KI-VISA Library
VISA COM ガイドブック
（KI-VISA 5.0.9 対応版）

Feb 2013 Revision 2.5

KIKUSUI ELECTRONICS CORP.

(This page is intentionally left blank)

Contents

改定履歴

Rev	Date	Filename	Comments
1.0	13/May/2003	KiVisaGuide_J.doc	初版 1.0
1.1	21/Sep/2003	KiVisaGuide_J.doc	改訂版、誤記修正、他
1.2	16/Oct/2003	KiVisaGuide_J.doc	.NET 解説修正、KI-VISA 2.2.4 対応の修正
1.3	29/Jan/2004	KiVisaGuide_J.doc	KI-VISA 2.2.5 対応の修正
2.0	06/May/2010	KiVisaGuide_2010_J.doc	KI-VISA 4.2.5 対応の修正
2.1	06/July/2010	KiVisaGuide_2010_J.doc	KI-VISA 4.2.6 対応の修正
2.2	24/Aug/2011	KiVisaGuide_2011_J.doc	KI-VISA 5.0.x 対応の修正、誤記修正、他
2.3	02/Mar/2012	KiVisaGuide_2012_J.doc	KI-VISA 5.0.6 対応の修正、誤記修正、他
2.4	10/July/2012	KiVisaGuide_2012(July)_J.docx	KI-VISA 5.0.7 対応の修正、誤記修正、 C++Builder XE2 サンプル追加 Delphi XE2 サンプル追加 IVI Config Utility 説明移植
2.5	12/Feb/2013	KiVisaGuide_2013(Feb)_J.docx	KI-VISA 5.0.9 対応の修正、誤記修正、他

ライセンスの同意

1. 権利の許諾

菊水電子工業株式会社（以下「当社」といいます）はお客様に対して、本使用許諾契約に同意いただいて利用可能となる KI-VISA ライブラリ・ソフトウェア及びその関連資料（以下「本ソフトウェア」といいます）に関し、以下の権利を許諾します。

(a) お客様は、本ソフトウェアの無制限数の複製を作成する事が出来ます。

(b) お客様は、本ソフトウェアを無制限数のコンピュータにインストールする事ができます。

(c) お客様は、本ソフトウェアを当社製品を使用する目的で商用及び非商用ソフトウェアに使用する事が出来ます。

(d) お客様は、本ソフトウェアがインストールされたコンピュータが一台以上の当社製品を制御する場合に限り、本ソフトウェアを当社以外の製品を使用する目的で商用及び非商用ソフトウェアに使用する事が出来ます。

(e) お客様は、本ソフトウェアを、当社もしくは当社が容認した第三者によって提供されたアプリケーション・ソフトウェアと共に使用する事が出来ます。

(f) お客様は、上記(c)項もしくは(d)項で示される条件下において、本ソフトウェアをお客様のソフトウェアと共に第三者に再配布する事が出来ます。

2. 追加許諾条項

本ソフトウェアを定められた目的に従って使用した結果、作成された各種のファイルは、お客様の著作物となります。

3. 著作権

本ソフトウェア及びその複製物の著作権は当社又は当社が認めた者が有するものであり、日本国著作権法及び国際条約によって保護されています。本使用許諾契約に基づき、お客様が本ソフトウェアを複製する場合は、ダウンロードされた本ソフトウェアに付されていたものと同一の著作権表示がなされることを要します。

4. 禁止事項

(a) 本ソフトウェアを再配布する場合、一切の改変を行うことなくそのままの状態で再配布しなければなりません。

(b) 本ソフトウェアがバイナリ形式で提供される場合、お客様は、本ソフトウェアをリバースエンジニアすることはできません。

5. 無保証

当社は、本ソフトウェアがお客様の特定の目的のために適当であること、もしくは有用であること、又は本ソフトウェアに瑕疵がないこと、その他本ソフトウェアに関していかなる保証もいたしません。

6. 免責

当社は、いかなる場合においても、本ソフトウェアの使用又は使用不能から生ずるいかなる損害（事業利益の損害、事業の中断、事業情報の損失、又はその他金銭的損害）に関して、一切責任を負いません。

7. 契約の解除

お客様が本使用許諾契約に違反した場合、当社は本使用許諾契約を解除することができます。その場合、お客様は本ソフトウェアを一切使用しないものとします。

7. その他

本ソフトウェア及び本書に記載されている内容は予告なく変更される事があります。

http://www.kikusui.co.jp

1- 概要

1-1 本書について

本書では、KI-VISA ライブラリを使用して計測器のリモート IO 動作を行うプログラムの作成手順を解説します。計測器 IO を行うプログラムは、主にアプリケーション・プログラムと計測器ドライバに大別されます。KI-VISA ライブラリはどちらのタイプのプログラムでも同じように扱う事ができますが、本書では主にアプリケーション・プログラムを作成する手順を前提として説明していきます。

本書は下記に示されるプログラミング言語に対応していますが、特にプログラミング言語やツールに依存した部分を除いて、全て C#言語でサンプル･コードを提示します。

Microsoft C# 2008

Microsoft Visual Basic 2008

Microsoft Visual C++ 2008 (.NET 版 C++/CLI)

Microsoft Visual C++ 2008 (アンマネージド、MFC/ATL)

Microsoft Visual Basic 6.0

言語やツールに依存した部分は別途解説を行います。最後にエラー・ステータス・コードのリファレンスを掲載します。VISA COM API の言語リファレンスに関しては KI-VISA COM オンライン・ヘルプを参照して下さい。

本書で扱われていない言語やツールでも、Microsoft COM (Component Object Model)のクライアント・プログラムの作成機能をサポートするものであれば原則として利用できますが、一部古いツールなどでは使えない事があります。

本書の内容は KI-VISA COM ライブラリ・バージョン 5.0.9 に合わせて記述されています。

1-2 VISA ライブラリとは

VISA (Virtual Instrument Software Architecture)は、VXIplug&play Systems Alliance (現 IVI Foundation)によって策定された、計測器接続ソフトウェアの標準仕様です。「VISA ライブラリ」は VISA 仕様に従って実装されたライブラリ・ソフトウェアであり、Windows の場合は通常 DLL の形式を取ります。VISA ライブラリは、計測制御アプリケーションや計測器ドライバを実行するための IO ドライバであり、それ自体は単体で実行可能なアプリケーション・ソフトウェアではありません。

VISA は、計測器制御に使われる一般的な IO インターフェースとして GPIB、シリアル(RS232)、LAN、USB などを仮想化します。VISA ライブラリを使うことにより、計測制御アプリケーションや計測器ドライバのプログラムは IO インターフェースの種別や GPIB ボードのブランドに依存しない共通の API (Application Programming Interface)セットを通じて、統一されたプログラミング手順で計測器を制御することができます。

Windows ベースの VISA ライブラリには大きく分けて 2 種類の API スタイル 「VISA C」と「VISA COM」があります。

VISA C API

VISA C はその名のとおり C 言語から使われる事を念頭に置いて設計されています。初期の VISA ライブラリではこれが VISA の実体であり、Windows3.1 時代は 16 ビットの VISA.DLL として NI-VISA が存在しました。現在の Windows では VISA32.DLL(32 ビット版 DLL)及び VISA64.DLL(64 ビット版 DLL)がその実体です。VISA C DLL は通常の関数ベースの DLL として実装されており、DLL を呼び出す事の可能なプログラミング言語であれば、C 言語以外でも利用する事ができます。但し、関数のプロトタイプ宣言を記述したヘッダ・ファイルが言語ごとに必要になります。KI-VISA では C/C++及び Visual Basic 6.0 用のヘッダ・ファイルを同梱しています。(他の言語から使用する場合はヘッダ・ファイル又はそれに準ずる物を自作する必要があります。)

Notes:

32 ビット版 VISA32.DLL は VISA の実装ベンダーによって直接提供されますが、64 ビット版 VISA64.DLL は VISA Router DLL と呼ばれる物で VISA Shared Components 64bit Edition によってインストールされます。

VISA Router DLL は実際の IO 処理は行わず、VISA 実装ベンダーによって提供される別の DLL(KI-VISA の場合は System32 ではない別のディレクトリに置かれた 64 ビット版 KIVISA32.DLL)へリダイレクトされます。

VISA COM API

VISA COM は Microsoft COM (Component Object Model)技術に基づいた API スタイルになっています。この API を使うアプリケーション(又は計測器ドライバ)は VISA C DLL を直接参照しません。

COM スタイル API の長所は、やはりなんと言ってもオブジェクト指向である事です。特にアプリケーションの規模がある程度以上大きくなると、オブジェクト指向的な設計アプローチは威力を発揮します。また.NET 開発環境では、VISA COM API を使うための.NET アセンブリ(Primary Interop Assembly)が標準で提供されるので、VISA C API よりも使いやすくなっています。

VISA C API 同様 VISA COM API もまた、言語依存がありません。COM クライアント機能に対応した開発言語ツール或いは.NET 仕様の任意の言語開発ツールであれば基本的にどれでも利用する事ができます。Microsoft Visual Basic であれば Ver5.0 以降、Microsoft Office であれば Office 2000 以降、National Instruments LabVIEW であれば Ver6.0 以降で使用可能です。.NET 環境の言語ではバージョンを問わずインタロップ・アセンブリ経由で COM API を使う事が可能です。

KI-VISA ライブラリでは、VISA COM と VISA C の両方の API セットが提供されます。本書及びオンライン・ヘルプも含め VISA COM API を使うことを前提とし、これを推奨します。

Notes:

VISA COM は IDispatch インターフェースを持たず、IUnknown ベースのカスタム・インターフェースのみを装備しているので、IDispatch インターフェースに依存した(オートメーション或いはデュアル・インターフェースを前提とする)開発ツールからは直接利用できません。オートメーションまたはデュアル・インターフェースのみをサポートするツールの例は、Excel95 VBA、Visual Basic 4.0、Delphi 2、LabVIEW 5.0、および Visual C++で MFC の COleDispatch クラスなどを使用する場合など、殆どが古い開発ツールばかりです。

1-3 KI-VISA の動作環境

KI-VISA ライブラリを使ったプログラムを動作させるには、下記の何れかの OS が動作する PC が必要です。(これ以外にも最近の Windows Server なら動作します。)

- Windows 2000 (x86) SP4 + Update Rollup 1 必須
- Windows XP (x86) (SP2 以降を推奨)
- Windows Vista (x86/x64)
- Windows 7 (x86/x64)
- Windows 8 (x86/x64)

1-4 サポートされる計測器 IO インターフェース

KI-VISA では下記の計測器 IO インターフェースを使用する事ができます。GPIB に関しては、使用するハードウェア(GPIB ボード)に付属のドライバをインストールしておく必要があります。USB、RS232C、LAN に関しては、KI-VISA がドライバを提供します。

GPIB インターフェース (コンテック社製 GPIB ボード)

GPIB インターフェース (インターフェース社製 GPIB ボード、LabVIEW 対応版ドライバ使用時)

GPIB インターフェース (National Instruments 社製 GPIB ボード)

GPIB インターフェース (Agilent Technologies 社製 GPIB ボード、Agilent 488 有効設定時)

シリアル・インターフェース (RS232C)

USB インターフェース (USBTMC)

LAN インターフェース(SCPI-RAW ソケット、VXI-11、HiSLIP)

Notes:

GPIB ボードに関しては、現在各社の製品をサポートしていますが、現時点で KI-VISA の動作を確認できているのは下記の機種だけです。他の機種の動作状況については、動作する可能性は高いですが確認をしていません。また括弧内に記載されているよりも古いバージョンの IO ドライバでの動作も確認していません。

コンテック社:	GP-IB(PCI)L、GP-IB(PM)、GP-IB(PCI)F、GP-IB(CB)F (API-GPIB ドライバ VER4.80 以上推奨)
インターフェース社:	PCI-4301 (LabVIEW 対応版 GPC-4301N ドライバ VER3.00 以上推奨)
National Instruments 社:	PCI-GPIB、GPIB-ENET/100、GPIB-USB-B、GPIB-USB-HS (NI-488.2M Software For Windows, VER2.7 以上推奨)
Agilent 社:	82350 (PCI/GPIB) (IO Libraries Suite 15.5 以上推奨)

コンテック社製 GPIB ボード使用時は、API-GPLV ドライバ(LabVIEW 対応版、NI-488.2M API 互換)ではなく、通常版の API-GPIB ドライバを使用して下さい。インターフェース社製 GPIB ボード使用時は逆に、通常版 GPC-4301 ドライバではなく、GPC-4301N(LabVIEW 対応版、NI-488.2M API 互換)を使用して下さい。

Agilent 製 GPIB ボード使用時は、Primary VISA を非選択、Agilent 488 オプションを選択して下さい。

NI 製 GPIB ボード使用時は、NI-488.2M のインストール時に NI-VISA インストールを除外して下さい。

シリアル・インターフェースに関しては、PC に標準装備されている通信ポート及び、USB-RS232C 変換器などによる仮想シリアル通信ポートをサポートしています。ただし、変換機などによる仮想通信ポートの場合は、提供されるデバイス・ドライバによっては必ずしも正常動作しない場合があるので注意して下さい。(可能であれば Microsoft 製のドライバを使用してください。)

USB 計測器は、USBTMC プロトコルに準拠しているか又は仮想シリアル・ポートをサポートしている必要があります。また USBTMC プロトコルの場合は、計測器の USB インターフェースが IVI Foundation 版の USBTMC デバイス・ドライバでホストされている必要があります。それ以外の条件では KI-VISA からは使用出来ません。

1-5 参考文献および関連仕様書

Table 1-1 参考文献

文献名	**著者・出版社**
Inside COM	Dale Rogerson 著、Microsoft Press (日本語版・アスキー出版 IDBN4-7561-2176-4)

Table 1-2 関連仕様書

仕様書名	**出典**
VPP-4.3 The VISA Library Rev5.0	IVI Foundation http://www.ivifoundation.org
VPP-4.3.2: VISA Implementation Specification for Textual Languages Rev5.0	
VPP-4.3.4: VISA Implementation Specification for COM Rev5.0	
VPP-4.3.5: VISA Shared Components Rev5.0	
IVI-6.1: IVI High-Speed LAN Instrument Protocol (HiSLIP) Rev1.0	
Universal Serial Bus Specification Rev2.0	USB Implementers Forum http://www.usb.org/
Universal Serial Bus Test and Measurement Class Specification (USBTMC) Rev1.0	
Universal Serial Bus Test and Measurement Class, Subclass USB USB488 Specification (USBTMC-USB488) Rev1.0	
LXI Device Specification 2011, Rev1.4	http://www.lxistandard.org/

1-6 商標

本文中に登場する企業名・商品名は各社の商標又は登録商標です。

2- 用語の説明

このドキュメントでは様々な専門用語が使われます。ここでは比較的重要なものを幾つか紹介します。特に、VISA はアプリケーション・ソフトではなく中間レイヤー(ミドルウェア)に位置づけられるため、Windows のシステムに強く依存した説明も多数使われます。

2-1 x86/x64

現在 Windows には 32 ビット版と 64 ビット版が存在します。KI-VISA はこれら両方のシステムに対応します。このビット数を示す総称として「ビットネス」という言い方をします。

x86 は Intel i386 CPU 以降の 32 ビット・アーキテクチャを指す総称です。Windows NT3.1 及び 95 以降現在でもこのアーキテクチャは広く使われていますが、Windows Vista/7 の登場以降、主流は x64 へ移って来ています。尚デバイス・ドライバなどカーネル寄りの技術書では i386 と表記されていることもあります。

x64 は AMD Athlon64 CPU 以降の 64 ビット・アーキテクチャを指す総称です。当初は amd64 と言われていましたが、その後 Intel が Core2 プロセッサで参入してからは x64 と呼ばれるようになりました。(x64 は、Intel Itanium プロセッサで採用している IA-64 というアーキテクチャとは別物です。)

このガイドブックでは x86 及び x64 という統一表記をします。KI-VISA の幾つかのプログラム(特にドライバ関連)では、i386 又は amd64 という名称を使っている部分がありますが、同義語です。

2-2 WOW64

WOW64 (Windows-On-Windows 64)は x64 OS で従来の x86 プログラムを実行するエミュレータです。エミュレータといっても、amd64 系(x64 系)の 64 ビット CPU がネイティブに x86 命令を実行できるため速度低下はありません。

但し WOW64 はユーザー・モードでの実行(EXE, DLL 等)に限られており、デバイス・ドライバ等カーネル・モードで実行されるプログラムは 64 ビット・コードのみ実行できます。つまり、x64 OS 環境では x86 OS 用の古い 32bit デバイス・ドライバは一切動作せず、x64 版デバイスドライバを用意する必要があります。System32 ディレクトリ、SysWow64 ディレクトリ

x86 OS では気にする必要がありませんが、x64 OS では System32 ディレクトリは 64bit のシステム DLL 専用の場所になっています。このディレクトリは通常、32bit プロセスからは見えません。一方 32bit の WOW64 環境では、32bit システム DLL 類は全て SysWow64 ディレクトリに置かれますが、32bit プロセスからは SysWow64 こそが System32 ディレクトリの幻として見えます。このガイドブックでは、このシステム・ディレクトリの事を総称的に<WINSYSDIR>と表記します。

> Notes:
>
> System32 は 64 ビット(x64)のシステム DLL、SysWow64 は 32 ビット(x86)のシステム DLL を置くディレクトリです。「32」と「64」の名前が逆転しているように見えますが、間違いではありません。また 64 ビット・システム DLL のファイル名の多くは xxxx32.dll のように「32」が付きます。これは 32 ビット時代のアプリケーションとファイル名互換を維持するためにそのようになっています。
>
> WOW64 環境で実行される 32 ビットソフトウェアから見た場合、見掛け上の System32 ディレクトリは WOW64 レイヤーによって仮想化されているため、実際には SysWow64 ディレクトリにリダイレクトされます。通常は 32 ビットソフトウェアからは真の System32 ディレクトリは見えません。

2-3 VXIPNPPATH (VISA 標準ルートディレクトリ)

これは VISA ライブラリがインストールされるディレクトリの最上位階層を指します。元々VISA は IVI ではなく VXI Plug&Play で策定された仕様であり、VISA インストール先ディレクトリをこのように呼んでいた為、その言い方が今も残っています。VXIPNPPATH は通常、下記のディレクトリを指す場合が殆どです。

Table 2-1 VXIPNPPATH

OS 環境	**一般的なディレクトリ**
32 ビット(x86) OS	C:/Program Files/IVI Foundation/VISA 又は C:/VXIpnp (←古いバージョンの VISA からアップデートした場合等)
64 ビット(X64) OS	C:/Program Files/IVI Foundation/VISA (64 ビットネイティブモジュール) C:/Program Files (x86)/IVI Foundation/VISA (32 ビット WOW64 モジュール)

このガイドブックでは、<VXIPNPPATH>という表記を行います。これは、実際には上の表に書かれているディレクトリを指します。

2-4 インターフェース

インターフェースという言葉は、ソフトウェアに関連した用語としては様々な意味を持ちます。VISA COM ライブラリに関しては、主に 2 つの意味があります。一つは「ハードウェア IO インターフェース」で、この場合は GPIB、RS-232C、USB、LAN などの通信インターフェースを指します。もう一つの意味は「COM インターフェース」です。これは Microsoft COM (Component Object Model)での専門用語なのですが、IUnknown インターフェースや IVisaSession インターフェースなど、プログラミング上の用語もしくはオブジェクト指向設計の専門用語として登場します。COM インターフェース名は全て大文字の"I"から始まります。

2-5 SCPI-RAW

元々は TCP/IP 無手順ソケットによるコマンド送受信インターフェースですが、最近は SCPI-RAW と呼ばれ、また使用する TCP ポートも 5025 が推奨されています。プロトコル自体に特別な仕掛けは無く、RS232 全二重通信のような動作をします。また GPIB のマルチライン、ユニライン・メッセージやサービス・リクエストに相当する部分を備えていません。

2-6 VXI-11

1995 年に策定された、LAN 用の計測器標準プロトコルの一つ。SUN(現 Oracle)の ONC RPC (Open Network Computing Remote Procedure Call)技術を利用します。SCPI-RAW で欠落している GPIB マルチライン、ユニライン・メッセージやサービス・リクエスト機能を LAN で実現しています。計測器側に実装される RPC Port Mapper によってポート番号が決まるため、使用する TCP ポート番号は不定です。(RPC Port Mapper サービス自体は TCP/UDP ポート 111 で動作します。)

2-7 HiSLIP

High-Speed LAN Instrument Protocol の略。2010 年に IVI Foundation によって策定された LAN 用計測器プロトコルの一つ。VXI-11 のような GPIB 的動作と SCPI-RAW に匹敵する速度性能を実現します。使用する TCP ポートとして 4880 が推奨されています。

2-8 LXI

LAN eXtensions for Instrumentation の略。LAN インターフェースを装備した計測器の業界標準仕様。LXI は LAN の特定通信プロトコルを指すのではなく、LAN 搭載計測器全般の標準仕様について規定しています。

2-9 USBTMC

USB Test & Measurement Class の略。2003 年に策定された、USB I/F(インプリメンターズ・フォーラム)公認のデバイス・クラスの一つ。VXI-11 同様、GPIB 機能を USB に置き換えたアーキテクチャになっています。

2-10 VXI-11 Discovery

VXI-11 プロトコル(UDP 111)を使ったブロードキャスト通信によって計測器を検索する手法。LXI 準拠の計測器は(その LXI Spec バージョンに拠らず)必ずサポートしています。

2-11 mDNS

マルチキャスト DNS。DNS (Domain Name System)をマルチキャストに拡張した名前解決プロトコル。従来の DNS が UDP ポート 53 を使っていたのに対し、mDNS は UDP ポート 5353 を使用します。

2-12 DNS-SD

mDNS を更に拡張してサービス検索を行なうプロトコル。ここでいう「サービス」とは、該当するサーバーがどのような機能を提供しているのかを示します。Apple の独自プロトコル Rendezvouz(ランデブー)を拡張した Bonjour(ボンジュール)が正に DNS-SD そのものです。LXI Spec 1.3 以降準拠の計測器は、この仕様に従った「LXI サービス」の照会に必ず応答します。またこれを利用して計測器の検索にも利用されます。

2-13 IO リソース

IO リソースとは、計測器を接続している具体的な IO インターフェース経路の事を指します。例えば、GPIB ボード(ボード・インデックス 0)に接続されているプライマリ・アドレス 3 の計測器はその接続経路が IO リソースとなり、"GPIB0::3::INSTR"という VISA アドレスで表現されます。

2-14 VISA アドレス

VISA アドレスとは特定の計測器の IO インターフェース、即ち IO リソースを表現する文字列です。かつては IO リソース・デスクリプタ、或いは VISA リソース文字列という言い方をしていた事もありますが、全て同義語です。このガイドブックでは「VISA アドレス」と表記します。

VISA API で計測器との接続を開始するには、この VISA アドレスをパラメータにして計測器接続の「VISA セッション」をオープンする必要があります。VISA セッションをオープンするには、VISA COM リソース・マネージャの Open()メソッドを使うか、VISA C の viOpen()関数を使います。

パラメータに渡す VISA アドレスの表現方法は、IO リソースのタイプ(GPIB, ASRL, USB, LAN 等)やセッションのタイプ(INSTR、INTFC 等)によって異なります。下記に示されるのは、一般的な例です。

Table 2-2 VISA アドレスの例

リソース文字列	解説
"GPIB0::3::INSTR"	GPIB0 に接続されたプライマリ・アドレス 3 の計測器
"GPIB1::4::12::INSTR"	GPIB1 に接続されたプライマリ・アドレス 4、セカンダリ・アドレス 12 の計測器
"GPIB0::INTFC"	GPIB0 のコントーラ・インターフェース
"ASRL1::INSTR"	シリアル・ポート 1 に接続された計測器
"USB::0x0B3E::0x1005::SB001839::INSTR"	USB ポートに接続された、ベンダーID 2878(hex 0B3E)、プロダクト ID 4101(hex 1005)、シリアル番号 SB001839 の計測器
"TCPIP::192.168.0.1::5025::SOCKET"	IP アドレス 192.168.0.1 を持ちポートアドレス 5025 でソケット通信する LAN 計測器(SCPI-RAW)
"TCPIP::192.168.1.23::inst0::INSTR"	IP アドレス 192.168.1.23 を持つ LAN 計測器(VXI-11)
"TCPIP::my-test-instr.local::inst0::INSTR"	mDNS ホスト名 my-test-instr.local を持つ LAN 計測器(VXI-11)
"TCPIP::192.168.1.23::hislip0::INSTR"	IP アドレス 192.168.1.23 を持つ LAN 計測器(HiSLIP)

2-15 セッション・タイプ

セッション・タイプは、VISA では通常 INSTR と INTFC に分けられます。INSTR は計測器(Instrument)のセッションを指し、INTFC はコントローラ・インターフェース(Interface)セッションを指します。但し INTFC が使えるのは GPIB のみです。

例えば"GPIB0::3::INSTR"という VISA アドレスは、GPIB0(GPIB のボードインデックス 0)のバスに接続されたプライマリ・アドレス 3 の「計測器」を指し、"GPIB0::INTFC"は「コントローラ・インターフェース」(この場合はボードインデックス 0 の GPIB ボード自身)を指します。一般に INSTR セッションは計測器との IO を容易に行なうことが出来ますが、あまり細かい制御やトリッキーな操作は不得意です。一方 INTFC セッションは、特殊な動作やトリックを駆使した低レベル操作を行なうことができますが、扱いは難しくなります。また GPIB では通信バスが計測器間で共有に成っているため、INTFC セッションでの操作は、場合によっては同じボード上で動作する INSTR セッションを妨害する可能性もあります。これらはアプリケーションや計測器ドライバでの制御内容に応じて、どちらか一方或いは両方を同時に使うことが出来ますが、混在させる場合には注意して運用する必要があります。

Notes:

VISA では INSTR 以外にも、RAW(USB の場合)、SOCKET(LAN の場合)などがありますが、これらも INSTR 同様に計測器(Instrument)セッションの一種です。計測器業界で推奨される標準プロトコルではないため(デバイス・クラスに準拠しないベンダー固有 USB、TCPIP 無手順ソケットなど)、INSTR の名前が使われていません。

