体へのポインタを受け取ります。それでは、測定器でアボート・チャネル用に使用されているポート番号を知るにはどうすればよいのでしょうか？ この情報は、clnt_create() のコールによってコア・チャネル・リンクを作成する際に返されるパラメータに含まれています。

図 8 に示すように、アボート・チャネルの RPC サーバは、device_abort という 1 つの RPC 関数だけで用いられます。この関数は、コア・チャネルで処理中のすべての RPC 動作を中止するように測定器に指示します。ほとんどの測定器プログラミングは同期なので、この関数はあまり使用されません。

**図 8. アボート・チャネルの RPCL 記述**

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

**図 9. 割込みチャネルの RPCL 記述**

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

## SRQ（サービス・リクエスト）

VXI-11 には、サービス・リクエストのための機能が含まれています。測定器は、何か通知すべきことが起きたとき（エラーが発生したときや、測定器の出力バッファに測定結果が格納されたとき）、SRQ を使ってシステム・コントローラにシグナルを送ります。

SRQ は、独立した RPC リンク（割込みチャネル）を使用します。SRQ の（コントローラではなく測定器）性質により、サーバとクライアントの役割が逆転します。すなわち、SRQ を送る際に、測定器が RPC 関数 device_intr_srq() をコールし、コントローラ上でこの関数が実行されます。

図 9 に、割込みチャネルの RPCL 記述を示します。

測定器を制御するPCでは、この機能を実現するためのRPCサーバをセットアップする必要があります。RPCサーバ・プログラムはどのように作成すればよいのでしょうか？ここでも、rpcgenがほとんどの作業を行ってくれます。rpcgenは必要な初期化とディスパッチ・ルーチンを生成します（_svc.cファイルを参照）。ユーザが行う必要があるのは、RPCディスパッチャがコールする実際のdevice_intr_srq()関数を作成することだけです。図10に、rpcgenが生成したコードの主要部分を示します。

main関数で、TCPとUDPの両方のトランスポート・プロトコルに対してRPCサーバが登録されていることに注意してください。この例ではUDPのサポートは不要ですが、両方のプロトコルに対してディスパッチ・ルーチンを登録しても害はありません。

**図10. rpcgenが生成した割込みチャネル・サーバのコード**

```
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;
}
```

**図 10、続き**

```
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 */
}
```

rpcgen が生成したコードに対して、printf() ステートメントが追加されています。これは、RPC サーバに割り当てられたポート番号を取得する方法を示すためです。この情報は後で必要になります。

svc_run() は RPC サーバを起動します。このコールは正常な場合には戻り値はありません。したがって、これは通常別のプロセスまたはスレッドで実行されます。

ディスパッチ・ルーチン device_intr_1() は、受信した RPC コールをローカル関数のインプリメンテーションにディスパッチします。この例では、作成する関数は device_intr_srq_1_svc() の 1 つだけです。図 11 に非常に基本的な例を示します。通常は、シグナルやその他のプロセス間通信手段を使って、イベントをメイン・アプリケーション・プログラムに渡します。

RPC サーバが準備できたら、通常のコア・チャネルの VXI-11 コールを使って測定器の SRQ をオンにできます。表 4 を参照してください。

## まとめ

VXI-11 プロトコルは、LAN-GPIB ゲートウェイと、多くのネイティブ LAN ベースの測定器で用いられます。VXI-11 は標準の TCP/IP プロトコルである RPC に基づいたものなので、Linux のすべてのフレーバとバージョンでサポートされています。RPC はシンプルな TCP リンクよりも複雑ですが、rpcgen を使うことでプログラミングが比較的容易になります。

**図 11. device_intr_srq サービス・ルーチンのインプリメンテーション**

```
#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);
}
```

**表 4. SRQ 用の VXI-11 関数**

| VXI-11 関数 | 概要 |
|---|---|
| create_intr_chan() | SRQ 用に使用できる RPC サーバを測定器に通知します。<br><br>割込みチャネルを確立するように測定器に要求します。<br><br>パラメータには、コントローラが用意する RPC サーバの IP アドレスとポート番号が含まれています。 |
| device_enable_srq() | 割込みチャネルの使用を有効／無効にします。 |
| destroy_intr_chan() | 割込みチャネルを解放するように測定器に要求します。 |

1 LXI（LAN Extensions for Instrumentation）と LXI Consortium の詳細については、**http://www.lxistandard.org** をご覧ください。

2 VISA と VXI*plug&play* Alliance の詳細については、**http://www.vxipnp.org** をご覧ください。

3 VXI-11 の詳細については、**http://www.vxibus.org/freepdfdownloads/vxi-11.pdf** をご覧ください。

## Agilent の関連カタログ

1465 シリーズのアプリケーション・ノートは、テスト・システムの構築、テスト・システムで有効に LAN/ 無線 LAN/ USB を使用する方法、RF/ マイクロ波テスト・システムの最適化と拡張についての豊富な情報を提供しています。

### テスト・システム開発

- 『システム開発者ガイド：テスト・システムでの LAN の使用：基礎』AN 1465-9
（カタログ番号 5989-1412JA）
**http://cp.literature.agilent.com/litweb/pdf/5989-1412JA.pdf**

- 『テスト・システムでの LAN の使用：ネットワークの設定』AN 1465-10
（カタログ番号 5989-1413JA）
**http://cp.literature.agilent.com/litweb/pdf/5989-1413JA.pdf**

- 『システム開発ガイド テスト・システムでの LAN の使用：PC の設定』
AN 1465-11
（カタログ番号 5989-1415JA）
**http://cp.literature.agilent.com/litweb/pdf/5989-1415JA.pdf**