2-16 VISA セッション

VISA セッションとは、特定の計測器を制御する通信ハンドルのようなものです。実際「VISA ハンドル」、或いは「計測器ハンドル」などという言い方もされますが、これらは同義語です。本書では「VISA セッション」という言い方で統一します。

VISA セッションが Open()メソッド又は viOpen()関数によってオープンされると、アプリケーションからは VISA セッションを通じて計測器との通信を行う事ができます。セッションをオープンせずに、いきなり「GPIB0 ボードのアドレス 3 の計測器に"VSET 20"という文字列を送信する」という考え方は VISA では通用しません。必ず VISA セッションが必要です。また、VISA セッションは必要がなくなった時点でクローズしなくてはなりません。

VISA セッションと IO リソースはどちらも、「計測器を接続している特定の経路もしくはそのハンドル」という点で非常に似ています。また区別もつきにくいのですが、これらは明確に異なります。例えば、アプリケーション内に複数のスレッドがあり、それぞれが同じ VISA アドレス「GPIB0::3::INSTR」で示される 1 台の計測器に接続するために、別々に VISA セッションをオープンした場合を考えてみましょう。どちらの場合も、「GPIB0::3::INSTR」という VISA アドレスで VISA セッションをオープンします。この場合、「IO リソースは 1 個だが、同じ IO リソースに接続する 2 個の VISA セッションが存在する」という事になります。単一のリソースオブジェクト(=単一の計測器)を複数のハンドルから操作する事は VISA では特に珍しい事ではありません。アプリケーション内の複数スレッドでなくても、全く別々の複数のアプリケーションが偶然(或いは意図的に)同じ IO リソースに対して VISA セッションをそれぞれオープンする事もあります。更にネットワーク上で動作する LAN-GPIB 変換モジュールや LXI 計測器のようなケースでは、同じ IO リソースに複数の PC からアクセスする場合も想定されます。

これらはいずれの場合も IO リソースは同じですが、VISA セッションは別々のものです。

3- Setup

概要説明と用語説明はこのくらいにして、早速 KI-VISA をセットアップしましょう。

3-1 KI-VISA のセットアップ

KIKUSUI WEB サイトからダウンロード可能な KI-VISA SETUP は 2 種類あります。それぞれ対象 OS が異なるので適切なほうを選んで下さい。(サーバーOS にもセットアップ出来ます。)

Table 3-1　KI-VISA SETUP プログラムの種類

対象 Windows バージョン	SETUP Program (ファイル名は KI-VISA バージョンに依存、下記は VER5.0.9 の例)
8(x64)、7(x64)、Vista(x64)	Kivisa_5_0_9_265(x64).exe
8(x86)、7(x86)、Vista(x86) XP(x86, SP2 以降) 2k(x86, SP4 with Update Rollup 1)	Kivisa_5_0_9_265(x86).exe

Notes:

SETUP は必ず管理者権限でログインした状態で作業して下さい。Vista/7 では UAC(User Account Control)による管理者権限への一時昇格を問われるので、必ず昇格を受理して下さい。

KI-VISA SETUP は必要に応じて.NET Framework 2.0、Visual C++ 2008 再頒布パッケージ、VISA/IVI Shared Components、Microsoft XML Engine 等を自動インストールします。また古い KI-VISA のアンインストールも行います。

x64 版 SETUP ではインストールが終了すると引き続き WOW64 版 SETUP が起動します。そのまま WOW64 版の SETUP を実行して下さい。WOW64 版の SETUP は x86 版のそれと良く似たものですが、x64 OS で 32 ビット・アプリケーションを実行する為の条件に特化させています。

SETUP の起動

管理者権限で SETUP を起動すると、下のようなウェルカム画面が出てきます。

Figure 3-1 Welcome 画面

Next ボタンをクリックすると、ライセンス同意書の確認が出てきます。ライセンス同意書は必ず読んで下さい。更に **Next** ボタンをクリックすると、コンポーネント選択の画面になります。通常は全選択のままにして下さい。

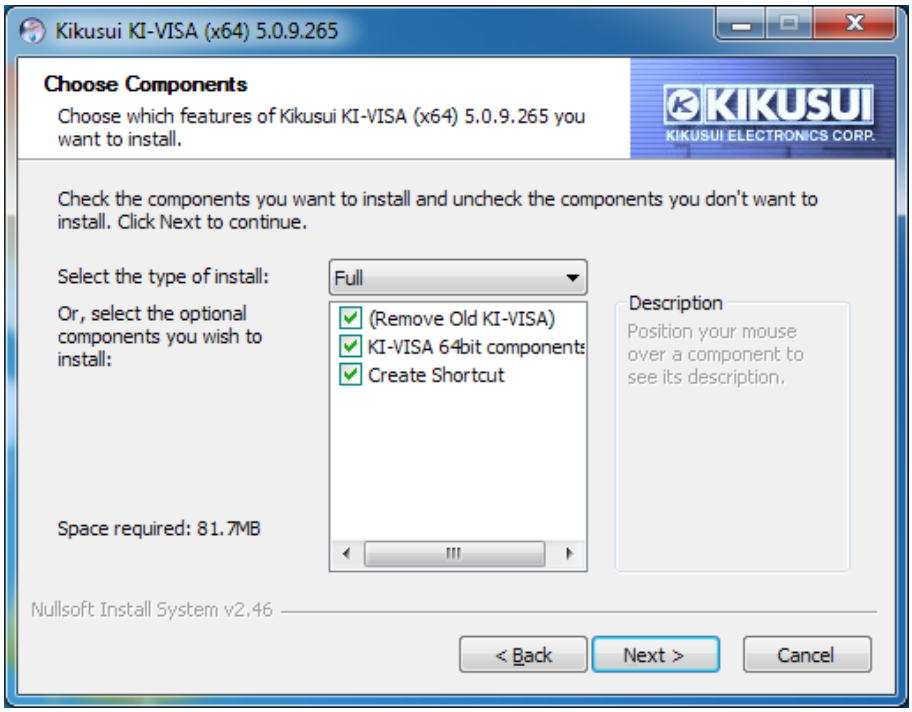

Figure 3-2 コンポーネントの選択

更に **Next** ボタンをクリックすると、インストール先の選択画面になります。

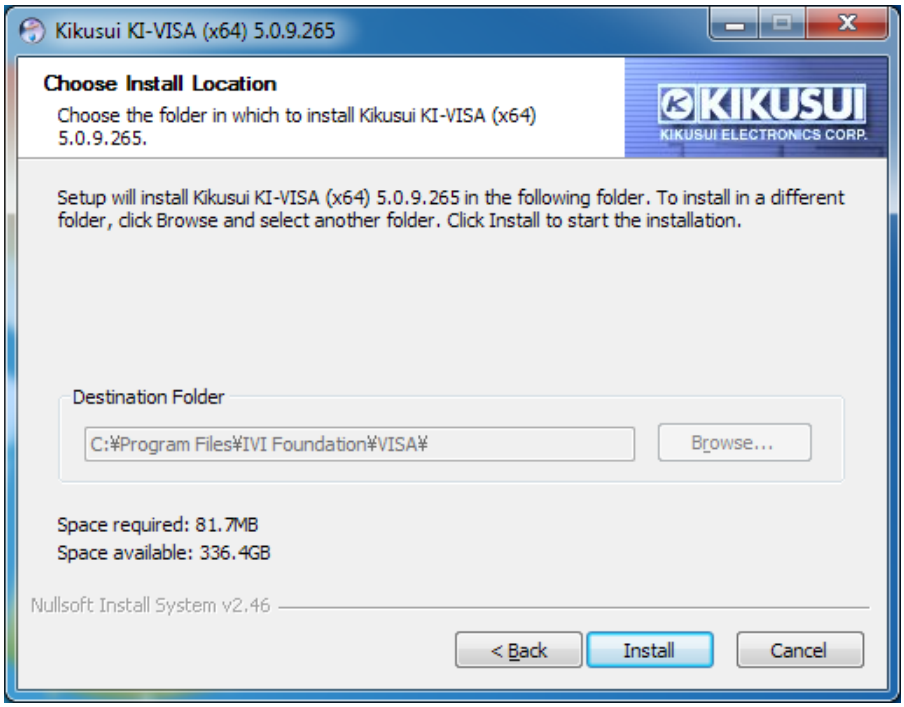

Figure 3-3 インストール先の選択

インストール先のディレクトリは、デフォルトでは<VXIPNPPATH>(VISA 標準ルート・ディレクトリ)になっています。既に VISA ライブラリをインストールした事がある場合は、インストール先がグレーのまま変更不可になっている場合があります。変更可能な場合であっても、通常はデフォルトのままインストールして下さい。

更に **Install** ボタンをクリックするとセットアップが開始します。ファイルのコピーやレジストリへの登録が終了すると、完了画面が表示されます。**Close** ボタンをクリックするとセットアップを終了します。

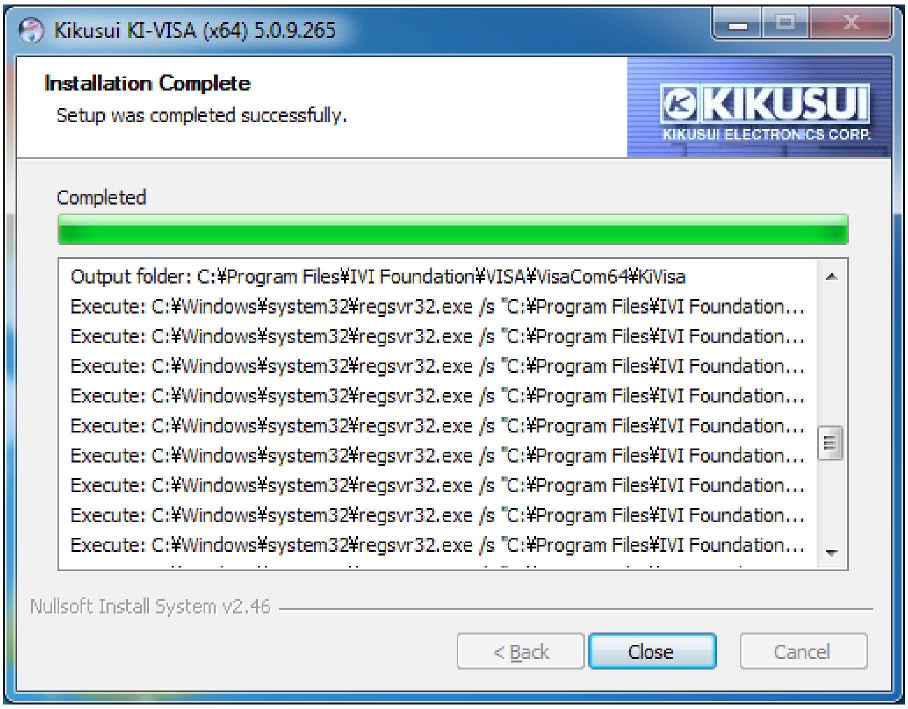

Figure 3-4 インストールの完了

x86 OS へのインストールの場合はセットアップはこれで終了です。x64 OS へのインストールの場合、引き続き WOW64 版 KI-VISA のセットアップが自動的に起動します。WOW64 版 KI-VISA は 32 ビットプログラムで必要になるので必ずインストールしてください。

Figure 3-5 WOW64 版 KI-VISA のセットアップ(x64 OS のみ)

作成されるショートカット

インストールが完了すると **Start** ボタン➔**All Programs**➔ **Kikusui IO Software** にフォルダの傘下にショートカットが作成されます。そこには x64 版(x64 OS の場合)又は x86 版(x86 OS の場

合)のフォルダと、各プログラム(ユーティリティ等)へのショートカットが作られます。**KI-VISA Instrument Explorer** のと **KI-VISA SPY** のショートカットはデスクトップにも作られます。

- KI-VISA Instrument Explorer
- IVI Configuration Utility
- Release Note (E/J)
- Help (VISA COM リファレンス・マニュアル)
- KI-VISA SPY
- KI-VISA Help

KI-VISA Instrument Explorer は、通信インターフェースの環境設定を行ったり、計測器との簡単な IO 接続テストをを行うための対話式ツールが含まれます。**KI-VISA SPY** は、KI-VISA を経由して行われる IO トラフィックの内容を傍受するデバッグ支援ツールです。**IVI Configuration Utility** は、IVI 計測器ドライバーの仮想計測器(Logical Name)や計測器セッションなどを設定するツールです。

KI-VISA Instrument Explorer と KI-VISA SPY については、このガイドブックで説明します。

Notes:

x64 OS にインストールした場合、WOW64 版(32 ビット版)のユーティリティへのショートカットは作られません。通常 32 ビット版のユーティリティを使用する必要はありませんが、どうしても起動したい場合は C:/Program Files (x86)/IVI Foundation/VISA/VisaCom/KI-VISA ディレクトリから直接起動して下さい。

4- KI-VISA Insrument Explorer (IO Config)

KI-VISA の SETUP が終了したら、まず Instrument Explorer を起動して IO コンフィグレーションをして下さい。Instrument Explorer の起動ショートカットアイコンはデスクトップにあります。

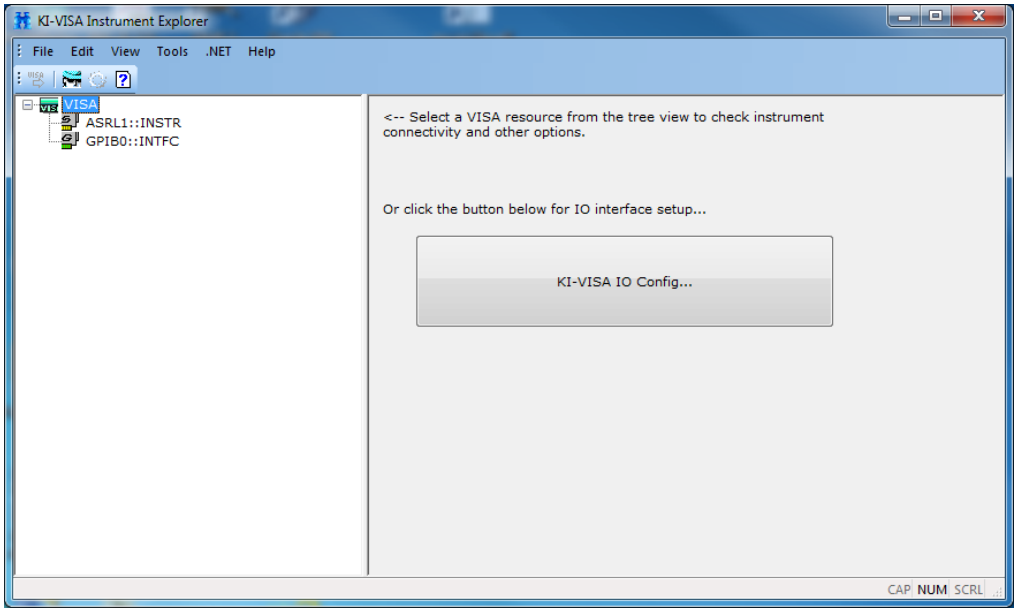

Figure 4-1 KI-VISA Instrument Explorer

4-1 IO コンフィグレーション

まず最初に、画面に大きく表示されている **KI-VISA IO Config...**ボタン(又は **Tools➔Kikussui IO Config** メニュー)から **KI-VISA IO Config** ダイアログを表示します。では、各 IO インターフェースの機能と設定方法を説明します。

Serial

RS-232 又は RS-485 などのシリアル・ポートの設定です。仮想 COM ポートもこれに該当します。(USB-シリアル変換ケーブル、USB を装備していながら仮想 COM ポートを提供する計測器等。)

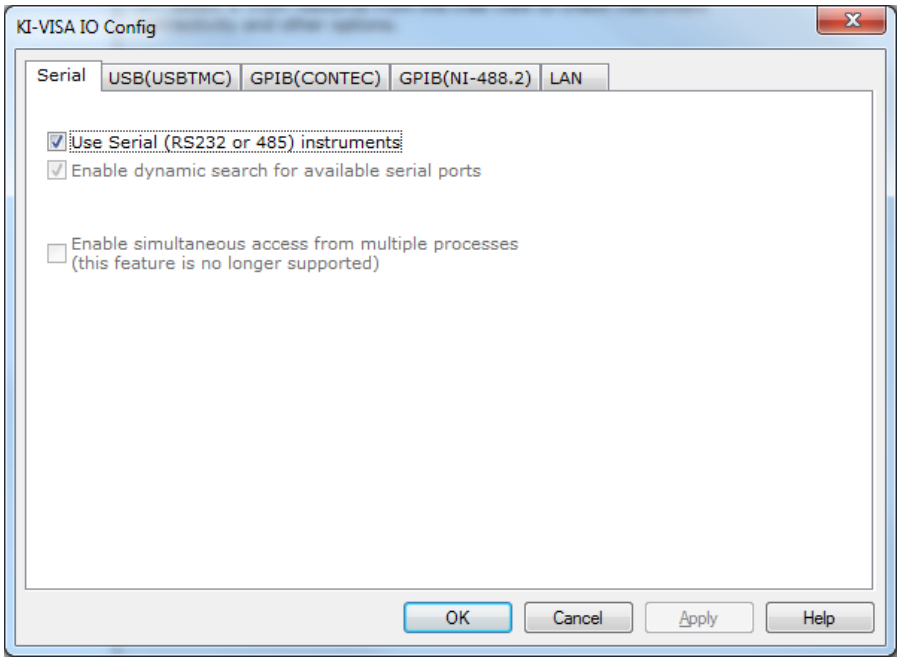

Figure 4-2 IO Config (Serial)

Use Serial (RS232 or 485) instruments のをチェックした場合、KI-VISA はシリアル接続の計測器 IO を有効にします。

Enable dynamic search for available serial ports のチェックボックスは常にチェックされているので、利用可能な IO リソースの検索は自動的に行われます。

Enable simultaneous access from multiple processes は機能削除されたため選択できません(下記参照)。

Notes:

複数プロセスから同一 COM ポートの VISA セッションを多重オープンする機能は、以前はサポートされていましたが、KI-VISA VER5.0.4 以降正常に動作しなくなり、VER5.0.6 から正式に機能削除されました。これは受信バッファの処理メカニズムが大きく設計変更されたため、複数プロセスからの多重アクセスに対する受信データの振り分けが困難になったためです。

同一プロセスから同一 COM ポートへの VISA セッションを多重オープンする機能は現在もサポートされています。但し、仮想 COM ポートを使用した場合はデバイスに依っては動作が不安定になる場合があります。

USB(USBTMC)

USBTMC プロトコルに準拠した計測器を接続するための設定です。

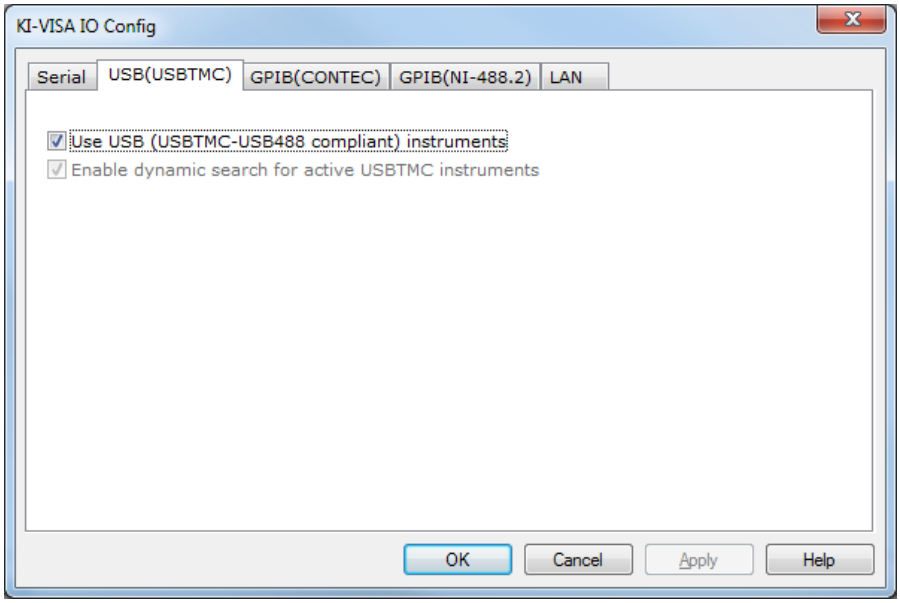

Figure 4-3 IO Configuration (USB)

Use USB (USBTMC-USB488 compliant) instruments のをチェックした場合、KI-VISA は USB 計測器の IO 機能を有効にします。

Enable dynamic search for active USBTMC instrument のチェックボックスは常にチェックされているので、利用可能な IO リソースの検索は自動的に行われます。

GPIB (CONTEC)、GPIB(NI-488.2)

CONTEC 製又は NI-488.2M 互換の GPIB を使用するための設定です。

CONTEC サポート機能では CONTEC 製の GPIB ボードのみ使用できます。NI-488.2M サポートでは、National Instruments NI-488.2M 互換の GPIB ボードをメーカー問わず使用できます。但し、互換を謳う全ての GPIB 製品を動作検証しているわけではありません。現時点動作確認できている主なメーカーは、National Instruments、Agilent Technologies、INTERFACE です。

CONTEC と NI-488.2M では設定用のページが別々に用意されていますが、ここではまとめて説明します。

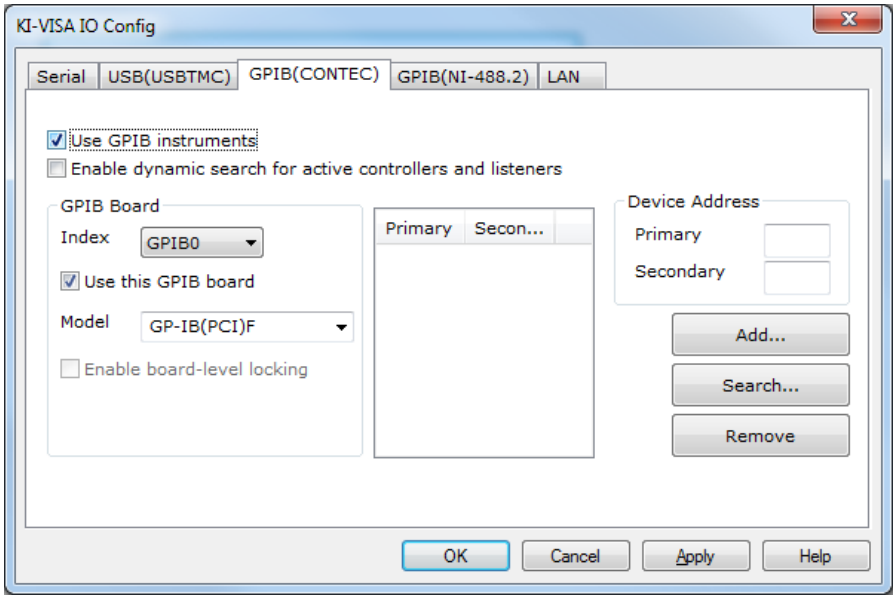

Figure 4-4 IO Config (GPIB CONTEC)

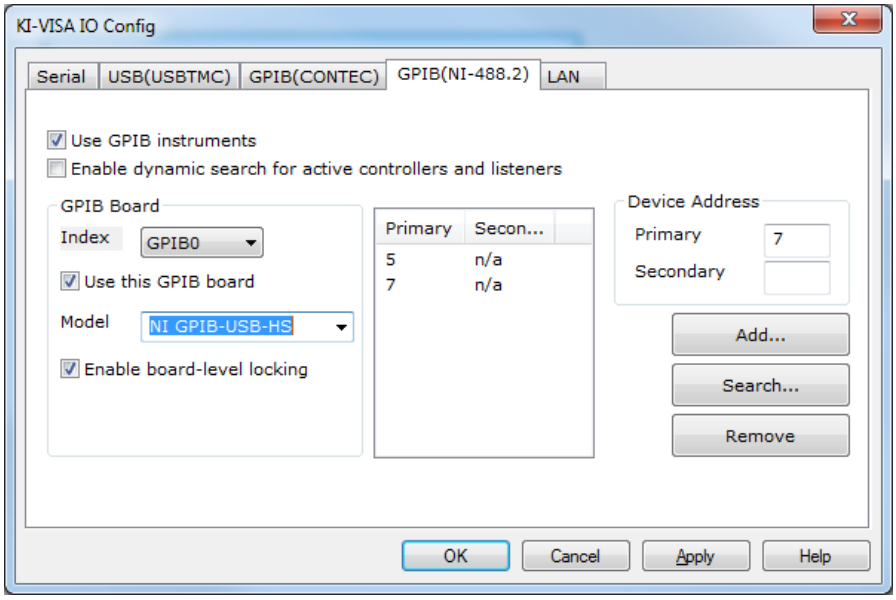

Figure 4-5 IO Config (GPIB NI-488.2)

Use GPIB instruments のをチェックした場合、KI-VISA は該当するタイプの GPIB 機能を有効にします。

Enable dynamic search for active controllers and listeners のをチェックした場合は、利用可能な IO リソースの検索は動的に行われます。そうでない場合は、このコンフィグレーション・プログラムで事前に行っておく必要があります。通常はこのダイナミック検索機能を選択しておけば良いでしょう。

GPIB Board の区画では、GPIB ボードインデックス毎に使用するかどうかを設定します。例えば、GPIB0 を使用する場合、まず **Index** から GPIB0 を選択肢して **Use this GPIB board** をチェックします。すると **Model** のコンボボックスが有効になるので、実際に使用する GPIB ボードの機種名を選択します。

ダイナミック検索がチェックされていない場合、**Device Address** の **Primary/Secondary** に入力した計測器のアドレスを **Add** ボタンで 1 個ずつ追加してやる必要があります(セカンダリ・アドレスを使用しない場合は **Secondary** を空白にして下さい。)。或いは、ここで **Seach** ボタンで一時的な検索を行う方法もあります。

Enable board-level locking(NI-488.2 のみ)は、VISA のリソースロッキング機能を使う際、NI-488.2M GPIB ドライバー又はボードハードウェア自身が提供する排他制御メカニズムを利用するかどうかのオプションです。これは、通常のローカルバス(PCI バス、PCIe バス、USB バスなど)接続型の GPIB ボードではチェックする必要ありませんが、GPIB-ENET/100 のようなネットワーク対応の GPIB ボード(アダプタ)では意味を持ちます。ボードレベルのロック機構が有効になっている場合、あるマシンから VISA でロックを掛けた場合、他のマシンから同じネットワーク型 GPIB ボードへのアクセスは制限されます。

Notes:

CONTEC と NI-488.2M の両方を有効にして、例えば CONTEC 製 GPIB ボードと NI 製 GPIB ボードの両方を同時に使用する事は可能です。但し、この場合、同じボードインデックス(GPIB0 等)を両方で同時に使用しないで下さい。

Model コンボボックスでは、GPIB ボードの機種選択あるいは手入力を必ず行って下さい。ここが空白のまま放置された場合、KI-VISA が正常に機能しない場合があります。使用しないボード・インデックスに関しては空白のままで問題ありません。

■CONTEC 固有の注意

CONTEC の場合には、Model コンボボックスで設定するコンテック社製 GPIB ボードの機種名を正確に選んで下さい。KI-VISA は、設定された GPIB ボードの機種名によってボード機種に合わせた微細な分岐処理を行っている個所があります。実際には、F シリーズと呼ばれる機種(型名の後ろに F が付くモデル、但し GP-IB(PC)F は除外)と、F が付かない古い機種とではデバイス・ドライバが異なる為、KI-VISA でも動作を調整しています。

コンテック社製 GPIB ボードでは、F が付かない古い機種を x64 OS で使う事はできません。(デバイス・ドライバが対応していないため。)

コンテック社製 GPIB ボードを使用するにはコンテック API-TOOL (API-GPIB VER4.80 以上)を推奨します。可能な限り最新版の GPIB ドライバをインストールして下さい。インストールの順序は KI-VISA より前でもあとでも構いません。また API-GPIB の開発環境は必要なく、実行環境のみで十分です。

コンテック社製 GPIB には、LabVIEW 対応版 API-GPLV というドライバも存在します。これは事実上 NI-488.2M 互換 API(GPIB-32.DLL)を提供するものです。理論的には、次に説明する GPIB(NI-488.2)機能を通じて動作させる事は可能かも知れませんが、現在 KI-VISA では動作保証していません。

■NI-488.2M 固有の注意

NI-488.2M の場合には、Model で設定する GPIB ボードの機種名は動作上特別な意味はありません。機種名に関わらず同じ動作となります。

KI-VISA は NI-488.2 互換 API の DLL として、GPIB-32.DLL を第一候補、それが見つからない場合 NI-4882.DLL を使用します。x86 プログラム(32 ビット OS 又は WOW64 環境)ではほぼすべての場合で GPIB-32.DLL ですが、x64 プログラムの場合 GPIB ボードベンダーによって DLL が二通りあります。しかし KI-VISA ではどちらもサポートしています。

NI-488.2M 互換 API を提供する GPIB ボードは沢山ありますが、全てを動作保証しているわけではありません。互換品質の悪い DLL を提供する製品では、正常動作しない、あるいはシステムがクラッシュする可能性があります。

インターフェース社製 GPIB ボードを使用する場合は、LabVIEW 対応版 GPC-4301N を必ず使用して下さい。インターフェース社からは、LabVIEW 対応版でない通常版のドライバ(GPC-4301)も提供されています。しかし KI-VISA が要求する「複数プロセスからの同一 GPIB ボードの同時運転」が正常に機能しない事が確認されているため、KI-VISA では GPC-4301 を利用した IO 機能はサポートしていません。

Agilent 製の GPIB ボードを使用する場合、Agilent IO Libraries Suite 15.5 以降を使用して下さい。また、Primary VISA の選択を外す事、Agilent 488 オプションを有効にすることを忘れないで下さい。

ナショナル・インスツルメンツ社製 GPIB ボードを使用する場合、使用する GPIB ボード対応した NI-488.2M ソフトウェア(ドライバ)が必要になりますが、それをインストールする際 NI-VISA のインストールを必ず除外して下さい。NI-VISA と KI-VISA の共存は出来ません。

一般に、NI-488.2M 互換 API (GPIB-32.DLL)は、異なるベンダー同士の物を混在させて使用することはできません。これは異なるバージョンの GPIB-32.DLL を同時にインストールする事ができない為です。

Enable board-level locking を有効にした場合、VISA の LockRsrc()/viLock()関数は、NI-488.2M API の iblck(1)関数を呼び出してロックの獲得(又は回数のインクリメント)を行います。同様に UnlockRsrc()/viUnlock()関数は iblck(0)を呼び出してロックのデクリメント(0 に成れば開放)を行います。

前記のボードレベルロッキング機能を利用できるのは、NI-488.2M 互換の GPIB ボードで、セッションタイプが GPIBx::INTFC の場合だけです。INSTR セッションには関与しません。また、一部の NI-488.2M 互換 API では iblck()関数自体が実装されていないケースがあります。その場合には、設定に関わらず iblck()は呼び出されません(KI-VISA はドライバに iblck()関数エントリが存在する場合のみ呼び出します。)。尚、必要が無い場合には、この機能を出来るだけ無効にして下さい。順不同で Lock/Unlock を繰り返した場合に Lock/Unlock 管理の複雑度が上がってしまうため、デッドロックしてしまう危険性があります。

LAN

LAN に接続された計測器の設定を行います。ここで扱う LAN インターフェースは SCPI-RAW (TCP/IP ソケット)、VXI-11、HiSLIP に対応した計測器です。LXI 規格に準拠した計測器であれば、これらのうち最低限 1 つ以上は機能が備わっています。(LXI 計測器では、計測器自身の埋め込み WEB ページを見ることで機能確認が出来ます。)

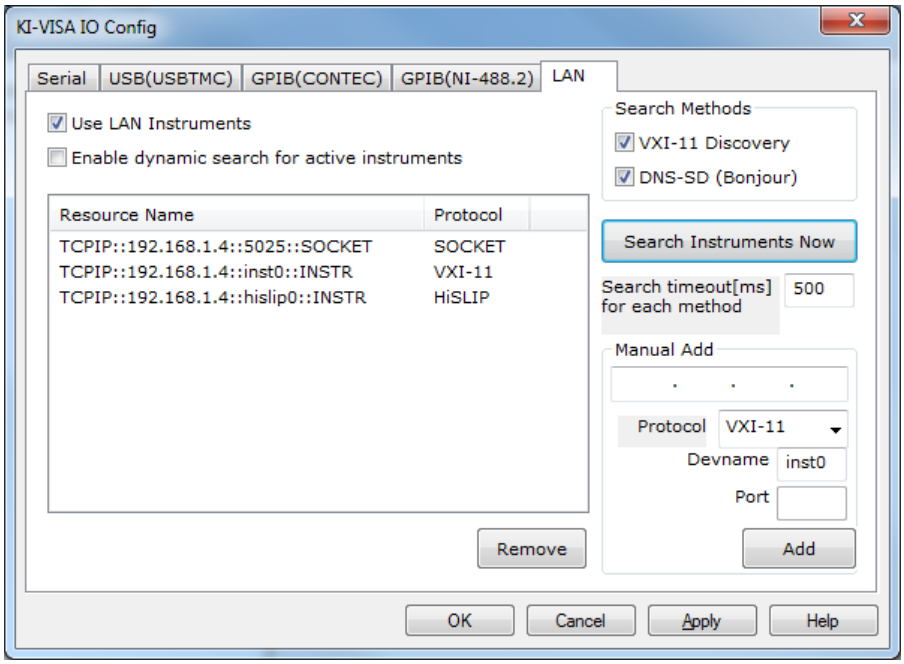

Figure 4-6 IO Config (LAN)

Use LAN instruments のをチェックした場合、KI-VISA は LAN 接続された計測器との通信を有効にします。

Enable dynamic search for active instruments をチェックした場合は、利用可能な IO リソースの検索は動的に行われます。そうでない場合は、このコンフィグレーション・プログラムで事前に行っておく必要があります。LAN の場合は検索に時間がかかる場合があるので、通常は動的検索はオフにしておいた方が良いでしょう。

Search Method(検索方法)として、**VXI-11 Discovery** 及び **DNS-SD (Bonjour)**を選択する事ができます。前者は VXI-11 Discovery 機能を利用した検索で、後者は mDNS(マルチキャスト DNS) を利用した検索です。少なくともどちらかは選択しておく必要があります。

Search Instruments ボタンをクリックすると、現在選択されている検索方法を利用して、KI-VISA が稼働している PC と同じサブネット内に存在する LAN 計測器を検索します。**Search Timeout** は検索の際の待ち時間です。両方の検索方法が選択されている場合は、それぞれに待ち時間がかかります。

Manual Add の区画では、検索とは別に手動で計測器を追加する事ができます。IP アドレスの他には、**Protocol** の選択(SOCKET/VXI-11/HiSLIP から選択)、**Devicename** の指定(VXI-

11/HiSLIP のみ)、**Port** 指定(SOCKET 及び HiSLIP のみ)が必要です。これらを入力したら Add Instrument Manually ボタンをクリックする事で追加できます。検索が利かない LAN 計測器やルータを越えてアクセスする場合などは、この方法で VISA アドレスを追加してください。

Notes:

DNS-SD (Bonjour)による検索にヒットするには、計測器が mDNS/DNS-SD に対応している必要があります。このプロトコルは LXI 仕様 Ver1.3 以降に準拠する計測器には必ず実装されています。

KI-VISA の DNS-SD 機能は、使用する PC に Apple Bonjour がインストールされている場合のみ動作します。Bonjour がインストールされていない場合にこれを選択すると、Apple Bonjour のランタイム・エンジン(実際には Bonjour Printer Wizard)をダウンロード可能な URL に誘導されます。必要に応じてインストールすることです。

VXI-11 Discovery 及び DNS-SD はそれぞれ UDP のブロードキャスト(ポート 111)又はマルチキャスト(ポート 5353)で通信が行われる為、KI-VISA を実行する PC と検索対象の計測器が同じサブネット内のアドレスを持つ必要があります。異なるサブネットを跨いでの検索及びルーターを越えての検索はできません。

サブネットが違っても VISA による計測器の通信が出来ないわけではありません。但しその場合は検索にはヒットしないので、手作業による IP アドレスの追加が必要です。

特定 IP アドレスやアドレス範囲がファイアーウォール等でブロックされている場合は、計測器との通信ができない場合があります。

PC 側のポート 111 をファイアーウォールでブロックすると、VXI-11 機能は使えなくなります。

PC 側のポート 5353 をファイアーウォールでブロックすると、mDNS/DNS-SD 機能は使えなくなります。

コンフィグレーションの保存

コンフィグレーションを終えたら、**Apply** 又は **OK** ボタンをクリックして変更を適用して下さい。その際、全ての設定がシステム・レジストリに保存されます。レジストリの場所は HKEY_CURRENT_USER の下の、SOFTWARE\Kikusui\KiVisaCom\CurrentVersion 以下のブランチです。同じコンピュータを複数ユーザで別々にログオンしている場合は、レジストリ設定はユーザ毎に別々に管理されます。

4-2 KI-VISA Insrument Explorer (対話式制御)

IO コンフィグレーションが完了している状態では、Instrument Explorer 画面には、設定又は認識された全ての VISA アドレスが左側のツリーに表示されています。(何らかの理由でツリーが最新表示になっていない場合は、F5 キーを押して下さい。)

Notes:

通常、OS によって認識される有効なシリアルポート(計測器が接続されているかどうかとは無関係)と現在接続されている USBTMC デバイスに関しては、VISA アドレスが常に表示されます。

どれかひとつの VISA アドレスをクリックすると、右側の画面に詳細設定が表示されます。この内容は全ての IO インターフェースで共通の項目と、インターフェースによって異なるものがあります。

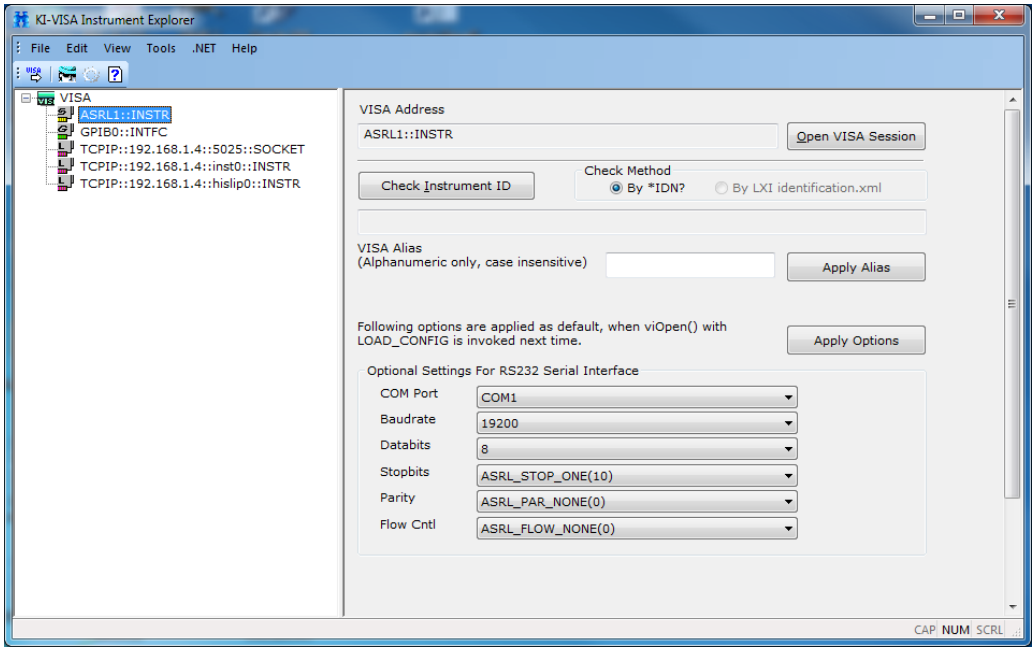

Figure 4-7 Instrument Explorer 全体画面

Check Instrument ID

Check Instrument ID ボタンは、簡易的な機種 ID(*IDN?クエリ相当)チェックを行います。LAN の場合は*IDN?クエリによる照会(**By *IDN?**)のほかに **By LXI identification.xml** も選択可能です。この方法は LXI spec 1.2 以降で必須要件となっている identification.xml を HTTP(WEB)インターフェースで照会して機種 ID を取得します。

Notes:

By LXI/identification.xml での動作は、LXI spec 1.1 以前及び LXI 準拠を謳っていない古い LAN 計測器では応答しません。

VISA Alias

現在選択されている VISA アドレスに対してエイリアス(別名)を追加します。この機能を使うと、長い VISA アドレスの代わりにエイリアスを使って VISA セッションをオープンすることが出来ます。また、異なる VISA アドレスに対してエイリアスを割り当てしなおせば、計測器本体を入れ替えた際に、計測アプリケーション側での VISA アドレス設定を変更せずにシステムを運用できます。この点は VISA アドレスに計測器のシリアル番号を含んでいる USB のようなケースでは特に有効です。

エイリアスに使用できる文字は英数字(0..9, A..Z, a..z)とアンダースコア(_)のみです。ブランク文字列を指定するとエイリアスを削除する事を意味します。エイリアスはケース・インセンシティブ(大文字小文字の区別なし)なので、例えば、"com1"と"COM1"は同じものとみなされます。KI-VISA のデフォルトでは、エイリアスは何も設定されていません。

Notes:

同じ VISA アドレスに対して複数のエイリアスを指定することはできません。また 1 つのエイリアスの参照先を複数の VISA アドレスにすることもできません。既に割り当て済みのエイリアスを別の VISA アドレスに割り当てた場合、後から割り当てた物が有効になります。

エイリアスを設定したら忘れずに Apply Alias ボタンをクリックして下さい。適用せずに他のページへ移動したり KI-VISA Instrument Explorer を終了すると設定が適用されません。

VISA エイリアスの管理方法は VISA 仕様書によって標準化されているわけではなく、VISA ライブラリの実装によって様々です。KI-VISA の場合は、レジストリ内の HKEY_CURRENT_USER\Software\Kikusui\KI-VISA\CurrentVersion\Aliases に保存されています。レジストリ・エディタで直接編集することも可能です。

Optional Settings For RS232 Serial Interface

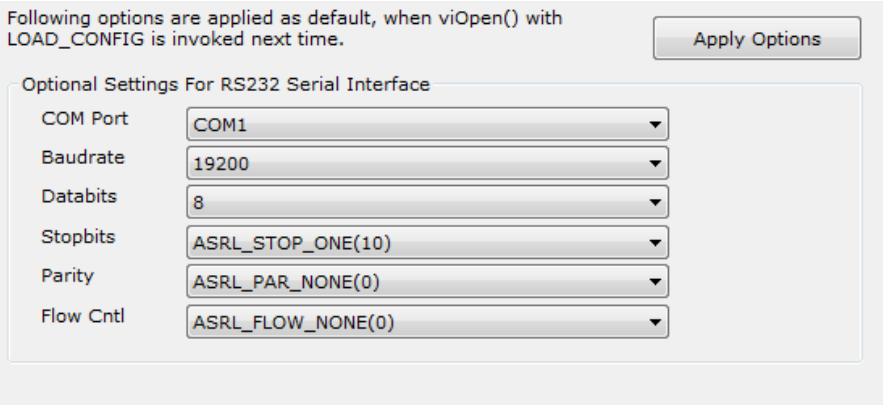

Figure 4-8 RS232 オプション

この機能はシリアル・インターフェース(ASRL)専用です。VISA セッションをオープンする際、LOAD_CONFIG オプションを指定すると、VISA 仕様によるデフォルト設定(9600/N/8/1/FLOW 無し)ではなく、ここで指定した条件が即座に適用されます。後述する **Open VISA Session** で VISA セッションをオープンする場合もこのオプションを指定してします。

Optional Settings for LAN Interface

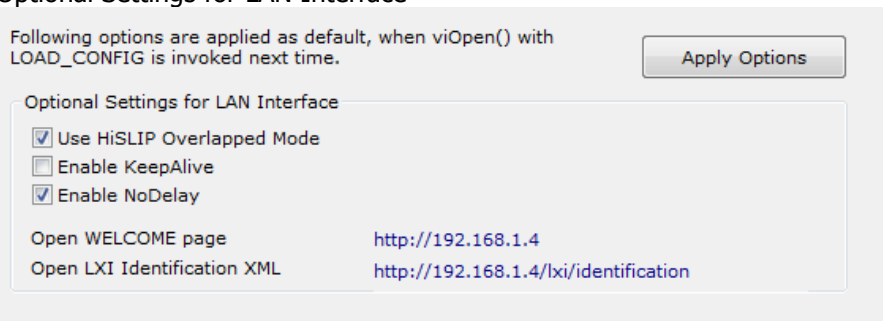

Figure 4-9 LAN オプション

この機能は LAN インターフェース専用です。**Use HiSLIP Overlapped Mode** をチェックすると HiSLIP でのオーバーラップド・モードの動作になります。オーバーラップド・モードは HiSLIP が備える Query INTERRUPTED の検出機構を停止し、クエリと応答回収の同期維持管理をやて SOCKET のような動作になります。**Enable KeepAlive** をチェックすると、キープアライブ・パケットの送信を行い、長時間通信トラフィックが無い場合の自動切断をしないようになります(通常キープアライブは 2 時間)。**Enable NoDelay** は通常はチェックされていますが、これをアンチェックすると Nagle(ネイグル)アルゴリズムによる遅延 Ack をサポートするようになります。

Open WELCOME page は IP アドレスの手前に「http://」を付けた URL を指定してブラウザを起動します。通常は計測器自身の WEB サーバーに用意されたデフォルトの WELCOME ページが表示されます。LXI 準拠の計測器では WELCOME ページの提供が必須要件なので、必ず何かしらのページが表示されます。コンテンツや外見は計測器ごとに異なります。**Open LXI identification XML** は LXI 1.2 以降で必須となっている LXI Identification.XML ファイルを照会します。

Open VISA Session

選択された VISA アドレスで VISA セッションをオープンします。オープンに成功すると、対話形式の制御画面が表示されます。画面左のツリーから VISA アドレスをダブル・クリックしても同じ動作になります。VISA Session を開いて対話形式で計測器と通信する方法は後述します。ここでの Open 操作は、上で説明したオプション設定が適用されるように LOAD_CONFIG オプション付きで呼び出されます。

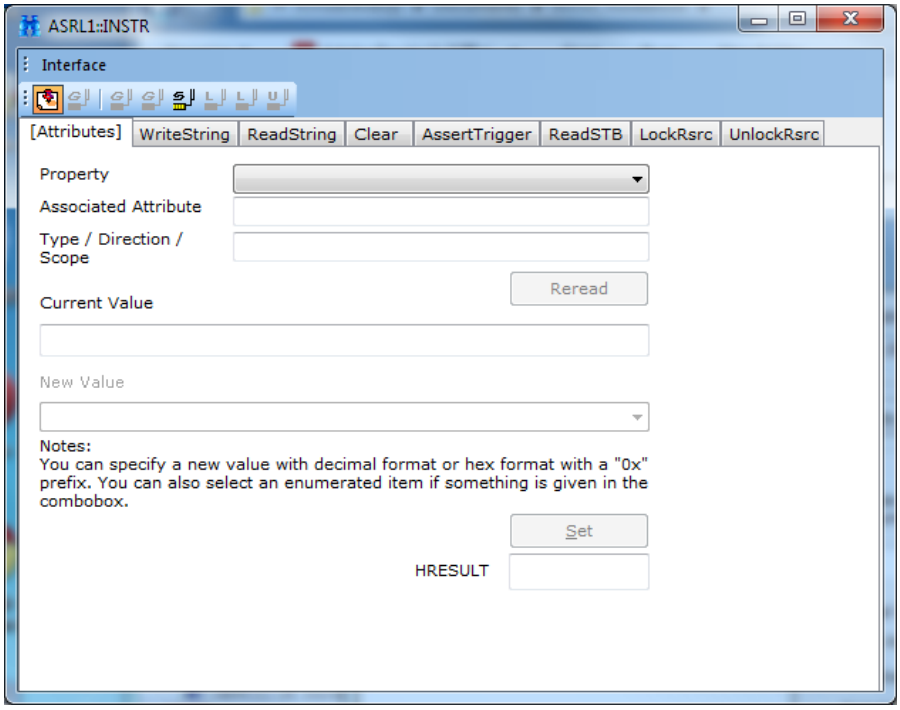

Figure 4-10 対話式制御(ASRL1::INSTR の例)

Interface メニューからは、オープンした VISA セッションに該当する VISA COM インターフェース(**IMessage**, **ISerial** 等)が選択できます。

画面上に複数表示されている全てのタブは、現在選択されている COM インターフェースで利用可能なメソッドと同じになっています。例えば **IMessage** という COM インターフェースが選択されている時には **WriteString** や **ReadString** メソッドが利用できます。

但し**[Attributes]**だけはメソッドでなく、該当 COM インターフェースで参照できるプロパティの一覧です。

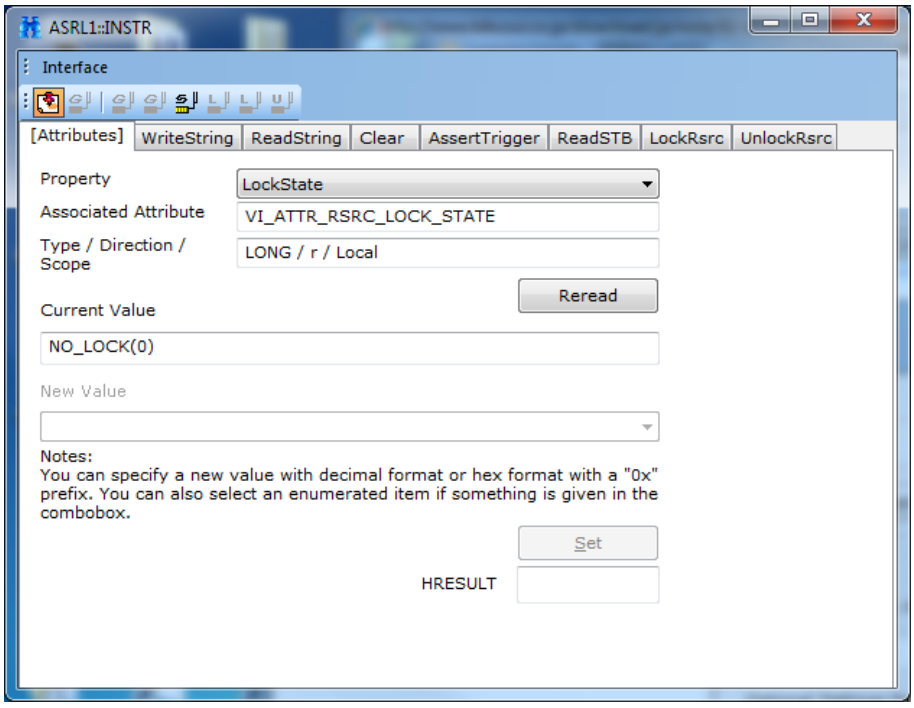

Figure 4-11 対話式制御([Attribute]ページ)

[Attribute]を選択した場合に **Property** コンボボックスに複数表示される項目は、全て該当インターフェースで設定又は読み取りの出来るプロパティです。**Associated Attribute** は

SetAttribute()又は **GetAttribute()**メソッドを使ってプロパティにアクセスする際のシンボル名です。(SetAttribute/GetAttribute は有効なメソッドですが、タブには表示されません。) **Type/ Direction/ Scope** はプロパティの型、方向、アクセス・スコープを示しています。型は LONG や SHORT など型を COM 仕様での標準的表現(Visual Basic 6 と同じ)で記載しています。方向は、r(読み込み)、w(書き込み)またはその両方です。アクセス・スコープは Local 又は Global です。Local は、該当 IO リソースに対して VISA セッション毎に独立して管理される属性値です。Global は、該当 IO リソースに対する複数 VISA セッション間で共通管理される属性値です。ある VISA セッションから Global 属性値を変更した場合、同じ IO リソースに接続する別の VISA セッションにも変更が波及しますが、ローカル属性ではそのような事はありません。

これより先は、実際に KI-VISA を使用して計測器 IO プログラミングを行う手順を説明します。説明で使用するコード・サンプルは C#言語(Visual Studio 2008)を前提としたものですが、他の言語でも基本的な手順は同じです。尚、プログラミング言語やツールに依存した内容(タイプ・ライブラリのインポート方法や、エラー処理ロジックなど)は、更に後のほうで言語別に説明します。またサービス・リクエスト通知ハンドラの処理方法は、言語ごとに大きく異なるので、これについても言語別パートで説明します。