- 『システム開発ガイド 計測環境での USB 使用』AN 1465-12
（カタログ番号 5989-1417JA）
**http://cp.literature.agilent.com/litweb/pdf/5989-1417JA.pdf**

- 『システム開発ガイド SCPI + ダイレクト I/O、ドライバの使用法』AN 1465-13
（カタログ番号 5989-1414JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-1414JAJP.pdf**

- 『システム開発ガイド テスト・システムにおける LAN の使用法：アプリケーション』AN 1465-14
（カタログ番号 5989-1416JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-1416JAJP.pdf**

- 『システム開発ガイド テスト・システムでの LAN の使用：システム I/O のセットアップ』AN 1465-15
（カタログ番号 5989-2409JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-2409JAJP.pdf**

- 『LXI による次世代テスト・システム』
AN 1465-16
（カタログ番号 5989-2802JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-2802JAJP.pdf**

### RF/ マイクロ波テスト・システム

- 『RF/ マイクロ波テスト・システムの構成要素の最適化』AN 1465-17
（カタログ番号 5989-3321JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-3321JAJP.pdf**

- 『RF/ マイクロ波テストシステムのテスト品質向上のための 6 ヒント』
AN 1465-18
（カタログ番号 5989-3322JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-3322JAJP.pdf**

- 『システムの信号経路の校正：ベクトルおよびスカラ補正法による測定精度の向上』AN 1465-19
（カタログ番号 5989-3323JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-3323JAJP.pdf**

### LXI（LAN eXtensions for Instrumentation）

- 『次世代 LXI テスト・システム』
AN 1465-20
（カタログ番号 5989-4371JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-4371JAJP.pdf**

- 『LXI に移行する 10 の理由』
AN 1465-21
（カタログ番号 5989-4372JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-4372JAJP.pdf**

- 『GPIB から LXI への移行』
AN 1465-22
（カタログ番号 5989-4373JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-4373JAJP.pdf**

- 『PXI、VXI、LXI によるハイブリッド・テスト・システムの構築』AN 1465-23
（カタログ番号 5989-4374JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-4374JAJP.pdf**

- 『テスト・システムにおけるシンセティック測定器の使用法：利点とトレードオフ』
AN 1465-24
（カタログ番号 5989-4375JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-4375JAJP.pdf**

- 『GPIB から LXI への移行（システム・ソフトウェア編）』AN 1465-25
（カタログ番号 5989-4376JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-4376JAJP.pdf**

- 『LAN/LXI を組み込むための GPIB システムの変更』AN 1465-26
（カタログ番号 5989-6824JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-6824JAJP.pdf**

### テスト・システムでの Linux の使用

サンプル・コードは **http://www.agilent.co.jp/find/linux** からダウンロードできます。

- 『Linux を使用したテスト・システム：Linux の基礎』AN 1465-27
（カタログ番号 5989-6715JAJP）
**http://cp.literature.agilent.com/litweb/pdf/5989-6715JAJP.pdf**

メモとしてお使いください

メモとしてお使いください

電子計測UPDATE

**www.agilent.co.jp/find/emailupdates-Japan**

Agilent からの最新情報を記載した電子メールを無料でお送りします。

**Agilent Direct**

**www.agilent.co.jp/find/agilentdirect**

測定器ソリューションを迅速に選択して、使用できます。

Agilent Open

**www.agilent.co.jp/find/open**

Agilent は、テスト・システムの接続とプログラミングのプロセスを簡素化することにより、電子製品の設計、検証、製造に携わるエンジニアを支援します。Agilent の広範囲のシステム対応測定器、オープン・インダストリ・ソフトウェア、PC 標準 I/O、ワールドワイドのサポートは、テスト・システムの開発を加速します。

LXI

**www.lxistandard.org**

LXIは、GPIBのLANベースの後継インタフェースで、さらに高速かつ効率的なコネクティビティを提供します。Agilentは、LXIコンソーシアムの設立メンバです。

## Remove all doubt

アジレント・テクノロジーでは、柔軟性の高い高品質な校正サービスと、お客様のニーズに応じた修理サービスを提供することで、お使いの測定機器を最高標準に保つお手伝いをしています。お預かりした機器をお約束どおりのパフォーマンスにすることはもちろん、そのサービスをお約束した期日までに確実にお届けします。熟練した技術者、最新の校正試験プログラム、自動化された故障診断、純正部品によるサポートなど、アジレント・テクノロジーの校正・修理サービスは、いつも安心で信頼できる測定結果をお客様に提供します。

また、お客様それぞれの技術的なご要望やビジネスのご要望に応じて、

- アプリケーション・サポート
- システム・インテグレーション
- 導入時のスタート・アップ・サービス
- 教育サービス

など、専門的なテストおよび測定サービスも提供しております。

世界各地の経験豊富なアジレント・テクノロジーのエンジニアが、お客様の生産性の向上、設備投資の回収率の最大化、測定器のメインテナンスをサポートいたします。詳しくは：

**www.agilent.co.jp/find/removealldoubt**


## アジレント・テクノロジー株式会社

本社〒192-8510 東京都八王子市高倉町 9-1

## 計測お客様窓口

受付時間 9:00-19:00 (土・日・祭日を除く)

**FAX、E-mail、Web は 24 時間受け付けています。**

TEL ■■ 0120-421-345
(042-656-7832)

FAX■■ 0120-421-678
(042-656-7840)

Email contact_japan@agilent.com

電子計測ホームページ
www.agilent.co.jp


● 記載事項は変更になる場合があります。
ご発注の際はご確認ください。



Agilent Technologies


April 8, 2008
5989-6716JAJP
0000-00DEP