5- タイプ・ライブラリのインポート

ここでは VISA COM ライブラリによって提供されるタイプ・ライブラリをインポートする方法について言語別に手順を説明します。

タイプ・ライブラリ(TypeLib, TLB 等と略される)とは、COM サーバが提供するインターフェース定義(型情報、メソッドとプロパティの文法、シンボル定数など)を格納したライブラリ・モジュールです。COM サーバを作成する際、全てのインターフェース定義は IDL(Interface Definition Language)という言語で記述されますが、TLB はこれを MIDL コンパイラで変換したものです。従って、ライブラリといってもインターフェース定義が含まれるだけなので、プログラムとしての実行コードはありません。タイプ・ライブラリは通常、.tlb ファイルとして単独に用意されるか、又はそのイメージをリソースとして.dll や.exe に埋め込んだ形で提供されます。

.NET Framework を必要とするマネージド言語(Visual Basic、C#、C++/CLI 等)の場合は、VISA COM モジュールに関するタイプ・ライブラリは、プライマリ・インタロップ・アセンブリ(Primary Interop Assembly、略して PIA)として提供されます。インタロップ・アセンブリは、本来ネイティブコードで実装されている COM コンポーネントのタイプ・ライブラリを.NET 環境から使うためのラッパーです。その中でも「プライマリ」に格付けされているものは、グローバル・アセンブリ・キャッシュ(Global Assembly Cache, 略して GAC)に、あたかも OS の一部のように登録されています。

.NET 言語で VISA COM を利用する場合は、VISA COM タイプ・ライブラリを直接参照するのではなく VISA COM の PIA を参照する事になります。

> Notes:
>
> GAC に登録されているアセンブリは、Windows Explorer で C:/Windows/Assembly ディレクトリを参照することで確認できます。但し、Assembly ディレクトリに表示されるファイルのレイアウトは、Explorer のシェル拡張によって実際の HDD 内の物理レイアウトとは違った物にすり替えられています。

5-1 Visual Studio 2008 (マネージド言語)での参照設定と名前空間の指定

Visual Studio 2008 を起動し、Visual Basic、C#、C++/CLI の何れかの言語でプロジェクトを作成して下さい。プロジェクトのタイプはフォーム・アプリかコンソール・アプリが良いでしょう。

参照設定

それぞれの言語でプロジェクトの統合環境が起動したら、メニューバーから**プロジェクト➔ 参照の追加**を選択し、下の図のような**参照の追加**ダイアログを表示します。

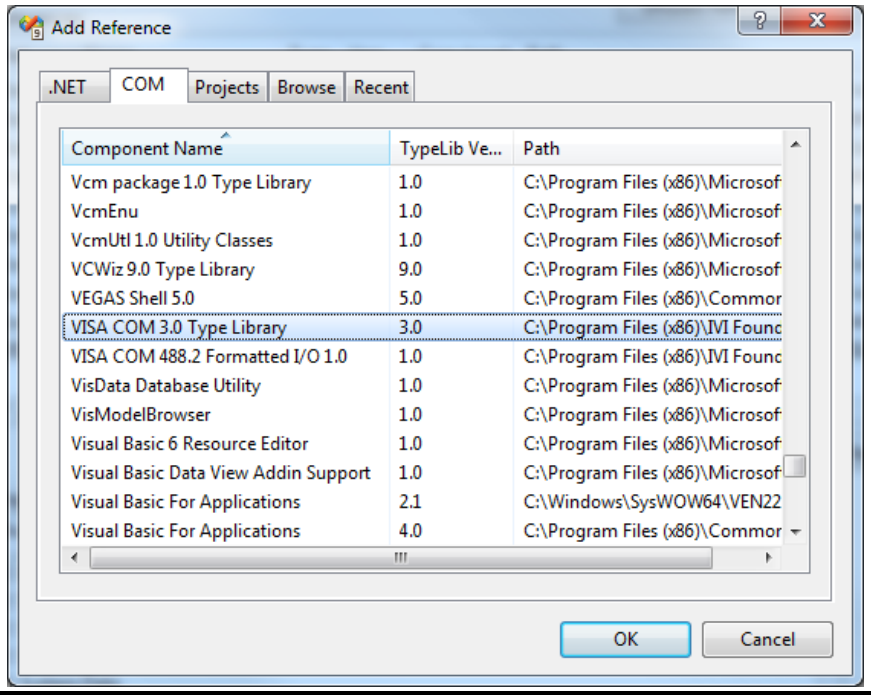

Figure 5-1 参照設定ダイアログ(Visual Studio.NET)

そこで、**COM** タブに表示されるリストから **VISA COM 3.0 Type Library** を選択して下さい。見つからない場合は、**参照**タブから Ivi.Visa.Interop.dll を直接指定して下さい。このファイルは通常 <VXIPNPPATH>/VisaCom/Primary Interop Assemblies デイレクトリに置かれています。

この設定は、一度行ったらそのプロジェクト内では再度行う必要はありません。しかし、別のプロジェクトでは別途同じ操作をする必要があります。

名前空間の指定

VISA COM ライブラリの名前空間は Ivi.Visa.Interop です。`Imports` 又は `using` 文を使ってそれを指定します。

```
//--------------------
// C#
//--------------------

using Ivi.Visa.Interop;
```

```
' --------------------
' Visual Basic.NET
' --------------------

Imports Ivi.Visa.Interop
```

```
//--------------------
//C++/CLI
//--------------------

using namespace Ivi::Visa::Interop;
```

Notes:

.NET 言語では、ターゲット・プラットフォームを指定する事により、x86 と x64 のどちらのプログラムも作成できます。ターゲットを明示していせずに Any とした場合は、アプリケーションの実行時に OS 環境に合わせて適宜ビットネスが設定されます。

5-2 Visual Basic 6.0 での参照設定

VB6 を起動し標準 EXE のプロジェクトを作成して下さい。次にメニューバーから**プロジェクト➔ 参照設定**を選択し、下の図のような**参照設定**ダイアログを表示します。

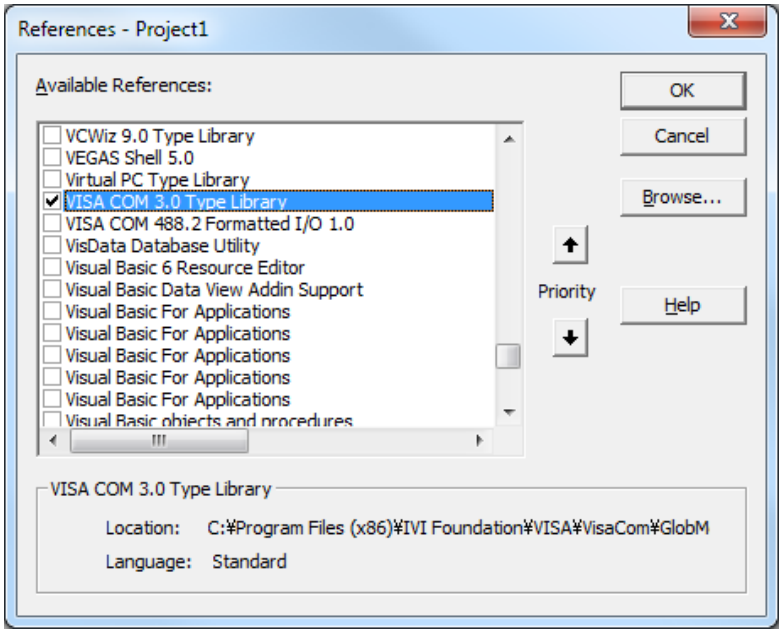

Figure 5-2 参照設定ダイアログ(VB6)

そこで、VISA COM 3.0 Type Library を選択してチェックして下さい。あとは **OK** ボタンをクリックするだけです。リスト中に見つからない場合は、**参照...**ボタンをクリックし、GlobMgr.DLL を探して下さい。このファイルは通常<VXIPNPPATH>/VisaCom ディレクトリに置かれています。

この設定は、一度行ったらそのプロジェクト内では再度行う必要はありません。しかし別のプロジェクトでは別途同じ操作をする必要があります。参照設定はプロジェクトごとに管理されています。

Notes:
VB6 では 32 ビットのプログラムのみを作成できますが、x64 環境の WOW64 上でも動作します。

5-3 Excel 2007 VBA での参照設定

Excel VBA の場合は、まず Visual Basic Editor の環境を用意しなければなりません。

まずリボンメニューに**[開発]**タブが表示されているかどうか確認して下さい。無い場合には次の手順で表示します。(既に表示されている場合は読み飛ばして下さい。)

- Microsoft Office ボタン(左上の )をクリックし、[Excel のオプション]をクリックします。
- **[基本設定]** カテゴリの **[Excel の使用に関する基本オプション]** で、**[[開発] タブをリボンに表示する]** チェックボックスをオンにし、**[OK]**をクリックします。

[開発]タブが表示されたら、次にセキュリティレベルを一時的に低く変更します。

- **[開発]** タブの **[コード]** で、**[マクロ セキュリティ]** をクリックします。
- **[マクロの設定]** カテゴリの **[マクロの設定]** で、**[すべてのマクロを有効にする (推奨しません。危険なコードが実行される可能性があります)]** をクリックし、**[OK]** をクリックします。

これで Excel VBA が使用できるようになりました。**[開発]** タブの **[コード]** で、**[Visual Basic]** をクリックして Visual Basic の環境を起動して下さい。VB6 の場合と異なり Excel VBA ではデフォルトでフォームが 1 つもありません。しかし VISA COM は ActiveX コントロールではないので、フォームを使う事は必須条件ではありません。もし Excel VBA 環境でフォームを使用したい場合は、Visual Basic Editor 環境のメニューから**挿入➔ ユーザーフォーム**を選択し、フォームを追加します。

参照設定を行うには、メニューバーから**ツール ➔ 参照設定**を選択し、下の図のような**参照設定** ダイアログを表示します。そこで、VISA COM 3.0 Type Library を選択してチェックして下さい。あとは **OK** ボタンをクリックするだけです。リスト中に見つからない場合は、**参照...**ボタンをクリックし、GlobMgr.DLL を探して下さい。このファイルは通常<VXIPNPPATH>/VisaCom ディレクトリに置かれています。

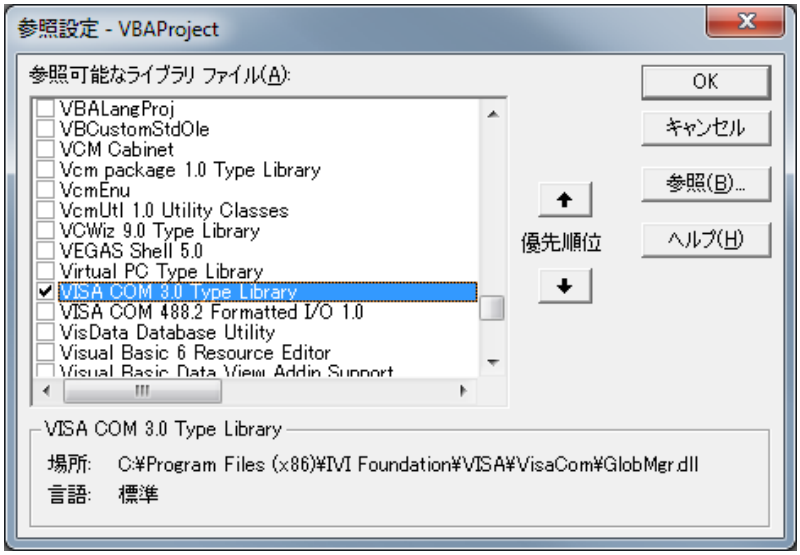

Figure 5-3 参照設定ダイアログ(Excel 2007 VBA)

この設定は、一度行ったらその VBA プロジェクト(Excel ワークブック)内では再度行う必要はありません。しかし、別の Excel ワークブックでは別途同じ操作をする必要があります。参照設定はワークブックごとに管理されています。

Notes:

Excel 2007 に搭載されている VBA は Visual Basic 6.5 と刻印されています。VB6 と同様、Excel は 32 ビットのプログラムですが、WOW64 上でも動作します。

5-4 Visual C++ 2008(アンマネージド)での参照設定

アンマネージドの Visual C++(MFC や ATL 等を使う場合)では.NET 言語のような参照設定を統合環境内から行う事はありません。その代わりに、下記のような`#import` 擬似命令を stdafx.h 内の適当な場所に書きます。

```
// x64 OS で、x64 アプリケーションを開発する場合
#import  "C:/Program Files/IVI Foundation/VISA/VisaCom64/GlobMgr.DLL"
no_namespace named_guids
```

```
// x64 OS で、x86 アプリケーションをクロス開発する場合
#import  "C:/Program Files (x86)/IVI Foundation/VISA/VisaCom/GlobMgr.DLL"
no_namespace named_guids
```

```
// x86 OS で、x86 アプリケーションを開発する場合
#import  "C:/Program Files/IVI Foundation/VISA/VisaCom/GlobMgr.DLL"
no_namespace named_guids
```

Notes:

実際のコードでは、上記の擬似命令は改行せずに一行で書いて下さい。

GlobMgr.DLL の実際のロケーションに合わせてパスの指定を変更して下さい。このファイルは通常 <VXIPNPPATH>/VisaCom ディレクトリに置かれていますが、Visual Studio でアプリケーション開発をする際の OS、及びターゲットプラットフォームの OS で、ディレクトリが微妙に異なるので注意して下さい。

これから先は、具体的なプログラムのコーディング方法について説明していきます。代表的に C#を例にします。

6- VISA セッションのオープン

6-1 リソース・マネージャ・オブジェクトの作成

VISA COM ライブラリを使うアプリケーションで最初にしなければならない事は、リソース・マネージャ・オブジェクトを作成する事です。リソース・マネージャ・オブジェクトは下の様に記述する事で新規に作成する事ができます。

```
IResourceManager3 rm = new ResourceManagerClass();
```

ここで、変数 rm は IResourceManager3 というインターフェース型で宣言されていますが、右辺で新規作成されるのは ResourceManagerClass コンポーネントです。最後に()が付いているのは、そのコンポーネント型のコンストラクタを呼び出している事を意味します。

リソース・マネージャの最も重要な仕事は、指定された VISA アドレスを元に VISA セッションをオープンする事です。VISA セッションをオープンしなければ計測器との IO は何も始まらないからです。

リソース・マネージャの実体は、<VXIPNPPATH>/VisaCom ディレクトリに置かれた Global Resource Maneger (略して GRM)という COM コンポーネント(GlobMgr.dll)です。GRM は IVI Foundation が提供する VISA Shared Components の一部で、実際の IO 処理は全く行いません。しかし GRM はシステムに登録されている複数のベンダー提供コンポーネントを階層的に取り扱う機能を持っています。

6-2 VISA セッションのオープン

リソース・マネージャ・オブジェクトが作成できたら、Open メソッドを使って VISA セッションをオープンします。

```
IMessage msg =
   (IMessage)rm.Open("GPIB0::3::INSTR", AccessMode.LOAD_CONFIG, 0, "");

...

msg.Close();
```

まず、オープンした VISA セッションの参照を保持するための変数を宣言しておきます。ここでは IMessage インターフェース型の変数 msg を宣言します。型名の先頭に大文字の"I"が付いている場合、一般にそれはインターフェース型になっています。この場合は IMessage の"I"です。

コンポーネント・クラス型と COM インターフェース型の違いを理解するのは少々難しいのですが、一般にコンポーネント・クラス型は、new 演算子でオブジェクトのインスタンスを作成できる型です。先の例では、ResourceManagerClass コンポーネントがそうです。それに対し、インターフェース型はオブジェクトのインスタンスを作成する事はできません。それは、コンポーネント・クラスと異なりインターフェース自体は抽象的なものであり、何ら実体を伴わないからです。その代わりに、何か別の手段で作成されたオブジェクトのインスタンス(実体)に対し、特定のインターフェースを参照するために使います。

次に Open メソッドの呼び出し部分を見て下さい。厳密には Open メソッドが返す型は IVisaSession 型なのですが、ここではその派生型である IMessage 型の変数でそれを受取る事にします。つまりここでは、リソース・マネージャ・オブジェクトの Open メソッドを呼び出し、その戻り値を IMessage 型 COM インターフェースとして変数 msg に代入しています。(この場合タイプキャストが必要です。)呼び出しに成功すれば、以降 IMessage インターフェースで提供される全てのメソッドとプロパティを利用できます。

ここでリソース・マネージャの Open メソッドについて簡単に説明しておきます。

```
IVisaSession IResourceManager3.Open(
   string ResourceName,
   AccessMode mode,
   int initTimeoutm
   string OptionString
);
```

Open メソッドに渡している文字列 *ResourceName* は VISA アドレスと呼ばれるもので、特定の IO リソースを指し示す名前になっています。VISA アドレスの書式は VISA の仕様書で明確に定められており、GPIB、シリアル、USB、LAN の場合は下記のような構文になっていなければなりません。

GPIB[*board*]::*PrimaryAddress*[::*SecondaryAddress*][::INSTR]

GPIB[*board*]::INTFC

ASRL[*board*][::INSTR]

USB[*board*]::*VendorID*::*ProductID*::*SerialNumber*[::*InterfaceNumber*][::INSTR]

TCPIP[*board*]::*IPAddress*[::*devicename*][::INSTR]

TCPIP[*board*]::*IPAddress*::*devicename*[::INSTR]

TCPIP[*board*]::*IPAddress*::*portno*::SOCKET

[]で囲まれた部分は省略が可能です。上記のプログラム例では"GPIB0::3::INSTR"というリソース名を使っているので、これは GPIB0 のバス上に接続されたプライマリ・アドレス 3、セカンダリ・アドレスなし、の計測器を示します。

下の表は比較的良く使われるリソース文字列の例を示したものです。

Table 6-1　　リソース文字列の例

リソース文字列	**説明**
GPIB0::3::INSTR	GPIB0 に接続されたプライマリ・アドレス 3 の計測器
GPIB1::3::12::INSTR	GPIB1 に接続されたプライマリ・アドレス 3、セカンダリ・アドレス 12 の計測器
GPIB0::INTFC	GPIB0 のコントローラ・インターフェース
ASRL1::INSTR	シリアル・ポート COM1 に接続された計測器
USB0::0x0B3E::0x1005::ABC12345::INSTR	ベンダーID(VID)2878、プロダクト ID(PID)4101、シリアル・ナンバー"ABC12345"を持つ USBTMC 計測器
TCPIP0::192.168.1.23::inst0::INSTR	IP アドレス 192.168.1.23 の VXI-11 計測器
TCPIP0::192.168.1.23::hislip0::INSTR	IP アドレス 192.168.1.23 の HiSLIP 計測器
TCPIP0::192.168.1.23::5025::SOCKET	IP アドレス 192.168.1.23、ポート番号 5025 の SCPI-RAW 計測器

Notes:

[board]を省略した場合、0 を指定したものとして扱われます。従って"GPIB"は"GPIB0"と解釈されます。USB の場合[board]の値は特に意味を持たないので、通常は 0 にして下さい。

[INSTR]を省略する事はできますが、INTFC は省略できません。

GPIB セカンダリ・アドレスには 0～30 を指定して下さい。システムによってはセカンダリを 96～126 で表現しているものもありますが、VISA では 0～30 です。実際のアドレッシング・コマンド送出時には 60(hex)が加算されたものが使用されます。

USB の VID/PID 部分は 16 進と 10 進のどちらでも指定できます。ただし 16 進書式は C 言語スタイルと同じ 0x が手前に必要です。

LAN の IP アドレス部分は有効なホスト名でも指定できます。(有効なホスト名は、計測器の埋め込み WEB ページで確認できます。)

VISA アドレスは、ケース・インセンセィティブ(大文字小文字の区別なし)です。

Open メソッドの 2 番目以降のパラメータについて説明しましょう。*mode* パラメータはリソース・ロッキングの為のアクセス・モードを指定します。リソース・ロッキングについては後述します。通常は NO_LOCK か LOAD_CONFIG を指定します。パラメータ *initTimeout* は、同じリソースが他のセッションで既にロックされている際にロックを獲得できるまでのタイムアウト(ms)を指定します。*mode* パラメータに EXCLUSIVE_LOCK 又は SHARED_LOCK を指定した場合のみ意味を持ちます。*OptionString* パラメータは、このセッションに対して設定可能な属性値を一括指定できる便利な機能です。属性は個別にも指定できるので、ここでは省略又はブランク文字列を指定します。

Open メソッドで VISA セッションをオープンしたら、それ以降そのセッションを通じて IO を行う事が出来ます。全ての IO 動作を終了して VISA セッションが不要になったら、Close メソッドを呼び出して下さい。

```
void IMessage.Close();
```

Notes:

IResourceManager3.Open メソッドの戻り値は、厳密には IVisaSession インターフェースへの参照となっています。従ってこれを受取る変数は本来であれば IVisaSession 型でなければなりません。しかし、"INSTR"を含むリソース文字列をリソース・マネージャに指定して作成された VISA セッション・オブジェクトは、必ず IMessage インターフェースをサポートしています。従って厳密な戻り値の型ではなく IMessage インターフェースへの参照として取得する事ができます。但し、明示的なタイプキャストをする必要があります。

VISA アドレスのタイプが GPIB INSTR の場合、Open メソッドの呼び出しに成功すれば下のサンプルに示される COM インターフェースを全て参照することが出来ます。

```
IVisaSession vi;
vi = rm.Open( "GPIB0::3::INSTR", AccessMode.NO_LOCK, 0, "");

IBaseMessage bmsg  = (IBaseMessage)vi;

IAsyncMessage amsg = (IAsyncMessage)vi;

IMessage msg = (IMessage)vi;

IEventManager evm = (IEventManager)vi;

IGpib g = (IGpib)vi;

(略)

vi.Close();
```

VISA アドレスのタイプが ASRL INSTR の場合、Open メソッドの呼び出しに成功すれば下のサンプルに示される COM インターフェースを全て参照することが出来ます。

```
IVisaSession vi;
vi = rm.Open( "ASRL1::INSTR", AccessMode.NO_LOCK, 0, "");

IBaseMessage bmsg  = (IBaseMessage)vi;

IAsyncMessage amsg = (IAsyncMessage)vi;

IMessage msg = (IMessage)vi;

IEventManager evm = (IEventManager)vi;

ISerial s = (ISerial)vi;
```

VISA アドレスのタイプが GPIB INTFC の場合、Open メソッドの呼び出しに成功すれば下のサンプルに示される COM インターフェースを全て参照することが出来ます。

```
IVisaSession vi;
vi = rm.Open( "GPIB0::INTFC", AccessMode.NO_LOCK, 0, "");

IGpibIntfc gi = (IGpibIntfc)vi;

IGpibIntfcMessage gim = (IGpibIntfcMessage)vi;
```

VISA アドレスのタイプが USB INSTR の場合、Open メソッドの呼び出しに成功すれば下のサンプルに示される COM インターフェースを全て参照することが出来ます。

```
IVisaSession vi;
vi = rm.Open(
   "USB0::0x0B3E::0x1005::ABC12345::INSTR", AccessMode.NO_LOCK, 0, "");

IBaseMessage bmsg  = (IBaseMessage)vi;

IAsyncMessage amsg = (IAsyncMessage)vi;

IMessage msg = (IMessage)vi;

IEventManager evm = (IEventManager)vi;

IUsb u = (IUsb)vi;
```

VISA アドレスのタイプが LAN INSTR の場合、Open メソッドの呼び出しに成功すれば下のサンプルに示される COM インターフェースを全て参照することが出来ます。

```
IVisaSession vi;
vi = rm.Open("TCPIP0::192.168.1.23::INSTR", AccessMode.NO_LOCK, 0, "");

IBaseMessage bmsg  = (IBaseMessage)vi;

IAsyncMessage amsg = (IAsyncMessage)vi;

IMessage msg = (IMessage)vi;

IEventManager evm = (IEventManager)vi;

ITcpipInstr t = (ITcpipInstr)vi;

. . .

vi.Close();
```

Notes:

COM インターフェースから別の COM インターフェースへのタイプキャストを伴う参照は、COM DLL の内部実装では、QueryInterface()の呼び出しを伴っています。言語記述上の単純な型変換ではなく、インターフェース参照に関する問い合わせを DLL 自体に問い合わせを行っています。

複数の COM インターフェース型変数で参照を保持するとその数だけ COM オブジェクトの参照カウントは増加します。これは QueryInterface されるたびにオブジェクト自身が AddRef を呼び出して参照カウントを増加するからです。逆に、その変数がスコープを失ったときには Release を呼び出します。しかし変数の数に関わらず、オブジェクトのインスタンス(実体)は同じものです。(上記最後の LAN INSTR の記述例では、IVisaSession 以外の 5 つのインターフェース型変数でそれぞれ vi を参照していますが、オブジェクトのコピーを 5 個作ったわけではなく、オブジェクトの実体は依然 1 個です。)

言い換えると、複数の COM インターフェース型変数で幾通りもの参照を行なっても、VISA セッション自体は一つです。従って最後に Close メソッドを呼び出すのはセッションに対して一度で十分です。COM インターフェース型の変数ごとに何度も Close を呼び出してはなりません。

7- 基本的な IO

まずは、下記のコードを見て下さい。

```
IResourceManager rm = new ResourceManagerClass();

string strVisaAddress = "GPIB0::5::INSTR";
// 必要に応じて下記のような VISA アドレスに置き換えて
//string strVisaAddress = "ASRL1::INSTR";
//string strVisaAddress = "USB0::0x0B3E::0x1005::ABC12345::INSTR";
//string strVisaAddress = "TCPIP0::192.168.1.23::INSTR";

IMessage msg = (IMessage)rm.Open(
   strVisaAddress, AccessMode.NO_LOCK, 0, "Timeout = 3000");

msg.TerminationCharacter = 10;
msg.TerminationCharacterEnabled = false;
msg.SendEndEnabled = true;


IGpib gpib;
ISerial seri;
ITcpipInstr tcp;
IUsb usb;

switch( msg.HardwareInterfaceType) {
case 1:
    //GPIB
    gpib = (IGpib)msg;
    gpib.RepeatAddressingEnabled = true;
    gpib.UnaddressingEnabled = false;
    break;
case 4:
    //ASRL
    seri = (ISerial)msg;
    seri.BaudRate = 19200;
    seri.DataBits = 8;
    seri.StopBits = SerialStopBits.ASRL_STOP_TWO;
    seri.Parity = SerialParity.ASRL_PAR_NONE;
    seri.FlowControl = SerialFlowControl.ASRL_FLOW_XON_XOFF;
    break;
case 6:
    //LAN
    tcp = (ITcpipInstr)msg;
    break;
case 7:
    //USB
    usb = (IUsb)msg;
    break;
}

msg.TerminationCharacterEnabled = true;

int r = msg.WriteString( "*IDN?\n");
string rd = msg.ReadString(256);

msg.Close();
```

このサンプルは、GPIB インターフェース (或いはシリアル・インターフェース、USB インターフェース、LAN インターフェース等に) に接続された計測器の VISA セッションをオープンし、インターフェース・

タイプによって異なる処理を行い、"*IDN?"クエリを改行コード付きで送信し、そしてそのレスポンスを受取るだけの単純なプログラムです。

7-1 オプション文字列

Open メソッドについては既に説明しましたが、最後の"Timeout = 3000"という文字列に注目して下さい。このパラメータは Open メソッドの説明で *OptionString* として登場しました。必要が無ければブランク文字列を渡しても構わないのですが、この例のように設定可能な VISA 属性値を代入式のような構文で指定する事もできます。複数の属性値を一度に指定する場合はセミコロンで区切ります。また、これら属性値は、プロパティに値を書き込む事でも同様の設定を行う事が出来ます。ここで指定された"Timeout = 3000"は、IO タイムアウト(ms)です。VISA 仕様によりデフォルトでは 2000ms ですが、必要に応じて変更して下さい。Open メソッドに *initTimeout* という時間設定パラメータがありましたが、それは VISA セッションのオープンと同時にリソース・ロッキングを行う際の、ロック獲得までのタイムアウト指定です。IO タイムアウトと間違えやすいので注意して下さい。

下に示されるのは、Open メソッド呼び出し後に OptionString プロパティから読み出したものです。設定値はともかく、表現方法としては全て Open メソッドの *OptionString* パラメータに渡せる有効なものです。この中から必要な項目だけを絞って *OptionString* パラメータに渡すことができます。各項目は代入式のような表現で、項目同士の連結はセミコロンで行ないます。記号の両側には必ずスペースを開けて下さい。

```
Timeout = 2000 ; SendEndEnabled = TRUE ; TerminationCharacter = 10 ;
TerminationCharacterEnabled = FALSE ; MaximumQueueLength = 16 ; BaudRate =
9600 ; DataBits = 8 ; DataTerminalReadyState = STATE_UNKNOWN; EndIn =
ASRL_END_TERMCHAR; EndOut = ASRL_END_NONE; Parity = ASRL_PAR_NONE;
RequestToSendState = STATE_UNKNOWN; StopBits = ASRL_STOP_ONE;
ReplacementCharacter = 0 ; XONCharacter = 17 ; XOFFCharacter = 19
```

先ほどのサンプルでは、GPIB, Serial, USBTMC、LAN、それぞれの異なる IO インターフェース・タイプに対して基本的な IO の手順を説明しました。その中には IO インターフェース・タイプ毎に異なった書き方をしなければならない部分と共通な部分とが共存してます。

まず第一に、VISA は異なる IO インターフェースに対して全く同じプログラムが動作する事を保証しているわけではないという事を理解して下さい。VISA ライブラリの利点は IO リソースタイプに依存しない共通の API が提供されている事ですが、状況によってはインターフェース毎に違った処理が必要な場合もあるのです。

例えば、GPIB インターフェースにはリモート・ローカルや EOI 信号、サービス・リクエスト等の固有概念があったり、逆にシリアル・インターフェースにはこれらが無くボーレート等の概念が存在します。これらを上手く設定しないと、インターフェース固有の処理は行えません。ボーレートに関する属性値等はシリアルの場合には適切に設定しないと計測器との通信さえ出来ませんが、GPIB 接続の時にこれら属性を不用意に設定しようとすると例外が発生してしまいます。

作成する計測制御アプリケーションがもしこれら異なる IO インターフェース・タイプを透過的に扱いたいのであれば、ここで説明する手法を知っておいたほうが良いでしょう。

7-2 IO インターフェース・タイプの識別

上の例では、switch/case 文を使って、IO インターフェースによって異なる処理をいくつか行っています。しかし、このような IO インターフェース依存のアプローチを取らなければならないのはほんのわずかです

全ての VISA セッションでは、HardwareInterfaceType プロパティを使用する事で、IO インターフェース・タイプを識別する事ができます。またその識別を行う行為自体はインターフェース・タイプには依存しません。

short IVisaSession.HardwareInterfaceType;

このプロパティはリード・オンリーです。GPIB では 1、ASRL では 4、LAN では 6、USB では 7 を返します。この値を元に、上記サンプルの switch/case 文のような条件分岐処理を記述する事ができます。

IGpib インターフェース(タイプ 1)

VISA セッションが GPIB INSTR リソースに対してオープンされている場合、セッション・オブジェクトは IMessage インターフェースの他に IGpib インターフェースを同時にサポートしています。従ってタイプキャストを伴えば IGpib インターフェースの参照を取得する事ができます。IGpib インターフェースへの参照を獲得できれば、GPIB 固有の詳細処理を行う事ができます。

```
IGpib gpib = (IGpib)msg;
gpib.RepeatAddressingEnabled = true;
gpib.UnaddressingEnabled = false;
```

ここでは、リピート・アドレッシングとアンアドレッシングの設定をそれぞれ true、false に設定しています。リピート・アドレッシングは同じ方向の IO が連続する場合、2 度目以降の IO に対してアドレッシング・コマンドの再送出を行うかどうかを指定します。アンアドレッシング機能は、ATN FALSE メッセージが処理されたあと、再度 ATN TRUE の UNT(hex 5F)と UNL(hex 3F)を送出する機能です。GPIB で VISA セッションをオープンした直後はデフォルトで上記サンプルコードの記述と同じ設定になっているので、特に明示的に設定する必要はありません。しかし、これらの設定を変更する事も可能である事を覚えておいて下さい。IGpib インターフェースの各メソッド・プロパティの詳細については KI-VISA COM オンライン・ヘルプを参照して下さい。

ISerial インターフェース(タイプ 4)

VISA セッションが ASRL INSTR リソースに対してオープンされている場合、セッション・オブジェクトは IMessage インターフェースの他に ISerial インターフェースを同時にサポートしています。従って Set ステートメントによって ISerial インターフェースの参照を取得する事ができます。ISerial インターフェースへの参照を獲得できれば、シリアル・インターフェース固有の詳細処理を行う事ができます。

```
seri = (ISerial)msg;
seri.BaudRate = 19200;
seri.DataBits = 8;
seri.StopBits = SerialStopBits.ASRL_STOP_TWO;
seri.Parity = SerialParity.ASRL_PAR_NONE;
seri.FlowControl = SerialFlowControl.ASRL_FLOW_XON_XOFF;
```

シリアル・インターフェースの場合は固有設定項目がやや多くなります。BaudRate から FlowControl までの設定は特に説明の必要も無いでしょう。これらの値は通常は計測器側の設定と一致させれば問題ありません。

上記サンプルコードの記述は VISA のデフォルト値ではありませんが、当社製計測器の RS232-C インターフェースで最もよく使われる設定です。

Notes:

VISA セッションをシリアル・インターフェースに対してオープンする際に、LOAD_CONFIG オプションを指定すると上記サンプルで設定された最初の 5 個のプロパティは VISA 仕様によるデフォルト値ではなく、KI-VISA Instrument Explorer の **Optional Settings For RS232 Serial Interface** でコンフィグレーションした値がデフォルトになります。従って LOAD_CONFIG を指定することで、シリアル・ポートの詳細設定をプログラムコードから除去することでもできます。

IUsb インターフェース(タイプ 7)

IUsb インターフェースでは、USB で実際に使用するエンドポイントの情報や、コントロールエンドポイントを直接操作してベンダー依存リクエストなどを送ることもできます。但し、現時点でそのような操作を必要とする USBTMC 計測器は殆どありません。

ITcpipInstr, ITcpipSocket, IHislipInstr インターフェース(タイプ 6)

ITcpipInstr/ITcpipSocket/IHislipInstr インターフェースでは、接続された計測器のネットワーク接続に関する詳細情報を取得することができます。しかし、計測器として制御する際には特に意識する必要は無いでしょう。

7-3 計測器との通信

IO インターフェース・タイプ固有の処理が適切に行われたあとは、Write、Read、WriteString、ReadString メソッドを使って実際に計測器との通信を行ないます。IMessage インターフェースで最もよく使われるのは、ASCII ベースのメッセージを処理する WriteString と ReadString メソッドです。この 2 つのメソッドがあれば SCPI コマンドのような文字列ベースののの IO 作業は殆どすべて出来てしまうでしょう。上記の例では、SCPI コマンド対応の計測器に"*IDN?"クエリを改行コード付きで送り、その応答を受取っています。下の表に示されるのは、IMessage インターフェースでよく使われるメソッドです。

Table 7-1　　計測器との通信でよく使われるメソッド (IMessage インターフェース)

メソッド	説明
WriteString	ASCII ベースのメッセージを計測器に送信する
ReadString	ASCII ベースのレスポンス･メッセージを計測器から受信する
Write	バイナリ・メッセージを計測器に送信する
Read	バイナリ・レスポンス･メッセージを計測器から受信する
Clear	セレクテッド・デバイス・クリア(SDC)を計測器に送信する(GPIB、USB)
AssertTrigger	デバイス・トリガ(GET)を計測器に送信する(GPIB、USB)
ReadSTB	シリアル･ポーリングを実行してステータス･バイトを取得する(GPIB、USB)

ここでは最も使用頻度の高い WriteString と ReadString の説明をしておきましょう。その他のメソッドの詳細については VISA COM オンライン・ヘルプを参照して下さい。

int IMessage.WriteString(string *buffer*);

WriteString は文字列の送信を行ないます。*buffer* パラメータには送信したい文字列を渡します。戻り値は実際に計測器に送信されたメッセージバイト数です。

string IMessage.ReadString(int *count*);

ReadString は文字列の受信を行ないます。*count* パラメータは受信したいレスポンスの最大バイト数、戻り値は受信したレスポンス文字列です。計測器から返されるレスポンスのバイト数が count 指定値よりも短い場合、レスポンス全体を取得出来ます。計測器から返されるレスポンスのほうが長い場合、count 指定値の長さぶんだけが返されます。

Notes:

count 指定値が短すぎた為に取得しそこなった続きのレスポンスが残っている場合、再度の ReadString 呼び出しで残りのぶんを継続取得出来るかどうかは、状況によって異なります。GPIB、シリアル、LAN(ソケット、HiSLIP)の場合は継続取得できますが、USBTMC、LAN(VXI-11)の場合は、計測器側の出力バッファの振る舞いによって状況は違ってきます。

7-4 IO インターフェースによって扱いの異なるメッセージ・ターミネータ

WriteString と ReadString メソッドが前提としているメッセージ・ターミネータに関して少し触れておかなければなりません。メッセージ・ターミネータには幾つかの種類があり、計測器の IO 仕様や、IO インターフェースの種類によって様々です。またこの部分がうまく設定されていないためにデバイス・メッセージが計測器にうまく伝わらなかったり、或いはクエリに対するレスポンスをうまく受信できなかったりします。

GPIB の場合

GPIB 計測器の場合、メッセージ・ターミネータ(古くはデリミタなどとも呼ばれている)には、CR, LF, CR+LF, あるいはこれらに EOI(End Of Identify)信号が合わさったものが使われます。VISA ライブラリの場合、デフォルトでは EOI 以外は一切使われません。もちろんこれは EOI 以外の(LF 等の特定メッセージ・バイト)をターミネータとして利用できないという意味ではありません。デフォルトでは EOI のみであるという事です。

例えば WriteString("*IDN?")と書いた場合、GPIB では 5 バイトのメッセージを送信し、最終バイト(この場合は"?"文字、ASCII コード 0x3F)と同時に EOI 信号がアサートされます。改行コードが自動的に付加されることはありません。改行コードを必要とする場合は、送信する文字列に明示的にそれを含ませる必要があります。("*IDN?\n"等) しかし殆どの GPIB 計測器は EOI 信号を受信した際にターミネート処理を行なうので、改行コードを付加しなくても正常に通信できます。

"*IDN?"クエリに対する応答が"KIKUSUI,PLZ664WA,ABC12345,1.00\n"であると仮定しましょう。この場合、計測器は 31 バイト(表示可能な 30 バイトとメッセージターミネータの LF、ASCII コード 0x0A)を返そうとします。GPIB での ReadString メソッドは、デフォルトでは EOI 信号がターミネータに使われます。LF などの特定メッセージ・バイトでターミネートするのではありません。また有効なターミネータを検出していない場合でも、ReadString に渡された最大受信バイト数に達すると受信動作を終了します。

これが GPIB でのデフォルト動作です。ASCII ベースのコマンドで SCPI 対応の計測器(或いは EOI をサポートする全ての計測器)を扱うぶんには通常は特に設定を変更する必要は無いでしょう。しかし、EOI をターミネータとして扱わない古い計測器や文字列ベースでない Write メソッドや Read メソッドを使用してバイナリ転送を行う場合には、必要に応じてデフォルト動作を変更する必要があります。デフォルト動作を変更するには、TerminationCharacter、TerminationCharacterEnabled、SendEndEnabled の各プロパティを適切に設定変更する必要があります。これらはいずれも IMessage インターフェースで利用可能なプロパティです。

TerminationCharacter、TerminationCharacterEnabled の各プロパティは、受信動作(Read)に対して適用されます。送信動作(Write)には影響しません。

Table 7-2 GPIB のターミネータに有効なプロパティ(IMessage インターフェース)

プロパティ	タイプ	デフォルト値	IEEE488.2 計測器での推奨値
TerminationCharacter	Byte	0x0A	←
TerminationCharacterEnabled	Boolean	false	true
SendEndEnabled	Boolean	true	←

SendEndEnabled が true の場合、WriteString メソッドは、最終メッセージバイトと同時に EOI 信号をアサートします。CR や LF などのターミネータが自動的に付加されるわけではないので注意して下さい。逆に false に設定されている場合は EOI はアサートされないため、コマンド文字列中に明示的な LF コードなどが埋め込まれていない限り、計測器側ではターミネートされたとは解釈されません。TerminationCharacterEnabled が true の場合、ReadString メソッドは、EOI 受信又は指定された最大バイト数という条件に加えて、TerminationCharacter で指定された特定バイトをターミネータとして受信作業を終了します。TerminationCharacterEnabled と TerminationCharacter プロパティは ReadString にのみ効果があります。WriteString には何ら効果がないので注意して下さい。

ASRL(シリアル)の場合

RS232 などのシリアル・インターフェースを持つ計測器の場合、メッセージ・ターミネータには、CR, LF, CR+LF などがあります。GPIB の場合と異なり EOI は存在しません。また、シリアル・インターフェースには、ボーレート、データビット、ストップビット、パリティ、フロー制御など、固有の概念が存在します。VISA ライブラリの場合、シリアル・ポートの VISA アドレスに対するデフォルトでは、WriteString に関しては LF などの特定メッセージバイトの考慮は行われません。

例えば WriteString("*IDN?")と書いた場合、ASRL ではデフォルトでは 5 バイトのメッセージを送信するだけです。EOI 信号をアサートする術もありません。殆どの場合シリアル・インターフェースを持つ計測器は、これでは応答しません。

計測器がコマンドを正しくターミネートできる様にするには、送信するコマンドに明示的に改行コード明示的に含ませる必要があります。("*IDN?\n"等)

クエリに対するレスポンスを受信する場合も同様の問題がありますが、殆どの計測器はレスポンスに明示的な改行コードを付加してきます。VISA の ReadString()でそれを正しく受信してターミネートするには、TerminationCharacterEnabled プロパティを True にセットしておく必要があります(デフォルトは False)。

TerminationCharacter、TerminationCharacterEnabled の各プロパティは、受信動作(Read)に対して適用されます。送信動作(Write)には影響しません。

Table 7-3　　シリアルのターミネータに有効なプロパティ(IMessage インターフェース)

プロパティ	タイプ	デフォルト値	IEEE488.2 計測器での推奨値
TerminationCharacter	Byte	0x 0A	←
TerminationCharacterEnabled	Boolean	false	**true**

ISerial インターフェースでは、さらに二種類のプロパティをターミネーション設定に使用することができます。

Table 7-4　　シリアルのターミネータに有効なプロパティ(ISerial インターフェース)

プロパティ	タイプ	デフォルト値	IEEE488.2 計測器での推奨値
EndOut	SerialEndConst	ASRL_END_NONE	明示的に改行コードを付けるなら ASRL_END_NONE 明示的に改行コードを付けないなら ASRL_END_TERMCHAR
EndIn	SerialEndConst	ASRL_END_TERMCHAR	←

EndOut はデフォルトでは ASRL_END_NONE に設定されていますが、ASRL_END_TERMCHAR に変更すると、WriteString でのメッセージ送信時に、TerminationCharacter で指定されたメッセージ・バイトを付加して送信します。EndIn はデフォルトで ASRL_END_TERMCHAR に設定されていますが、ReadString メソッドは、指定された最大バイト数という条件に加えて、TerminationCharacter で指定された特定バイトをターミネータとして受信作業を終了します。TerminationCharacterEnabled プロパティは False のまま放置しておいても問題ありません。EndOut プロパティは WriteString に効果があり、EndIn プロパティは ReadString に効果があります。

Notes:

EndOut を ASRL_END_NONE に設定する事で、WriteString()での動作で改行コードを暗黙に追加送信する事ができます。しかし、ASRL 同様に EOI 機能を持たない TCP/IP ソケット(SCPI-RAW)の場合には送信文字列に明示的に改行コードを含ませる以外にそれを付加させる手段はありません。従って通信 IF に依らず共通のアプリケーション・ソフトを作成する場合、EndOut に頼らず WriteString()の呼び出し毎に明示的に改行コードを含ませるようにした方が良いでしょう。

USB の場合

USB(USBTMC)計測器の場合、メッセージ・ターミネータには、CR, LF, CR+LF, あるいはこれらに EOM(End Of Message)フィールドが合わさったものが使われます。EOM は GPIB の EOI 信号をを

模倣して設計されたものです。GPIB の場合とは異なり USB には EOM を表現するための専用ラインは有りませんが、メッセージの先頭に特殊なヘッダ部を設け、そこに EOM ビットというフィールドを持たせるようにしています。意味合い的には GPIB の EOI と全く同じと考えてよいでしょう。

Notes:

KI-VISA で USB インターフェースを装備する計測器を制御したい場合、計測器は USBTMC 仕様に準拠している必要があります。KI-VISA では USBTMC 仕様でない任意の USB 計測器を制御できる USB RAW セッションは提供していません。

USBTMC が使用する特殊なヘッダは、ホストからの送信時に使われるものは USBTMC バルクアウト・ヘッダと呼ばれ、計測器からの受信時に使われるものは USBTMC バルクイン・ヘッダと呼ばれます。EOM フィールドはこれらヘッダ内に存在します。USBTMC に関する詳細は、USBTMC および USBTMC-USB488 仕様書を参照して下さい。

USBTMC ヘッダの処理は VISA 又はデバイス･ドライバが行ないます。アプリケーションが特に意識する必要はありません。また、Write メソッドなどが返す転送バイト数情報にもヘッダ部分のサイズは含まれません。

VISA ライブラリの場合、デフォルトでは EOM 以外は一切使われません。もちろんこれは EOM 以外の(LF 等の特定メッセージバイト)をターミネータとして利用できないという意味ではありません。デフォルトでは EOM のみであるという事です。

先のプログラムの例 WriteString("*IDN?\n")では改行コードを含む 6 バイトのメッセージを送信し、この最終バイト(この場合は LF 文字、ASCII コード 0x0A)を以ってメッセージが完了する(EOM)という意思を計測器側に伝達します。

"*IDN?"クエリに対する応答が"KIKUSUI,PLZ664WA,BBB77700,1.00\n"であると仮定しましょう。この場合、計測器は 31 バイト(表示可能な 30 バイトとメッセージ・ターミネータの LF、ASCII コード 0x0A)を返そうとします。USB での ReadString メソッドは、デフォルトでは EOM フィールドがターミネータに使われます。LF などの特定メッセージ・バイトでターミネートするのではありません。また有効なターミネータを検出していない場合でも、ReadString に渡された最大受信バイト数に達すると受信動作を終了します。

これが USB でのデフォルト動作です。USBTMC は比較的新しいプロトコル(2003 年リリース)なので、この仕様に準拠する計測器は多くの場合 SCPI 仕様になっていると考えられます。ASCII ベースのコマンドで USBTMC 計測器を扱うぶんには通常は特に設定を変更する必要は無いでしょう。しかし、Write メソッドや Read メソッドを使用してバイナリ転送で受信を行う場合には、必要に応じてデフォルト動作を変更する必要があります。デフォルト動作を変更するには、TerminationCharacter、TerminationCharacterEnabled、SendEndEnabled の各プロパティを適切に設定変更する必要があります。これらはいずれも IMessage インターフェースで利用可能なプロパティです。

Table 7-5 USB のターミネータに有効なプロパティ(IMessage インターフェース)

プロパティ	タイプ	デフォルト値	IEEE488.2 計測器での推奨値
TerminationCharacter	Byte	0x 0A	←
TerminationCharacterEnabled	Boolean	false	**true**
SendEndEnabled	Boolean	True	←

SendEndEnabled が true の場合、WriteString メソッドは送信されるメッセージ全体が最終バイトを以って完結する事を明示的に示すため、バルクアウト・ヘッダの EOM フィールドを設定します。CR や LF などのターミネータが付加されるわけではないので注意して下さい。逆に False に設定されている場合は EOM は設定されないため、コマンド文字列中に明示的な LF コードなどが埋め込まれていない限り、計測器側ではターミネートされたとは解釈されません。

TerminationCharacterEnabled が true の場合 ReadString メソッドは、TerminationCharacter で指定されたメッセージ・バイトより後のレスポンスを送り返さないように、更に ReadString のパラメータで渡されたメッセージ・バイト数を上限としてレスポンスを返す

ように、計測器に対して指示します。USBTMC 計測器はこの条件に従ってレスポンスを返すように設計されているので、ReadString は実際には計測器から返されたメッセージを全て受信します。TerminationCharacterEnabled と TerminationCharacter プロパティは ReadString にのみ効果があります。WriteString には何ら効果がないので注意して下さい。

Notes:

TerminationCharacterEnabled=true を利用して USBTMC 計測器から受信を行なう場合、TerminationCharacter プロパティの値を任意のものに変更できるかどうかは計測器によって異なります。USBTMC 仕様では、デバイス(計測器)がレスポンスを送り返す場合の TerminationCharacter 設定機能は必須事項にはなっていません。その場合、TerminationCharacter をデフォルト値(0x0A)以外に設定しようとしたり、或いは単に(0x0A も含めて)設定しようとする行為そのものをエラー扱いする場合があります。

LAN(VXI-11)の場合

VXI-11 計測器の場合、メッセージ・ターミネータには CR, LF, CR+LF, あるいはこれらに end フラグが合わさったものが使われます。end フラグ GPIB の EOI 信号をを模倣して設計されたものです。GPIB の場合とは異なり LAN には end フラグを表現するための専用ラインは有りません。そこで VXI-11 仕様では、水面下で行われる RPC 関数コールのパラメータにそれを指定できるようになっています。意味合い的には GPIB の EOI と全く同じと考えてよいでしょう。

Notes:

全ての LAN 計測器が VXI-11 に対応しているわけではありません。使用する計測器が VXI-11 に対応しているかは、その機種の仕様書や説明書、或いは計測器自身が提供する埋め込み WEB サイトなどを参照して下さい。

KI-VISA IO Config での LAN 計測器検索機能は、VXI-11 /DNS-SN での検索ヒットの他に、LXI 標準の XML ファイルも参照しています。検索の結果 VXI-11 の VISA アドレスが表示された場合は殆どの場合その計測器は VXI-11 に対応しています。

VISA ライブラリの場合、デフォルトでは end フラグ以外は一切使われません。もちろんこれは end フラグ以外の(LF 等の特定メッセージバイト)をターミネータとして利用できないという意味ではありません。デフォルトでは end フラグのみであるという事です。

先のプログラムの例 WriteString("*IDN?\n")では改行コードを含む 6 バイトのメッセージを送信し、この最終バイト(この場合は LF 文字、ASCII コード 0x0A)を以ってメッセージが完了する(end)という意思を計測器側に伝達します。

Table 7-6　　VXI-11 のターミネータに有効なプロパティ(IMessage インターフェース)

プロパティ	タイプ	デフォルト値	IEEE488.2 計測器での推奨値
TerminationCharacter	Byte	0x 0A	←
TerminationCharacterEnabled	Boolean	false	**true**
SendEndEnabled	Boolean	True	←

LAN(HiSLIP)の場合

HiSLIP 計測器の場合、メッセージ・ターミネータには CR, LF, CR+LF, あるいはこれらに DataEND メッセージが適用されたものが使われます。HiSLIP では内部のプロトコルに Data(データ転送)と DataEND(データ転送、終端)の両方が用意されており、DataEND は GPIB の EOI 信号付送信をを模倣して設計されたものです。意味合い的には GPIB の EOI と全く同じと考えてよいでしょう。

Notes:

全ての LAN 計測器が HiSLIP に対応しているわけではありません。また仕様書自体が 2010 年公開と非常に新しい為、市場に存在する HiSLIP 計測器はごく僅かです。計測器が HiSLIP に対応しているかは、その機種の仕様書や説明書、或いは計測器自身が提供する埋め込み WEB サイトなどを参照して下さい。

KI-VISA IO Config での LAN 計測器検索機能は、VXI-11 /DNS-SN での検索ヒットの他に、LXI 標準の XML ファイルも参照しています。検索の結果 HiSLIP の VISA アドレスが表示された場合は殆どの場合その計測器は HiSLIP に対応しています。

VISA ライブラリの場合、デフォルトでは DataEND での転送によるターミネーションが行われます。もちろんこれは end フラグ以外の(LF 等の特定メッセージバイト)をターミネータとして利用できないという意味ではありません。デフォルトでは DataEND メッセージを使用しているという事です。

先のプログラムの例 `WriteString("*IDN?\n")`では改行コードを含む 6 バイトのメッセージを送信し、この最終バイト(この場合は `LF` 文字、ASCII コード 0x0A)を以ってメッセージが完了する(end)という意思を計測器側に伝達します。

Table 7-7　　HiSLIP のターミネータに有効なプロパティ(IMessage インターフェース)

プロパティ	タイプ	デフォルト値	IEEE488.2 計測器での推奨値
`TerminationCharacter`	`Byte`	0x 0A	←
`TerminationCharacterEnabled`	`Boolean`	false	**true**
`SendEndEnabled`	`Boolean`	True	←

LAN(SCPI-RAW、単純ソケット)の場合

単純ソケットの計測器の場合は、或る意味 ASRL(シリアル)全二重通信の場合と非常に似ています。メッセージ・ターミネータには CR, LF, CR+LF などはありますが、VXI-11 や HiSLIP のような end フラグ或いは DataEND メッセージのようなメカニズムは存在しません。VISA ライブラリの場合、単純ソケットの VISA アドレスに対するデフォルトでは、`WriteString` に関しては LF などの特定メッセージバイトの考慮は行われません。

例えば `WriteString("*IDN?")`と書いた場合、ソケットではデフォルトでは 5 バイトのメッセージを送信するだけです。end フラグのような物をアサートする術もありません。殆どの場合 LAN 単純ソケットを持つ計測器は、これでは応答しません。

計測器がコマンドを正しくターミネートできる様にするには、送信するコマンドに明示的に改行コード明示的に含ませる必要があります。(`"*IDN?\n"`等)

クエリに対するレスポンスを受信する場合も同様の問題がありますが、殆どの計測器はレスポンスに明示的な改行コードを付加してきます。VISA の ReadString()でそれを正しく受信してターミネートするには、`TerminationCharacterEnabled` プロパティを `True` にセットしておく必要があります(デフォルトは False)。

`TerminationCharacter`、`TerminationCharacterEnabled` の各プロパティは、受信動作(Read)に対して適用されます。送信動作(Write)には影響しません。

Table 7-8　　単純ソケットのターミネータに有効なプロパティ(IMessage インターフェース)

プロパティ	タイプ	デフォルト値	IEEE488.2 計測器での推奨値
`TerminationCharacter`	`Byte`	0x 0A	←
`TerminationCharacterEnabled`	`Boolean`	false	**true**

8- VISA アドレスの動的な設定

8-1 永続性のない VISA アドレス

これまで説明した内容を理解すれば、計測器に対して基本的な IO 動作を行う事はできるでしょう。しかし、ここで重要な問題があります。それは VISA アドレスの指定方法です。これまでの例では Open メソッドに特定の IO インターフェース・タイプに依存した VISA アドレスの文字列(例えば "GPIB0::3::INSTR"など)を直接渡していました。動作確認や実験検証のためのプログラムはともかく、最終的な計測制御アプリケーションにこのような文字列をプログラム内に直接コーディングする事は賢いやり方ではありません。このような事をすると、VISA アドレスの決定に関して実行時の柔軟性に欠けたアプリケーションになってしまいます。

通常、VISA アドレスの決定は自動システム・アプリケーションの最終ユーザ(工場でのオペレータ等)の動作環境や用意される計測器の IO インターフェースなどによって変化します。計測アプリケーションの開発時と同じ VISA アドレスが最終的なシステム稼動時に使える保証はありません。GPIB ならばデバイスアドレスを同じ値に設定してやれば、同じ VISA アドレスを別の実行環境で再現するのは比較的簡単です。しかし、シリアル・ポートでは同じ COM ポート番号が使えるとは限りません。LAN 計測器の場合はネットワーク環境が変わると(意図的にルーターや DHCP サーバーを環境設定しない限り)同じ IP アドレスが振られることはまずありません。

USB の場合は更に面倒な事が起こります。USB 計測器に対する VISA アドレスには計測器のベンダーID(VID)、プロダクト ID(PID)、そしてシリアル番号が含まれています。この場合、修理や計測器の貸し借りなどで計測器のセットを交換した場合、たとえ同じ機種であってもシリアル番号が変わってしまい、同じ VISA アドレスを再現することは不可能です。このように VISA アドレスには、計測アプリケーションの設計者が想定できるような永続性は無いのです。

このような問題を解決する有効な方法は 2 つあります。

8-2 FindRsrc を使ったリソースの検索

まずひとつの方法は、有効な VISA アドレスを実行時に検索し、候補から選択させる GUI をアプリケーションに用意し、使用者にそれを選択させる方法です。アプリケーション自体に、計測業務の本質とは直接関係のない機能(VISA アドレス選択 GUI、選択状態のレジストリへの永続保存等)を実装する面倒がありますが、それでもこの手法は最も多く使われます。

有効な VISA アドレスの検索にはリソース・マネージャで `IResourceManager3.FindRsrc` メソッドを使います。

```
string [] IResourceManager3.FindRsrc( string expr);
```

expr パラメータは正規表現と呼ばれる文字列で、検索対象の VISA アドレスを抽象表現します。戻り値は、見つかった複数の VISA アドレスを格納する文字列配列です。

正規表現と検索対象リソースの例を幾つか下の表に示します。

Table 8-1　　正規表現の例

正規表現	検索にヒットするリソース
?*	全てのリソース
?*INSTR	全ての INSTR リソース
ASRL[0-9]*::?*INSTR	全ての ASRL INSTR リソース
GPIB[0-9]*::?*INSTR	全ての GPIB INSTR リソース
GPIB?*INSTR	全ての GPIB INSTR と GPIB-VXI INSTR リソース
GPIB[^0]::?*INSTR	GPIB0 を除く全ての GPIB INSTR リソース
USB[0-9]*INSTR	全ての USB INSTR リソース
TCPIP?*INSTR	全ての LAN INSTR リソース(SCPI-RAW は INSTR でないので除外)
TCPIP?*	全ての LAN リソース(SCPI-RAW も含む)

FindRsrc で検索された VISA アドレスの文字列をアプリケーション GUI のコンボボックス等に提示すれば、アプリケーションに VISA アドレスを直接ハード・コーディングすることなく、いつでも適切な選択肢を使用者に提示することが出来ます。

次の C#プログラム例は、FindRsrc で検索された全ての VISA アドレスを Form ベースのアプリケーションのコンボボックス(変数 cboVisaAddress)に追加する方法を示したものです。

```
// 正規表現で VISA アドレスを検索する
IResourceManager3 rm = new ResourceManagerClass();
string[] adrsList = rm.FindRsrc("?*INSTR");

// 見つかった全ての VISA アドレスをコンボボックスに追加する
foreach( string s in adrsList) {
    cboVisaAddress.Items.Add( s);
}
```

Notes:

KI-VISA では、USB とシリアルに関しては FindRsrc 実行時に、有効な VISA アドレスを事前のコンフィグレーション無しに検索します。これらの検索は Plug&Play の Configuration Manager への問い合わせ(USB の場合)又はレジストリの探索(シリアルの場合)を利用して動作します。GPIB に関しては IO Config において、**Enable dynamic search for active controllers and listers** が有効な場合には GPIB のバス信号を通じてリスナー検索行い、そうでない場合には事前コンフィグレーションのテーブルを元にリストを提示します。LAN の場合は常に事前コンフィグレーションのテーブルを使用します。

GPIB での計測器検索は、実際に GPIB のバスに信号を発生します。そのため、別のプログラムが計測器と通信を行っている最中に FindRsrc が呼び出されると、場合によってはバスに妨害信号を出す危険性があります。この問題を回避したい場合は、**Enable dynamic search for active controllers and listers** を無効にして事前コンフィグレーションを行うことを推奨します。

8-3 VISA エイリアスを使う

VISA には VISA アドレスのエイリアス(別名)という考え方があります。例えばエイリアスとして "MYDMM"を作成し、これを外部のコンフィグレーション・テーブル等を通じて "GPIB0::4::INSTR"と解釈したり"ASRL2::INSTR"と解釈したりするという方法です。このような方法をとれば、アプリケーションからは VISA アドレスの代わりにエイリアスを使用してハード・コーディングしても、実行時に適切な VISA アドレス名に変換する事ができます。VISA アドレスを選択させる GUI を提供する必要はありません。エイリアスとその解決先 VISA アドレスとのマップは KI-VISA 自体によって管理され、アプリ外部に保存されます。

例えば上記の例では、次のような記述で VISA セッションをオープンできます。

```
IMessage msg = (IMessage)rm.Open( "MYDMM", AccessMode.NO_LOCK, 0, "");
```

この場合、"MYDMM"という名前の VISA エイリアスがどの VISA アドレスにマップされるかは VISA 自身によって管理されるため、マップを変更すればアプリケーションの変更無しに異なる VISA アドレスへ転換できます。

Notes:

VISA エイリアスは KI-VISA Instrument Explorer で設定できます。

VISA エイリアスは VISA をインストールした実行マシン毎に設定する必要があります。

9- リソース・ロッキング

9-1 なぜロックが必要か

VISA では、単一のリソース(計測器)に対してアプリケーションから複数の VISA セッションをオープンする事ができます。従って、アプリケーションはそのリソースを異なるセッションから同時にアクセスする事が可能になります。しかし状況によっては、リソースに対する複数セッションからのアクセスに対して制限をかけたい場合もあります。例えば、あるアプリケーションが一連の書き込み(メッセージ送信)を確実に行う必要がある場合、他のセッションがそのリソースに対して同時書き込みを行うとまずい事になります。

具体的な例を挙げてみましょう。例えば、あるプロセス(アプリケーション・プログラム)が複数のスレッドを作成し、プライマリ・スレッドが計測データを定期的に読み込み、バックグラウンドで動作するセカンダリ・スレッドが計測データ取得の合間を縫ってリアルタイム画像データを吸い上げるような場合が考えられます。このような場合、スレッド毎に VISA セッションをオープンするのが普通です。

もう一つの具体例としては、例えば、あるプログラムが"MEAS:VOLT?"クエリを発行してそのレスポンス(電圧計測データ)を受取るとします。それと同時に別のプログラムが、同じ計測器に対して "MEAS:CURR?" クエリを発行してそのレスポンス(電流計測データ)を受取るとします。これらのプログラムを同時に実行すると、期待した動作をしない場合があります。

このような動作を安全に行なう場合、計測器に対する IO 動作を同期させる必要がありますが、下記 A/B のプログラムを同時に実行した場合、期待した動作にはなりません。

例:

```
//プログラム A

int r = msg.WriteString("MEAS:VOLT?\n");
string rd = msg.ReadString(256);
```

```
//プログラム B

int r = msg.WriteString("MEAS:CURR?\n");
string rd = msg.ReadString(256);
```

実際の IO 動作を時系列順に見た場合、次のようになる可能性があります。

Table 9-1 リソース・ロッキングを使わない場合の問題例

実行順序	想定される動作
プログラム A	msg.WriteString("MEAS:VOLT?\n");
プログラム B	msg.WriteString("MEAS:CURR?\n");
プログラム A	msg.ReadString(…); 電圧計測データを期待するが、電流データを誤って取得
プログラム B	msg.ReadString(…); 電流計測データを期待するが、計測器の応答がなくタイムアウトする

このように同期を行わずに、複数スレッド又は複数プロセスで別々の VISA セッションから同じリソースに同時アクセスすると、プログラムは期待した順序で実行されない場合があります。このような状況を解決するのがリソース・ロッキングです。

Notes:

リソース・ロッキングを使わない場合の上記の問題は、IO リソース・タイプによって発生する場合としない場合があります。

GPIB、USBTMC、VXI-11 の場合は、クエリによって生成されたレスポンスはそのあとの明示的な読み込み動作(ReadString()等)が行われるまで計測器側で保持されています。ホストから読み込み動作が行われた時、その VISA セッションが該当クエリを送った時に使われた VISA セッションかどうかとは関係なく、計測器はレスポンスを送り出します。従って上記の問題を発生します。(セッション A から送られたクエリに対する応答をセッション B から受け取れます。)

SCPI-RAW(SOCKET)及び HiSLIP の場合は、クエリによって生成されたレスポンスは即座にホストへ返信されます。ホストが読み込み動作を開始する必要はありません。このような動作になるのは、いずれの IO リソースも全二重動作になっているからです。この場合、レスポンスは該当クエリを送った時の VISA セッションに対してそのセッション固有の TCP/IP ソケットで返信されるため、上記のような問題は発生しません。(セッション A から送られたクエリに対する応答はセッション A からのみ受け取れます。)

ASRL(シリアル)の場合は、基本的には全二重シリアルなので SCPI-RAW(SOCKET)や HiSLIP のようにレスポンスは即座にホストへ返信されますが、ホスト側のシリアル・ポート受信バッファが COM ポート毎に単一であり複数セッション間で共有されています。従って GPIB と同様に上記の問題を発生します。

9-2 ロックの仕組み

VISA のロッキング・メカニズムは、リソースへのアクセスを VISA セッション単位で調停します。ある VISA セッションがリソースをロックしている場合、他の VISA セッションから同じリソースに対する操作を行おうとすると、その動作の種類やロック・タイプによって動作が受け入れられる場合とエラーになる場合とがあります。

リソースがどのセッションからもロックされていない場合、IO 動作及びグローバル属性値を変更する事に関して何ら制限を受ける事は有りません。この場合、VISA セッションは特にロックを獲得する必要も無く、リソースに対するすべての操作を行う事が出来ます。しかし、他の VISA セッションがすでにリソースをロックしている場合、グローバル属性値を変更しようとしたり特定の動作(特に IO 動作)を行おうとすると失敗します。リソースをロックする事で他の VISA セッションからのアクセスが制限されているのです。特に排他的ロックを獲得している場合は、他の VISA セッションからのアクセスが原因で動作が失敗する事がないように、他の VISA セッションからのアクセスを完全に排除します。

Notes:

属性値とは、セッションを通じて設定又は取得可能な「状態変数」のようなものと考えて下さい。VISA COM では通常、属性をプロパティとしてアクセスするか、又は SetAttribute/GetAttribute メソッドを通じてアクセスすることができます。属性には、VISA セッションごとに管理されるもの(ローカル属性)と、リソースごとに管理されるもの(グローバル属性)があります。ローカル属性は、同じリソースに関連付けられた複数の VISA セッションで別々に管理されるので、たとえリソースがロックされていても自由に読み書きができます。一方グローバル属性は、ある VISA セッションがその値を書き換えると同じリソースに関連付けられた他のセッションにも影響が及ぶため、リソースがロックされているときは書き込み動作においてアクセス制限がかかります。

9-3 ロック・タイプ

VISA では 2 種類のロック・タイプ、「排他的ロック(Exclusive Lock)」と「共有ロック(Shared Lock)」が定義され、LockRsrc メソッドのパラメータにはそれぞれ、EXCLUSIVE_LOCK 及び SHARED_LOCK が使われます。LockRsrc メソッドはロックを獲得し、UnlockRsrc メソッドはロックを開放します。現在のロック状態は LockState プロパティを参照する事で行えます。LockState プロパティはロック状態に応じて EXCLUSIVE_LOCK 又は SHARED_LOCK を返すか、或いはロックを獲得していない場合には NO_LOCK を返します。

排他的ロック

アプリケーションは VISA セッションを通じて EXCLUSIVE_LOCK 指定で LockRsrc メソッドを呼び出す事により、リソースに対する排他的アクセス権限を獲得します。この場合、同じリソースに関連付けられている他の VISA セッションは無条件にアクセスを制限されます。

VISA セッションが排他的ロックを保持している場合、他の VISA セッションはグローバル属性値(プロパティ)を変更したり IO 動作を行う事はできません。しかし属性値(プロパティ)の読み出しは制限されません。

共有ロック

アプリケーションは VISA セッションを通じて SHARED_LOCK 指定で LockRsrc メソッドを呼び出す事により、他のセッションとロックを共有する事ができます。共有ロックにおけるアクセス権限は排他的ロックと同じですが、複数の VISA セッションでロックを共有できるところが違います。

VISA セッションが共有ロックを保持している場合、ロックを共有する他の VISA セッションからはグローバル属性(プロパティ)を変更したり IO 動作を行う事ができます。しかしロックを共有していない VISA セッションからのアクセスは制限されます。

VISA セッションは、複数回に渡ってロックを獲得する事ができます。その場合、排他的ロックと共有ロックを混在させて獲得する事ができます。下記の表は、ロックを獲得できるための条件を示したものです。

Table 9-2　ロックを獲得したいセッションが、現在何もロックを保持していない場合

獲得したいロック	他のセッションが保持しているロック			
	なし	Exclusive Lock	Shared Lock	Shared and Exclusive Locks
Exclusive	Yes	No	No	No
Shared	Yes	No	Yes*	Yes*

Table 9-3　ロックを獲得したいセッションが、Exclusive Lock だけを保持している場合(ネスティング)

獲得したいロック	他のセッションが保持しているロック			
	なし	Exclusive Lock	Shared Lock	Shared and Exclusive Locks
Exclusive	Yes	**	**	**
Shared	No	**	**	**

Table 9-4　ロックを獲得したいセッションが、Shared Lock だけを保持している場合(ネスティング)

獲得したいロック	他のセッションが保持しているロック			
	なし	Exclusive Lock	Shared Lock	Shared and Exclusive Locks
Exclusive	Yes	**	Yes	No
Shared	Yes	**	Yes	Yes

Table 9-5　ロックを獲得したいセッションが、Shared Lock と Exclusive Lock の両方を保持している場合(ネスティング)

獲得したいロック	他のセッションが保持しているロック			
	なし	Exclusive Lock	Shared Lock	Shared and Exclusive Locks
Exclusive	Yes	**	Yes	**
Shared	Yes	**	No	**

Notes:

Yes* - ロックを獲得したいセッションが LockRsrc 呼び出しで正しいアクセス・キーを提示した場合に限る

** - ロッキング・メカニズム上、このような場面が発生する事はない

共有ロックを保持した状態で排他的ロックを獲得する

複数の VISA セッションが共有ロックを獲得している場合、そのうちの一つのセッションだけは共有ロックを保持したまま排他的ロックを獲得する事が出来ます。つまり、共有ロックを獲得しているセッションのうち一つは、更に LockRsrc メソッドを呼び出して排他的ロックを獲得できるという事です。排他的ロックと共有ロックの両方を獲得した VISA セッションは、共有ロックだけを獲得していた時と同じアクセス権限を持ちます。しかしこの場合、共有ロックを獲得していた他のセッションはリソースに対するアクセスを制限されるようになります。排他的ロックを獲得した VISA セッションが UnlockRsrc によってロックを開放した時、共有ロックを獲得していた他のセッションは再びリソースに対する全てのアクセス権限を与えられます。同じ共有ロックを獲得している複数の VISA セッション同士でリソース・アクセスに対する同期を取りたい場合、この仕組みは便利です。この逆の場合、つまり排他的ロックだけを(共有ロック無しで)保持している VISA セッションが共有ロックを更に獲得する事は、VISA では許されていません。

ロックのネスティング

VISA ではロックのネスティングがサポートされています。つまり、VISA セッションは複数回に渡って同じタイプのロックを行う事ができるという事です。リソースをアンロックするには、LockRsrc を呼び出した回数と同じ回数の UnlockRsrc 呼び出しが必要になります。それぞれの VISA セッションは、それぞれのロック・タイプに対して別々のロック・カウントを管理しています。同じ VISA セッションで LockRsrc を繰り返し呼び出した場合は、指定されたロック・タイプのロック・カウントが増えていきます。共有ロックを要求する LockRsrc の多重呼び出しでは、毎回同じアクセス・キーが返されます。排他的ロックを要求する LockRsrc の呼び出しでは、呼び出しがネストされているかどうかに関係なく、アクセス・キーが返されることはありません。VISA セッションが複数回に渡ってリソースをロックした場合、リソースを実際に開放するには同じ回数分の UnlockRsrc 呼び出しが必要になります。つまり、LockRsrc はロック・カウントを増加させ、UnlockRsrc はロック・カウントを減少させます。ロック・カウントがゼロになった時、リソース・ロッキングが実際に開放されます。

VISA セッションが共有ロックをネスティング状態で要求する場合、LockRsrc 呼び出しに対してアクセス・キーを提示する必要はありません。つまり VISA セッションは、以前に獲得した共有ロックのアクセス・キーを 2 回目以降の呼び出しでは提示する必要がないということです。しかし、ネスティングで共有ロックを要求する場合に、もしアクセス・キーを明示的に提示するのであれば、そのアクセス・キーは正しいものでなければなりません。アクセス・キーに関する説明は、LockRsrc の説明を参照して下さい。

9-4 リソースのロックとアンロック

ロックを獲得するには、LockRsrc メソッドを、ロックを開放するには UnlockRsrc メソッドを使用します。

```
string IVisaSession.LockRsrc(
   AccessMode type,
   int lockTimeout,
   string requestedKey
);


void IVisaSession.UnlockRsrc();
```

LockRsrc メソッドの *type* パラメータはロックの種類を指定します。指定できるのは、EXCLUSIVE_LOCK 又は SHARED_LOCK のどちらか一方です。両方を論理 OR で合成して同時に指定する事は出来ません。EXCLUSIVE_LOCK を指定した場合、そのリソースへの排他的アクセス権を獲得します。ただし、他のセッションが同一 IO リソースに対して既にロックを獲得している場合は、*lockTimeout* パラメータで指定された時間(ミリ秒)のあいだロックが開放されるのを待ちます。その時間内に、ロックを獲得中のセッションが UnlockRsrc メソッドでロックを開放すれば、入れ替わりに LockRsrc を呼び出したセッションがロックを獲得します。ロック・タイムアウト指定時間を過ぎてもロックを獲得できない場合、LockRsrc メソッドは失敗します。

requestedKey パラメータは *type* パラメータに SHARED_LOCK を指定した場合のパラメータです。共有ロックは、複数の VISA セッションで「アクセス・キー」を設定し、そのキーを提示する VISA セッション同士でロックを共有できる仕組みです。アクセス・キーには文字列を使用します。また戻り値は、SHARED_LOCK を指定した場合は、そのアクセス・キー文字列を返します。リソースがどのセッションからもロックされていない状態で共有ロックを獲得する場合、LockRsrc を呼び出す時点ではまだアクセス・キーは存在しません。この時 LockRsrc に渡す要求アクセス・キーを省略(ブランク文字列)する事ができます。この場合、共有ロックの文字列は VISA によって作成され、それが LockRsrc メソッドの戻り値となります。アプリケーションから明示的な要求アクセス・キーが提示された場合はそれが使われます。

EXCLUSIVE_LOCK を指定した場合は鍵を全く使用しないので、*requestedKey* パラメータは無視され、戻り値はブランク文字列になります。

UnlockRsrc にはパラメータがありません。このメソッドは既に獲得済みのロックカウントを 1 つ下げる効果があります。ロックを全く獲得していない VISA セッションがこれを呼び出すとエラーになります。また、VISA セッションが排他的ロックと共有ロックの両方を既に獲得している場合は、排他的ロックが優先的に開放されます。

先ほどの 2 つのプログラムにリソース・ロッキングを追加してみましょう。今度は、WriteString と ReadString の呼び出しを LockRsrc と UnlockRsrc の呼び出しで囲むことになります。ロックが獲得されてから開放されるまでの間、同じ VISA アドレスを操作する他の VISA セッションは IO アクセスを一切行なうことはできず、またロックの獲得も待たされる事になります。

```
//プログラム A

msg.LockRsrc( AccessMode.EXCLUSIVE_LOCK, 2000, "");
r = msg.WriteString("MEAS:VOLT?\n");
rd = msg.ReadString(256);
msg.UnlockRsrc();
```

```
//プログラム B

msg.LockRsrc( AccessMode.EXCLUSIVE_LOCK, 2000, "");
r = msg.WriteString("MEAS:CURR?\n");
rd = msg.ReadString(256);
msg.UnlockRsrc();
```

この例では、どちらのプログラムも、排他的ロックの獲得要求に関して最大 2000ms まで待つと指定しています。ここでは計測コマンドの応答が 2000ms 以内で得られるという前提なので、これらのプログラムはタイムアウトする事なく交互にロックを獲得し、しかも WriteString/ReadString のペアが互いに干渉することなく正常に目的のデータを取得できます。

ロック獲得から開放までの処理がもっと長い場合は、このロックが開放されるのを待つ他の VISA セッションでは、ロックタイムアウトをもっと長い時間に調整したほうが良いでしょう。そうでないとロックの獲得がタイムアウトして失敗し、VISA はエラーを発生します。

Notes:

LockRsrc でロックを獲得したら、忘れずに同じ回数分の UnlockRsrc を呼び出して下さい。UnlockRsrc を忘れると、同じ IO リソースを操作する他の VISA セッションは IO アクセスもロックの獲得も出来なくなります。

VISA セッションがロックを獲得したまま必要な回数分の UnlockRsrc を呼び出さずに Close メソッドを呼び出した場合、そのセッションが保持する全てのロック・カウントはゼロに戻され、同じ IO リソースをアクセスする他のセッションがロックを獲得できるようになります。

VISA セッションが Close メソッドを呼び出さずに廃棄された場合、セッション・オブジェクトのデストラクタによって自動的に Close 処理が行なわれます。すなわちロック・カウントもゼロに戻ります。

HiSLIP、SCPI-RAW の場合は、IO アーキテクチャの都合上リソースロッキングを行わないプログラム同士でも正常に同時実行できる場合があります。例えばクエリとレスポンス回収を行う場面であれば、複数プログラムの

処理タイミングが入れ子になっても問題は起きませんが、トリガ設定予約とトリガ遂行のように実行順序が重要なコマンドを分散処理させている場合は、リソースロッキングを使う必要があります。

9-5 ネットワーク越しのロック

ローカルバス(PCI, PCIe, USB、シリアルなど)に接続された計測器はでは、通常は複数の PC から同時にアクセスする事はできませんが、ネットワーク上に接続された計測器ではそれが可能です。ここでいうネットワークに接続された計測器とは、下記のような例を指します。

- LAN ベースの計測器(VXI-11、HiSLIP、SCPI-RAW ソケット等)
- LAN ベースの GPIB アダプタ(NI GPIB-ENET/100 等)に接続された計測器
- VISA が提供する Gateway 機能でブリッジ接続された計測器

このような接続形態を持つ計測器は、多くの場合ロックメカニズムは複数マシン間でも動作します。順番に例を見ていきましょう。

LAN ベースの計測器

例えば、同じサブネット内の複数の PC から、IP アドレス 192.168.1.50 の LAN ベースの計測に同時に VISA セッションをオープンするような場合です。この時マシン A から LockRsrc()で排他的ロックをを呼び出すと、マシン B からのアクセスはどうなるでしょうか。

この場合、VISA セッションが VXI-11 又は HiSLIP の場合にはロックの状態管理が計測器自身によって行われます。従って、マシン B からのアクセスはブロックアウトされます。(VXI-11 及び HiSLIP プロトコルは計測器ファームウェア側でのロック管理メカニズムの実装が必須になっています。)

VISA セッションが無手順ソケット(SCPI-RAW)の場合には、マシン間でロック状態を共通管理するメカニズムが根本的に存在しないので、アクセスはブロックされません。マシン A と B の VISA ライブラリがネットワーク通信をしてロック管理を同期させていれば可能かもしれませんが、一般に VISA ライブラリはそれをやっていないようです。KI-VISA でもそのような機能は提供していません。

Notes:

VXI-11 で提供されるロック管理メカニズムは、ネスティングをサポートしていません。そのため VISA がネスティング機構を提供します。HiSLIP で提供されるロック管理メカニズムは VISA 仕様と同じになっているため、計測器自身がネスティングを管理します。

LAN ベースの GPIB アダプタ

例えば、同じサブネット内の複数の PC から、IP アドレス 192.168.1.33 の LAN-GPIB アダプタに接続された GPIB 計測器に対して、セッションをオープンするような場合です。この時マシン A から LockRsrc()で排他的ロックをを呼び出すと、マシン B からのアクセスはどうなるでしょうか。

これは LAN-GPIB アダプタの仕様により異なる可能性があります。しかし National Instruments 製の GPIB-ENET/100 等では、GPIB アダプタ自身にロック管理のメカニズムが備わっています。

KI-VISA の NI-488.2M サポートでは、**Enable board-level locking** のオプションを有効にした場合は、LockRsrc()/UnlockRsrc()が VISA 内部で NI-488.2M API の iblck()関数を呼び出しています。従って、それが有効に働けばマシン B からのアクセスはブロックアウトされます。

VISA が提供する Gateway 機能

例えば、VISA の Gateway 機能を利用し、マシン C のローカルバスに接続された計測器を外部マシンから仮想的な VXI-11 計測器に見せてしまう方法です。この状態で、A から LockRsrc()で排他的ロックをを呼び出すと、マシン B からのアクセスはどうなるでしょうか。

この場合、マシン C で実行される VISA の Gateway 機能(仮想 VXI-11 デーモンのような物)がロックを管理する事になります。(VXI-11 計測器の場合と同じ。) 従って、マシン B からのアクセスはブロックアウトされます。

Notes:

現在の KI-VISA バージョンでは、Gateway 機能(仮想 VXI-11 デーモン)は用意されていません。

10- イベント・キュー

VISA ではイベントを扱う事が出来ます。イベントとは、アプリケーション・プログラムなどで注視しなければならないような事象発生のことを指します。イベントには、サービス・リクエストの発生(SRQ)、割り込み、ハードウェア・トリガ、IO 動作の完了、などがあります。

イベントの受け取り方には 2 種類あります。

イベント・キューを使う

EnableEvent で 1 つ又は複数のイベント発生を有効にし、WaitOnEvent でイベントの発生を待ちます。WaitOnEvent を呼び出したスレッドは実行を停止し、期待しているイベントが発生するか又は指定されたタイムアウト時間が経過すると呼び出しがリターンします。この方法はイベント・キューと呼ばれます。

イベント・コールバックを使う

InstallHandler でコールバック・ハンドラーを登録し、EnableEvent で 1 つ又は複数のイベント発生を有効にします。イベントが発生すると、登録済みのコールバック・ハンドラーが呼び出されます。この方法はイベント・コールバックと呼ばれます。コールバック・ハンドラーはアプリケーション・プログラム側で用意しなければなりません。

この章ではイベント・キューの使い方を、「11- イベント・コールバック」ではイベント・コールバックの使い方をそれぞれ説明します。

10-1 非同期 IO

VISA では非同期 IO 動作をサポートしています。非同期 IO とは、長くかかるかもしれない IO 動作の完了を待たずに、直ぐにメソッドの呼び出しがリターンするような動作です。例えば IMessage インターフェースの Write/WriteString/Read/ReadString メソッドは、IO 動作が完了するまでメソッドの呼び出しはリターンしません。しかし IAsyncMessage インターフェースが提供する Write/WriteString/Read メソッドは呼び出しが直ぐにリターンします。

非同期 IO を行うには IAsyncMessage インターフェースを使用します。また、非同期 IO 機能の管理や IO 結果の取得などで、IEventManager インターフェースと IEventIOCompletion インターフェースを使用します。

```
IAsyncMessage amsg =
   (IAsyncMessage)rm.Open("GPIB0::3::INSTR", AccessMode.NO_LOCK, 0, "");

IEventManager evm = (IEventManager)msg;
```

IAsyncMessage インターフェースは IMessage インターフェース同様に IO メッセージを処理するためのインターフェースですが、IAsyncMessage インターフェースは非同期 IO を行います。IEventManager インターフェースは、イベント・キューやイベント・コールバックなど、イベントが関与する機能全体の動作を管理するためのインターフェースです。IEventIOCompletion インターフェースは、IO 完了イベントにアクセスするためのインターフェースです。

まずはサンプルコードを見てみましょう。これは、2 行の SCPI コマンド送信とレスポンス回収を非同期動作で行い、最後に完了イベントを待つプログラムです。

```
IResourceManager3 rm = new ResourceManagerClass();

IAsyncMessage amsg =
   (IAsyncMessage)rm.Open(
   "TCPIP0::192.168.1.22::inst0::INSTR",
   AccessMode.NO_LOCK, 0, ""
   );
```

```
IEventManager evm = (IEventManager)amsg;

evm.EnableEvent(
   EventType.EVENT_IO_COMPLETION, EventMechanism.EVENT_QUEUE, 0);

int job1;
int job2;
int job3;

job1 = amsg.WriteString(":SOUR:CURR 10;VOLT 1.5;:OUTP ON\n");
job2 = amsg.WriteString(":MEAS:CURR?");
job3 = amsg.Read(256);

IEventIOCompletion ev;
ev =
(IEventIOCompletion)evm.WaitOnEvent(1000,EventType.EVENT_IO_COMPLETION, 0);
ev =
(IEventIOCompletion)evm.WaitOnEvent(1000,EventType.EVENT_IO_COMPLETION, 0);
ev =
(IEventIOCompletion)evm.WaitOnEvent(1000,EventType.EVENT_IO_COMPLETION, 0);

evm.DisableEvent(
   EventType.ALL_ENABLED_EVENTS,EventMechanism.EVENT_ALL_MECH, 0);

amsg.Close();
```

10-2 IO 動作の開始

まず、IEventManager インターフェースの EnableEvent メソッドを呼び出しています。

```
void IAsyncMessage.EnableEvent(
   EventType type,
   EventMechanism mech,
   int customEventType
);
```

EnableEvent メソッドは、*type* パラメータで指定されたイベントタイプを *mech* パラメータで指定されたメカニズムでイベントを発生させるように VISA に指示します。現在 KI-VISA の実装では、*type* に EVENT_IO_COMPLETION を指定した場合、*mech* は EVENT_QUEUE で無ければなりません。これは VISA の制限ではなく、KI-VISA ライブラリの実装による機能制限です。*CustomEventType* パラメータは使用しないので指定しても無視されます。

次に呼び出しているのは WriteString メソッドです。

```
int IAsyncMessage.WriteString( string buffer);
```

IMessage インターフェースの同名メソッドと全く同じプロトタイプ宣言がされていますが、戻り値の意味が違います。IMessage インターフェース版では実際に送信されたメッセージ・バイト数でしたが、IAsyncMessage インターフェースの場合はジョブ ID が返されます。ジョブ ID とは、あとで IO 結果を取得する際の識別に使われるもので、数値自体に特別な意味は有りません。非同期 IO なので、実際に送信されたメッセージ・バイト数はまだ判りません。

レスポンスを取得するメソッドは Read であって、ReadString ではありません。

```
int IAsyncMessage.Read( int count);
```

ここでの戻り値は、やはりジョブ ID です。

10-3 IO の完了待ちと結果の取得

IAsyncMessage インターフェースのメソッドからは IO 結果を直接知る事はできません。その代わりに、IEventManager インターフェースの WaitOnEvent メソッドを使います。

```
IEvent IEventManager.WaitOnEvent(
   int waitTimeout,
   EventType type,
```

```
    int customEventType
);
```

WaitOnEvent メソッドは、既に動作を開始した IO アクションが発生する完了イベントを待ちます。つまりこのメソッドは、イベントの発生待ち(この場合は IO 完了待ち)を行い、その結果を保持するイベントを回収します。*waitTimeout* パラメータはイベント待ちのタイムアウトです。*type* パラメータは待とうとするイベントのタイプで、非同期 IO の完了を待つ場合は、EVENT_IO_COMPLETION か又は ALL_ENABLED_EVENTS でなければなりません。

WaitOnEvent メソッドは、*waitTimeout* パラメータで指定された時間を上限としてイベントを待ちます。イベント待ちが完了すると、そのイベントオブジェクトを返します。メソッドの構文上は IEvent インターフェースを返す事になっていますが、IO 完了イベントの場合、そのイベントオブジェクトは IEventIOCompletion インターフェースをサポートしているのは間違い有りません。そこでイベントを受取る変数は IEventIOCompletion 型で宣言されたものを使います。

IO 結果の取得を行うには、WaitOnEvent メソッドから取得されたイベントを IEventIOCompletion インターフェースを通じてアクセスします。IEventIOCompletion インターフェースには幾つか有用なプロパティがあります。

Table 10-1　　IEventIOCompletion のプロパティ

プロパティ	説明
int jobId	ジョブ ID
int IOStatus	IO 結果を示す
int ReturnCount	IO のバイト数
byte[] ReadBuffer	バイト配列版 IO バッファ
string ReadBufferAsString	文字列版 IO バッファ

JobID は非同期版の WriteString や Read メソッドが返した値と同じです。これによってイベントと IO 動作の関係を見失わないようにする事ができます。IOStatus は VISA COM API の発生するエラーコードです。正常終了の場合はゼロ、警告であればプラスの値、エラーであればマイナスの値になります。これは VISA ステータス・コードと呼ばれるものです。このコードの詳細については、オンライン・ヘルプ又は本書の付録に掲載されています。

Notes:

ReadBuffer 及び ReadBufferAsString プロパティのどちらを使っても、送信又は受信されたバッファを確認する事ができます。

ReadBuffer 及び ReadBufferAsString プロパティは、非同期版メソッドの Write、WriteString、Read の全ての IO 完了イベントに対して内容を確認できますが、VISA の仕様では書き込み系動作では保持する必要はありません。KI-VISA では書き込み系アクションのバッファのコピーを保持しますが、他のベンダーの VISA では必ずしもそうなっていないので、注意して下さい。

現在の KI-VISA の実装では、GPIB に関しては非同期 IO 動作をサポートしていません。GPIB では IAsyncMessage を使用しても同期動作となります。ただし、アプリケーションのプログラミング上の扱いが同じになるように、イベントキューはサポートされています。KI-VISA の将来のバージョンでは非同期 IO が動作するように変更される場合があります。

10-4 IO 動作の終了

非同期 IO 動作を終了する場合は、IEventManager インターフェースの DisableEvent メソッドを呼び出します。

```
void IAsyncMessage.DisableEvent(
    EventType type,
    EventMechanism mech,
    int customEventType
)
```

DisableEvent メソッドは、*type* パラメータで指定されたイベントタイプに関して *mech* パラメータで指定されたメカニズムでのイベント発生を停止するように VISA に指示します。*type* に EVENT_IO_COMPLETION のような特定イベントタイプを指定しても良いし、全てのイベントタイプを停止させるために ALL_ENABLED_EVENTS を指定しても良いでしょう。*mech* に関しても EVENT_QUEUE のように特定メカニズムを指定しても良いし、全てのメカニズムを停止させるために EVENT_ALL_MECH を指定しても良いでしょう。

Notes:

EnableEvent でイベント発生を許可しないまま非同期 IO を行った場合、計測器との IO 自体は行われますが結果を取得するためのイベントは発生しません。またこの場合は IO は同期動作になります。この場合 WaitOnEvent を呼び出しても NULL 参照が返ってきます。

DisableEvent を呼び出した後は、既に発生したイベントはイベント・キューに保持されたままになります。また、新規のイベントはキューに積み込まれません。イベント・キューからイベントを削除するには DiscardEvents メソッドを使います。

WaitOnEvent で IO 完了を待たずに次の非同期 IO 動作(WriteString など)を連続動作させた場合、IO メソッド内部で WaitOnEvent 相当の処理が行われます。またこの時のイベントはイベントキューに積み込まれます。

イベント・キューのサイズを超えてイベントが発生した場合、新しいイベントは記録されず、既存のイベントがキュー内に維持されます。

11- イベント・コールバック

「10- イベント・キュー」では、キューを使ったイベントの扱い方を説明しました。この章ではもう一つのイベント・メカニズムである、イベント・コールバックについて説明します。

11-1 サービス・リクエスト通知

イベント・コールバックはサービス・リクエスト通知で最も頻繁に利用されます。サービス・リクエスト発生の有無を調べる単純な方法は、ReadSTB メソッドを使ってシリアル・ポーリングを行う事です。しかしその方法では、サービス・リクエストを検出したいアプリケーションは While ループや Do ループ、あるいはタイマ・イベントなどを駆使して、シリアル・ポーリングを定期的に行なわなければなりません。ここで説明する方法は、サービス・リクエストが発生したら予め指定されているイベント・ハンドラをコールバックしてもらうという方法です。少し高度な方法なので、実装は前述の方法よりも多少難しくなります。

Notes:

シリアル・ポーリングについては、IO インターフェース依存の機能として、「13-1 シリアル・ポーリング(GPIB、USB、VXI-11、HiSLIP)」で解説します。シリアル・ポーリング及びサービス・リクエスト・イベントは GPIB、USBTMC、LAN(VXI-11/HiSLIP)でのみ利用する事が出来ます。

VISA COM でのイベント・コールバック・メカニズム

一般にコールバック・ハンドラを扱う DLL は、アプリケーション側で用意したコールバック関数のアドレスをパラメータにしてハンドラ登録関数を呼び出し、必要に応じてライブラリからそれを呼び出してもらうというものです。VISA C API も含め、コールバックをサポートする多くの DLL がこの手法を利用しています。

しかし、VISA COM API でのイベント・コールバックの扱い方はこれとは全く異なります。予めアプリケーション側で用意したハンドラを DLL からコールバックする点は同じなのですが、そこには.NET のデリゲートの手法が関与します。COM インターフェースを通じたコールバックの手法は一般に、アプリケーション側でイベント・シンク・インターフェースを提示する事になります。イベント・シンク(イベントの流し口)は単純な関数として実装する事はできず、「IEventHandler インターフェースを実装した COM コンポーネント・クラス」を作成する必要があります。幸い Visual Basic や C#にはこれを実装する機能が備わっています。

イベント・シンクの作成

イベント・シンクを作成するための手順は、「特定の COM インターフェースを実装するコンポーネント・クラスを作成する」手順に他なりません。この手順はプログラミング言語や開発ツールによって大きく異なります。ここでは説明の都合上 C#を例にしていますが、コンポーネント・クラスの作成手順は言語別パートで更に詳細説明を行います。

Visual C#統合環境のメニューバーから **Project ➔Add Class** を選択して下さい。**Add New Item** ダイアログが表示されるので、**Templates** から **Clsss** を選択します。また生成されるクラスの名前を MyEventSink.cs としておきましょう。そして **OK** ボタンをクリックすると MyEventSink クラスが自動生成されプロジェクトに追加されます。

追加されたクラス・モジュールの Name プロパティを、MyEventSink としましょう。つまり今から作成するのは MyEventSink というコンポーネント・クラスです。

そしてクラス・モジュール MyEventSink.cs には以下のような内容を記述します。

```
using System;
using Ivi.Visa.Interop;

namespace VisaCom_cs2  ←この部分は実際にはプロジェクト名になる
{
    /// <summary>
    /// Summary description for MyEventSink.
    /// </summary>
```

```
    public class MyEventSink : IEventHandler
    {
        public MyEventSink()
        {
        }

        public  void HandleEvent(
            IEventManager vi,
            IEvent ev,
            int userHandle)
        {
            IMessage msg = (IMessage)vi;

            short stb;
            stb = msg.ReadSTB();

            // TODO: Add your own job depending on the stb value
        }
    }
}
```

MyEventSink クラスは IEventHandler インターフェースから派生したクラスとして作成します。IEventHandler には唯一のメソッドとして HandleEvent があります。このメソッドは VISA が用意するものではなく、作成するアプリケーション側で用意する責任があります。それはアプリケーションが呼び出すメソッドではなく、VISA がアプリケーションに向かって(逆方向に)呼び出すメソッドだからです。

HandleEvent()のパラメータは、IEventManager インターフェース、IEvent インターフェース、そして int 型の *userhandle* の 3 つです。IEventManager インターフェースは、実際には VISA セッション・オブジェクト内に生息するインターフェースなので、既にリソース・マネージャから作成された VISA セッションオブジェクトの IMessage インターフェースからを簡単に参照することができます。IEvent はサービス・リクエストによって生成されたイベント・オブジェクトを参照しています。最後の *userhandle* は後で説明します。

ハンドラ・イベントを有効にする

サービス・リクエスト・ハンドラのコールバックを有効にするには、InstallHandler メソッドと EnableEvent メソッドを使います。これらはどちらも IEventManager インターフェースで提供されるメソッドです。

```
void IEventManager.InstallHandler(
    EventType type,
    IEventHandler handler,
    int userHandle,
    int customEventType
);
void IEventManager.EnableEvent(
    EventType type,
    EventMechanism mech,
    int customEventType
);
```

InstallHandler は、イベント・ハンドラーの登録をします。イベント・ハンドラーというよりは、正確にはイベント・シンクを登録すると言ったほうが良いでしょう。ここではサービス・リクエストのイベント通知が目的なので、*type* パラメータには EVENT_SERVICE_REQ を渡します。*handler* にはアプリケーション側で用意されたイベント・シンクに生息する IEventHandler インターフェースへの参照を渡します。ここで注意しなければならないのは、前項で作成したクラス MyEventSink は、それだけではオブジェクトとして生命は吹き込まれていないので、インスタンスを作成しなければならないということです。そのためには下のようなコードでインスタンスを作成しておく必要があります。

```
//
//このコードはクラス・モジュールではなく、通常の IO を行う部分に書くこと
//
IResourceManager3 rm;
IMessage msg;
IEventManager evm;

...

rm = new ResourceManagerClass();
msg = (IMessage)rm.Open(
   "USB0::0x0B3E::0x1006::BBB77700::0::INSTR", AccessMode.NO_LOCK, 0, "");

evm = (IEventManager)msg;
IEventHandler eh = new MyEventSink();

evm.InstallHandler(EventType.EVENT_SERVICE_REQ, (IEventHandler)eh, 1, 0);
evm.EnableEvent(
   EventType.EVENT_SERVICE_REQ, EventMechanism.EVENT_HNDLR, 0);

int r;
r = msg.WriteString("*CLS\n");
r = msg.WriteString("*ESE 32\n");
r = msg.WriteString("*SRE 32\n");
r = msg.WriteString("AAAAA\n");      // ←このジャンク･コマンドが SRQ を発生する！
..
.
evm.DisableEvent(
   EventType.EVENT_SERVICE_REQ, EventMechanism.EVENT_HNDLR, 0);
```

このサンプルで宣言されている変数 *eh* は IEventHandler インターフェース型で宣言されていますが、このようなインターフェース型でインスタンスを直接作ることはできないと前に説明しました。ここでは MyEventSink クラスのオブジェクト・インスタンスを作成し、その中に生息する IEventHandler インターフェースを(暗黙のタイプキャスト)によって問合せ、その結果を変数 eh に保持する事になります。*userhandle* には任意の 32 ビット整数を設定する事ができます。これは、サービス・リクエストのイベント・ハンドラ・メソッド HandleEvent が呼び出された時にパラメータ *userhandle* にそのまま現れます。それ以上の深い意味は有りませんが、同じイベント・シンクを異なる複数の VISA セッションでサービス・リクエスト通知に多重利用するような場合、イベント・ハンドラで文脈を識別する為に利用することが出来ます。

EnableEvent メソッドは、*type* パラメータで指定されたイベントタイプを *mech* パラメータで指定されたメカニズムでイベントを発生させるように VISA に指示します。サービス･リクエスト通知が目的であれば、通常は *type* に EVENT_SERVICE_REQ を指定し *mech* は EVENT_HNDLR を渡すのが妥当でしょう。*CustomEventType* パラメータは使用しないので指定しても無視されます。

EnableEvent の *mech* パラメータには EVENT_HNDLR 以外にも EVENT_SUSPEND_HNDLR を指定する事もできます。EVENT_HNDLR と EVENT_SUSPEND_HNDLR、更に DisableEvent を呼び出した場合のイベント発生条件の違いを下の表に示します。

Table 11-1　イベント発生条件の比較

状態	その状態に遷移するための条件	動作
Disabled	`DisableEvent`	新規のイベントは監視されない。これが呼び出された時にイベント・キューに未処理のイベントがあっても、それを削除する事はしない。
Suspended	`EnableEvent(` `  mech = EVENT_SUSPEND_HNDLR)`	新規のイベントは監視されキューに積み込まれるが、コールバックは行わない。
Enabled	`EnableEvent(` `  mech = EVENT_HNDLR)`	EnableEvent が呼び出されたとき、イベント・キューに未処理のイベントがあればそのぶんのコールバックを行う。 以降、新規のイベントは監視されキューに積み込まれ、即時コールバックが行われる。
---	`DiscardEvents`	イベント・キュー内の未処理イベントを全て削除する。Enable/Disable の設定には何ら影響しない。

この状態で計測器からサービス・リクエストが発生すると、イベント・シンク・オブジェクトの `HandleEvent` メソッドが VISA からアプリケーションに向かって呼び出されます。

サービス・リクエストの発生

計測器からサービス・リクエストが発生するのを待っていても良いのですが、それなりの理由が無ければ計測器はサービス・リクエストを発生しません。また、IEEE488.2 仕様の計測器の場合では Service Request Enable Register や Standard Event Enable Register などを適切に設定しておかなければサービス・リクエストを観測する事はできません。そこで、ここでは強制的にサービス・リクエストを発生させてみることにします。

上記サンプル中に書かれている `WriteString` メソッドの呼び出しで"*CLS"、"*ESE 32"、"*SRE 32"を順番に設定している部分に注目して下さい。これは SPCI 仕様の計測器に対して、コマンド・エラー発生時にサービス・リクエストを発生させるように指示するためのものです。従ってその直後に送信されるジャンク・メッセージ"AAAAA"がコマンドエラーとして処理され、計測器はサービス・リクエスト・イベントを発生します。

ステータス・バイトの確認

サービス・リクエストの原因を知るには、ステータス・バイトの値を知る必要があります。しかし、残念な事に、`HandleEvent` に渡されるイベント・オブジェクトからは `IEvent` インターフェースを駆使しても、ステータス・バイトを知ることはできません。ステータス・バイトを知るには、`HandleEvent` 内からステータス・バイト取得のメソッド `ReadSTB` を呼び出せばよいでしょう。

```
IMessage msg = (IMessage)vi; //vi は HandleEvent で渡されたパラメータ

short stb = msg.ReadSTB();
```

Notes:

サービス・リクエスト発生の検出は、KI-VISA 内部の作業スレッド内で行なっています。

この動作は VISA の仕様によるものではなく、KI-VISA の実装上の仕様によるものです。

ハンドラ・イベントを無効にする

サービス･リクエストの通知機能を無効にするには `IEventManeger` インターフェースの `DisableEvent` を呼び出します。

```
Sub void IEventManager.DisableEvent(
    EventType type,
    EventMechanism mech,
```

```
    int customEventType
);
```

DisableEvent メソッドは、*type* パラメータで指定されたイベントタイプに関して *mech* パラメータで指定されたメカニズムでのイベント発生を停止するように VISA に指示します。*type* に EVENT_SERVICE_REQ のような特定イベントタイプを指定しても良いし、全てのイベントタイプを停止させるために ALL_ENABLED_EVENTS を指定しても良いでしょう。*mech* に関しても EVENT_HNDLR のように特定メカニズムを指定しても良いし、全てのメカニズムを停止させるために EVENT_ALL_MECH を指定しても良いでしょう。

更に、イベント・ハンドラ自体の登録抹消も行なえます。

```
void IEventManager.UninstallHandler(
    EventType type,
    int userHandle,
    int customEventType
);
```

ハンドラを示すパラメータが無い以外、InstallHandler とパラメータ・セットは同じです。

Notes:

サービス・リクエスト通知のイベント・コールバック・サポートは、KI-VISA 内部では作業用スレッドを利用して行なわれています。GPIB の場合は定期的(約 500ms 間隔)に自動的に並行処理でシリアル・ポーリングを実行します。USBTMC の場合はインタラプト・エンドポイントの常時監視をしてシリアル・ポーリングを並行処理します。LAN(VXI-11/HiSLIP)の場合は、それぞれのプロトコルで定義されている(計測器からの)コールバック・メカニズムを使用します。

ステータス・バイトの 0x40 のビット(RQS ビット)が新規にアサートされた場合は SRQ 発生とみなします。この時に、登録されて有効化されているイベント・シンクに対して即時イベントを発射する仕組みになっています。自動シリアル・ポーリングを行なう作業スレッドは、EVENT_SERVICE_REQ を指定して EnableEvent が呼び出された場合だけ作成されます。DisableEvent された場合、セカンダリ・スレッドも停止するので、自動シリアル・ポーリングは一切行なわれません。自動シリアル・ポーリングは副作用も大きいので、EVENT_SERVICE_REQ でイベント通知が有効になっている場合のみの限定的な動作になるように設計されています。自動シリアル・ポーリング動作の開始・停止の仕組みは VISA の仕様によるものではなく、KI-VISA の実装上の仕様によるものです。

アプリケーション側で用意されるイベント・シンクは、自動シリアル・ポーリングのスレッド・コンテキストでそのまま呼び出されます。VISA の SRQ イベント・コールバックは通常、アプリケーションの GUI スレッドとは異なるスレッド・コンテキストで呼び出されると考えて下さい。そのため、コールバックハンドラー内での動作に十分注意する必要があります。

VB6 と Excel VBA の統合環境では特に深刻です。VB6 と Excel VBA の VB エンジンはランタイム・ライブラリも含めてスレッド・セーフに出来ていますが、統合環境はそうではありません。従って、作業スレッドの文脈で動作する VB6/VBA コードをデバッガでブレークすると統合環境は確実にクラッシュします。

またデバッガでブレークしなくとも、コールバックハンドラーから直接 GUI 関連の操作を行なうと、やはり VB6/VBA ではクラッシュします。.NET Framework ではアクセス拒否(Access denied)の例外が発生します。コールバックハンドラーのような作業スレッドの文脈から GUI を直接操作することは出来ません。デリゲート機能などを必要に応じて使用し、GUI のスレッド文脈に移ってから GUI を更新するようにして下さい。、

但し、VISA COM API 全てスレッド・セーフ(Multithreaded Model)に設計されているため、作業スレッド文脈から VISA COM API を利用しても問題はありません。

サービス・リクエストのイベント機能は、コールバックの他にキューもサポートしています。イベント・キューを使ったサービス・リクエスト待ちは、IO 完了イベントと同様に WaitOnEvent メソッドを使用します。

12- KI-VISA SPY

KI-VISA SPY は、アプリケーションや計測器ドライバなどが実際に行う計測器との IO を傍受し、ログを表示又は記録するユーティリティです。KI-VISA Instrument Explorer のメニューバーから、**Tools | Launch SPY** を選択すると KI-VISA SPY が起動します。(SPY のショートカットはデスクトップにもあります。)

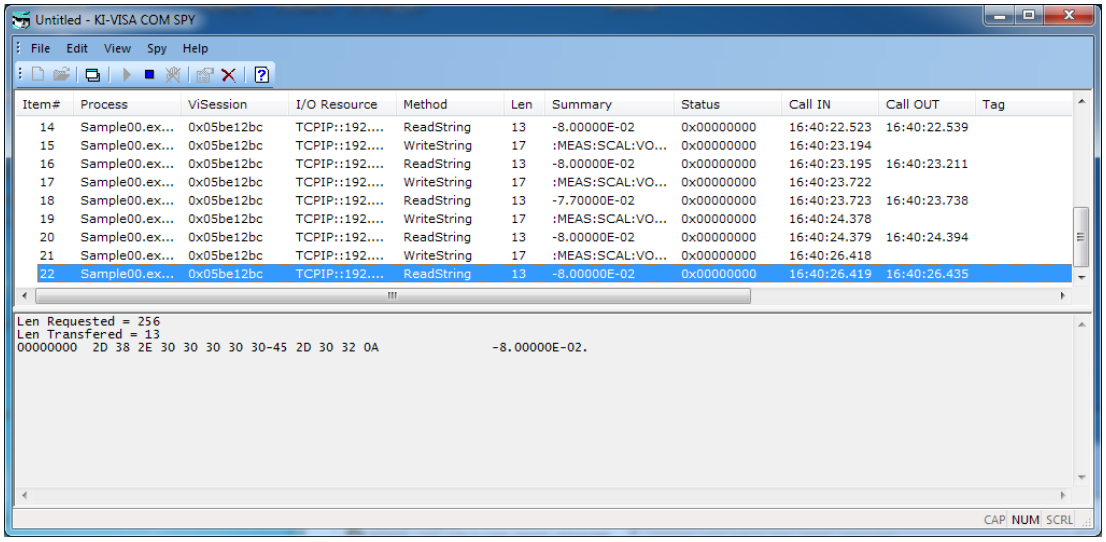

Figure 12-1 KI-VISA SPY

12-1 傍受の開始と停止

IO トラフィックの傍受を開始するには、KI-VISA SPY のメニューバーから **Spy ➔Capture On** を選択します。

傍受を停止する場合は **Spy ➔Capture Off** を選択します。但し、これは新規のデータログ取得を行わないという意味であり、高速で大量通信を行っている場合は SPY の画面表示(ログの追加と画面スクロール)は直ぐに停止しません。これは、キャプチャ自体は水面下でマルチスレッドで行なわれており、SPY 画面への表示とは別だからです。実際、キャプチャそのものよりも画面への表示のほうが遥かに時間が掛かります。

そのため、大量通信の場合はキャプチャを停止した後も暫くの間 SPY 画面が更新を続けます。この更新をとめたい場合には、**Spy➔No More Data** メニューを操作することで即座に停止することができます。

12-2 詳細情報の確認

画面上側のリストから見たい項目を選択すると、画面下側に詳細が表示されます。`Write/Read` 系統の各メソッドでは、IO の内容が 16 進ダンプでも表示されます。リストの各行はメソッド呼び出しの結果が色分けされています。黒文字表示の場合はメソッド呼び出しが成功(HRESULT=0)である事を示し、赤文字表示の場合はメソッド呼び出しが失敗(HRESULT<0)である事を示しています。青文字表示は、メソッドの呼び出し結果が警告(HRESULT>0)である事を示しています。いずれの場合も結果として HRESULT コードが存在するため、Status カラムの 16 進表示部にマウス・カーソルを持っていくと、ツールチップで HRESULT の詳細説明を見る事ができます。

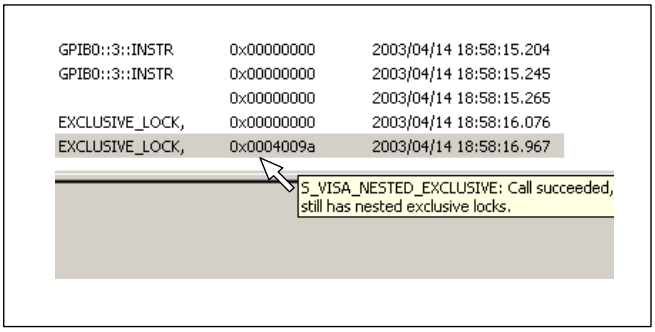

Figure 12-2 ステータス・コードのツールチップ表示

アイテムがグレー(淡色)表示されている場合、そのメソッドは呼び出しの途中であり、まだ結果が出ていない事を示します。例えば同期モードでの受信動作として IMessage インターフェースの ReadString メソッドを呼び出したあと計測器の応答が無い場合、タイムアウトするまでの間 KI-ん。メソッド呼び出しが終了した段階で、表示は黒(成功)、赤(失敗)、青(警告)、茶色(SCPI エラー)のいずれかに変わります。

茶色(SCPI エラー)だけは扱いが少し特殊で、これは通信自体のエラーではなく、直前に送ったエラー照会クエリに対する計測器からの応答が SCPI エラーを示唆している場合に着色表示されます。具体的には、SYST:ERR?又は*ESR?クエリに対する応答が 0 以外の値の示した時に SCPI エラーとみなされます。

Notes:

現在の KI-VISA の実装では、SPY を使用して傍受できるのはメソッド呼び出しのみです。プロパティのアクセスに関しては KI-VISA 内部で GetAttribute/SetAttribute メソッドを使用して実装されているため、これらのメソッド呼び出しとして SPY に傍受されます。

メソッド呼び出しの途中で(アイテムの表示が淡色表示の時に)キャプチャをオフした場合、メソッドの呼び出しが完了したあとも表示は更新されません。

SCPI エラー表示は、環境設定でそれをオンした場合のみの動作となります。また、非同期系メソッドでのエラー照会に関しては、クエリと応答の関連付けが難しいので着色表示にはなりません。またセミコロン区切りによる複合コマンドにエラー照会を混ぜた場合も機能しません。

SCPI エラー照会を行う WriteString ログ(SYST:ERR?又は*ESR?)とその応答を回収する ReadString ログが極端に離れている場合、SCPI エラーがカラー表示されない場合があります。

12-3 各表示カラムの説明

Table 12-1 各表示カラムの説明

Item#	ログの連番
Process	VISA コールを行ったプロセスの EXE 名 *32 は WOW64 上の 32bit プロセスを示す ツールチップではプロセス ID とスレッド ID が表示
ViSession	VISA-C API における VISA IO セッションの値
I/O Resource	該当する VISA アドレス (VISA エイリアスを使ったアクセスの場合も正規の VISA アドレスを表示)
Method	該当する VISA COM メソッド名
Len	O 処理が行われたデータの長さ(但しサイズの概念があるコールのみ)
Summary	IO の内容
Status	VISA コールによるステータスコード(マイナスはエラーを示す) ツールチップでは該当コードの説明文(英語)を表示
Call IN	メソッドコールを開始した時点のタイムスタンプ
Call OUT	メソッドコールがリターンした時点のタイムスタンプ (無表示はメソッドの所要時間が 1ms 未満又は計測されていない場合)
Tag	USBTMC、VXI-11、HiSLIP のプロトコル内部で使われるタグ (VISA を使用するアプリケーションからこの値にはアクセス出来ないが、計測器ファームウェアの実装時にデバッグ目的で利用される)

12-4 SPY ログの保存

キャプチャを停止した状態で、メニューバーから **File ➔Save As** を選択すると、SPY ログを保存する事ができます。保存された SPY ログはあとから KI-VISA SPY で閲覧する事ができます。

Notes:

現在の KI-VISA SPY の実装では、テキスト形式でのログ保存をサポートしていません。

KI-VISA SPY アプリケーションの画面が出ている状態では、**File ➔Exit** メニューは操作できません。システム・メニューから **Close** を選択するか、ウインドウ右上のクローズ・ボタンを操作した場合は、SPY アプリケーションは終了せず、単にウインドウを最小化します。(例外として、左 SHIFT キーを押しながらこの操作をした場合には SPY アプリケーションは終了します。) タスクトレイに KI-VISA SPY のアイコンが置かれているので、それをマウスで右クリックし、Exit を選択すると KI-VISA SPY は終了します。終了操作が二度手間になっていますが、不用意に SPY プリケーションを終了しないようにそのような設計になっています。タスクトレイ・アイコンを右クリックして **Restore Event Window** を選択すると SPY ウインドウが再度表示されます。

12-5 環境設定

Edit ➔Preferences メニューで環境設定を行う事ができます。この操作もキャプチャ停止中のみ可能です。

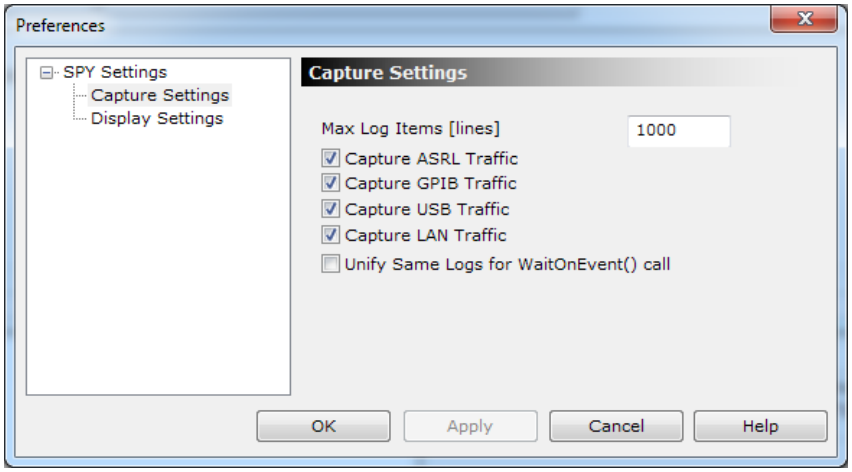

Figure 12-3 KI-VISA SPY 環境設定 (Capture Settings)

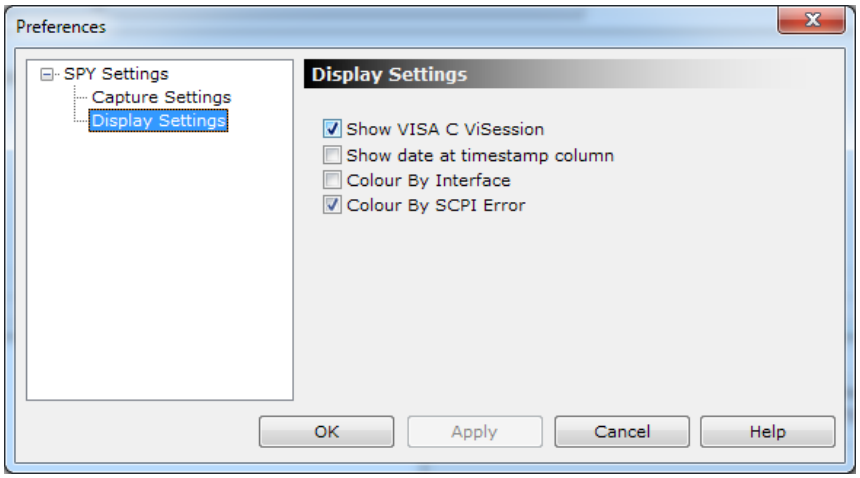

Figure 12-4 KI-VISA SPY 環境設定 (Display Settings)

Capture Sessings

このカテゴリはログの取り込みに関する設定です。**Max Log Items [lines]**では画面表示できる最大の行数を設定します。極端に大きくすると SPY アプリケーションが扱う情報量が大きくなって重くなるので注意してください。**Capture XXXX Traffic** は該当する IO リソースタイプのトラフィックをキャプチャするかどうかを選択します。**Unify Same Logs for WaitOnEvent() call** をチェックすると、`viWaitOnEvent()`関数のログで同一セッション・同一結果・同一時刻の物が連続した場合にそれを統合します。LabVIEW の非同期 VISA Write/Read 関数が連続してこの関数をコールしますが、膨大な数のログ(1ms あたり数百～数千回コール) を生成してしまうのでそれを間引くのに有効です。

Display Settings

このカテゴリは表示に関する設定です。**Show VISA C ViSession** をチェックすると ViSession の表示を行ないます。**Show date at timestamp colum** をチェックするとログ表示に日付時刻の両方を表示し、そうでない場合は時刻のみの表示となります。**Colour By Interface** をチェックすると IO インターフェースの種類ごとに色分け表示をします。**Colour By SCPI Errors** をチェックすると SCPI エラー照会クエリへの応答で SCPI エラーが示唆されている項目を色分け表示します。

12-6 常に最前面表示

SPY アプリケーションを常にデスクトップの最前面に表示したい場合は、メニューバーから **View ➔Always On Top** を選択します。この設定はトグル動作で切り替えられます。デフォルトでは常に最前面表示するようにはなっていません。

12-7 KI-VISA SPY の終了

KI-VISA SPY のウインドウが表示されている時は、終了操作(**File➔Exit**)してもタスク・トレイ・アインコンになるだけで実際には SPY は終了しません。SPY を終了するには、面倒ですが、タスク・トレイの状態から終了操作(**右クリック➔Exit**)して下さい。

13- IO インターフェースに依存した処理

13-1 シリアル・ポーリング(GPIB、USB、VXI-11、HiSLIP)

GPIB、USB(USBTMC)、LAN(VXI-11/HiSLIP)にはシリアル・ポーリングの機能があります。GPIB のシリアル・ポーリングは IEEE488.1 仕様で定義されているものなので特にここでは詳しくは説明しません。

USBTMC のシリアル・ポーリング機能は USBTMC-USB488 仕様で定義されたクラス・リクエスト READ_STATUS_BYTE によって実現されています。

VXI-11 のシリアル・ポーリング機能は、VXI-11 仕様で定義された device_readstb という RPC ファンクションで実現されています。

HiSLIP のシリアル・ポーリング機能は、内部の非同期通信ソケットを使って AsyncStatusQuery メッセージを使って実現されています。

VISA を使って計測器との IO 動作を行うプログラムからは、IO インターフェースの種別によらずどのシリアルポーリングも同じように見えます。ステータス・バイトは SCPI/IEEE488.2 仕様で 8 ビットが割り当てられています。ステータス・バイトの RQS ビット(0x40)がセットされている状態を特に、サービス・リクエスト発生状態といいます。

サービス・リクエストを検出する最も簡単な方法は、シリアル・ポーリングを行う事です。シリアルポーリングを行うには、IMessage インターフェース又は IAsyncMessage の ReadSTB メソッドを使います。どちらのインターフェースを使用しても動作は同じです。

```
short IMessage.ReadSTB();
```

Notes:

このメソッドの使い方は特に説明するまでもないでしょう。シリアル・ポーリングを行ってステータス・バイトを返します。サービス・リクエストを検出するには、ステータス・バイト中の RQS ビット(0x 40)が立っているかどうか調べればよいでしょう。

シリアル・ポーリング機能は GPIB/ USBTMC/LAN(VXI-11/HiSLIP)で接続された計測器のみ利用できます。

シリアル・ポーリングに似た機能として SCPI/IEEE488.2 で定義された"*STB?"共通クエリ・コマンドがありますが、これはシリアル・ポーリングと同じ動作をするわけではありません。またステータス・バイトのビット定義も一部異なります。

シリアル・インターフェース(ASRL)では、IOProtocol プロパティが VI_PROT_4882_STRS に明示的に指定されている場合には ReadSTB()メソッドを利用できますが、実際には*STB?クエリを送出しそのレスポンスを数値変換して返します。

13-2 デバイス・クリアの送出

GPIB、USBTMC、LAN(VXI-11/HiSLIP)ではセレクテッド・デバイス・クリア(Selected Device Clear, SDC)を送出する事ができます。シリアル(ASRL)では、IOProtocol プロパティが VI_PROT_4882_STRS の場合にはブレーク信号を送出しますが、そうでない場合にはエラーになります。

```
void IMessage.Clear();
```

13-3 デバイス・トリガの送出

GPIB、USBTMC、LAN(VXI-11/HiSLIP)ではデバイス・トリガ(Group Execute Trigger, GET)を送出する事ができます。

```
void IMessage.AssertTrigger(TriggerProtocol protocol);
```

AssertTrigger のパラメータ *protocol* に指定できるのは、GPIB/USBTMC/LAN の場合は TRIG_PROT_DEFAULT のみです。他の値は VXI bus などで使用するものです。シリアル(ASRL)でも IOProtocol プロパティが VI_PROT_4882_STRS の場合には AssertTrigger()を使用できますが、実際には*TRG コマンドを送出します。

13-4 リモート・ローカルの制御

GPIB、USBTMC、HiSLIP ではリモート・ローカルを切り替える為の ControlREN メソッドが IGpib インターフェース、IUsb インターフェース、IHislipInstr インターフェースにそれぞれ用意されています。

```
void IGpib.ControlREN( RENControlConst mode);
void IUsb.ControlREN( RENControlConst mode);
void IHislipInstr.ControlREN( RENControlConst mode);
```

パラメータに指定できるのは以下の表に示されるいずれかの一つです。

Table 13-1 ControlREN のパラメータ

RENControlConst	**説明**
GPIB_REN_DEASSERT	REN ラインをリセットする
GPIB_REN_ASSERT	REN ラインをセットする
GPIB_REN_GTL_AND_DEASSERT	Go To Local コマンドを送り、更に REN ラインをリセットする
GPIB_REN_ASSERT_AND_ADDRESS	REN ラインをセットし、更に計測器にアドレスコマンドを送る
GPIB_REN_LLO	Local Lockout コマンドを送る
GPIB_REN_ADDRESS_AND_LLO	計測器にアドレスコマンドを送り更に Local Lockout コマンドを送る
GPIB_REN_GTL	Go To Local コマンドを送る

Notes:

VXI-11 仕様にはリモート・ローカルを制御する機能が備わっていますが、VISA の ITcpipInstr インターフェースではそれを利用するメソッドが欠落しています。

13-5 コントローラ・インターフェース機能(GPIB)

これまでの説明では一度も登場していませんが、GPIB の場合は INSTR リソース以外に INTFC リソースを使う事が出来ます。INTFC というリソース・タイプは(計測器ではなく)GPIB ボードのコントローラ・インターフェースを操作するためのリソースです。

INTFC リソースをオープンするには次のように記述します。

```
IGpibIntfc intfc =
   (IGpibIntfc)rm.Open( "GPIB0::INTFC", AccessMode.NO_LOCK, 0, "");

IGpibIntfcMessage intfcm = (IGpibIntfcMessage)intfc;
```

ここで新しい COM インターフェース型が 2 つ登場します。IGpibIntfc インターフェースと IGpibIntfcMessage インターフェースです。これら 2 つの COM インターフェースは GPIB INTFC リソースでオープンした VISA セッションでは必ずサポートされています。

IGpibIntfc インターフェース

名前が示す通り、GPIB インターフェース機能を低レベル操作するためのインターフェースです。基本的に GPIB バスのユニライン・メッセージ操作(IFC、REN、ATN など)と ATN TRUE モードにおけるコマンドの送出を行う機能が用意されています。比較的利用頻度が高いと思われるものを下の表に示します。表に示されているもの以外のメソッド・プロパティについては KI-VISA オンライン・ヘルプを参照して下さい。

Table 13-2 IGpibIntfc インターフェースでよく使われるメソッド

メソッド	説明
SendIFC	IFC 信号を送出し、CIC(Controller In Charge)に移行する
ControlREN	REN ラインを制御する
ControlATN	ATN ラインを制御する
PassControl	指定されたアドレスの装置に CIC の権利を譲る
Command	ATN TRUE 状態で任意のコマンドを送信する

IGpibIntfcMessage インターフェース

GPIB インターフェース機能を低レベル操作するもののうち、デバイス・メッセージを操作するものが用意されたインターフェースです。メソッドとプロパティのセットは IMessage インターフェースとよく似ています。比較的利用頻度が高いと思われるものを下の表に示します。表に示されているもの以外のメソッド・プロパティについては KI-VISA オンライン・ヘルプを参照して下さい。

Table 13-3 IGpibIntfcMessage インターフェースでよく使われるメソッド

メソッド	説明
WriteString	デバイス・メッセージを送信する。ただし ATN TRUE 部分のアドレッシング・コマンドは送出せず、ATN FALSE 部分のみの送出をする。
Write	デバイス・メッセージをバイナリで送信する。ただし ATN TRUE 部分のアドレッシング・コマンドは送出せず、ATN FALSE 部分のみの送出をする。
ReadString	デバイスからレスポンス・メッセージを受信する。ただし ATN TRUE 部分のアドレッシング・コマンドは送出せず、ATN FALSE 部分のみを単純に受信する。
Read	デバイスからレスポンス・メッセージをバイナリで受信する。ただし ATN TRUE 部分のアドレッシング・コマンドは送出せず、ATN FALSE 部分のみを単純に受信する。
AssertTrigger	デバイス・トリガ(GET)を送信する。ただし、リスナー・アドレスの設定は行なわない。

WriteString/Write/ReadString/Read は IMessage インターフェースで用意されている同名メソッドとよく似た動作をします。しかし IGpibIntfcMessage 版では、アドレッシング・コマンドなどの ATN TRUE パートは一切送出されません。アドレッシング・コマンドは IGpibIntfc インターフェースの Command メソッドを使って事前に行っておく必要があります。例えば Command と WriteString を上手く組み合わせれば、複数リスナーを指定したメッセージの送出を行う事も可能です。

```
IResourceManager3 rm = new ResourceManagerClass();

IGpibIntfc intfc =
   (IGpibIntfc)rm.Open( "GPIB0::INTFC", AccessMode.NO_LOCK, 0, "");
IGpibIntfcMessage intfcm = (IGpibIntfcMessage)intfc;

intfc.SendIFC();
```

```
intfc.ControlREN( RENControlConst.GPIB_REN_ASSERT);

short intrAdrs = intfc.PrimaryAddress;

byte[] cmd = {
    0x3F,                       //UNL
    (byte)(0x40 + intrAdrs),    //TA0
    0x20 + 2,                   //LA2
    0x20 + 3,                   //LA3
    };

int r;
r = intfc.Command(ref cmd, 4);        //Sends UNL/TA0/LA2/LA3
r = intfcm.WriteString("CURR 3.0\n"); //Sends a dev-msg to instruments

intfc.Close();
```

Notes:

GPIB INSTR リソースと異なり、GPIB INTFC リソースでオープンされた VISA セッションは IMessage や IGpib インターフェースを実装していません。その代わりに、IGpibIntfc と IGpibIntfcMessage インターフェースを実装しています。

IGpibIntfc と IGpibIntfcMessage インターフェースは GPIB のボード・レベル・ファンクションを利用するためのものです。これらを使いこなすには IEEE488.1 に関する知識が必要になります。

NI-488.2 互換ボードでの GPIB INSTR リソースで、VISA セッションから Close()メソッドを呼び出すと、該当 GPIB ボードの REN(Remote Enable)ラインがディセーブル状態になります。この場合そのボードに接続された全ての GPIB 計測器がローカル操作に戻ります。この動作が計測アプリケーションにとって不都合になる場合は、INSTR リソースへの Close()操作をアプリケーションの終了時に行うように調整して下さい。

ここから先は、プログラミング言語ごとによって異なる内容を説明します。

14- 数値パラメータの書式制御 (FormattedIO488)

この章では VISA COM API が提供する書式制御コンポーネント、FormattedIO488 について説明します。

FormattedIO488 は、計測器通信インターフェースに対して直接 IO 処理を行うものではありません。しかし、計測器との IO 処理で必要な数値書式変換を行ったり、IEEE488.2 で定義されたブロック・データと配列変数との交換を行ったりする機能があります。

> Notes:
>
> FormattedIO488 を使用する際、計測器のコマンド体系が IEEE488.2 仕様に準拠している必要があります。そうでない古い設計の計測器では期待通りには動作しません。
>
> SCPI コマンド言語に対応した計測器は、IEEE488.2 仕様に必ず準拠しています。
>
> *IDN?クエリに応答できる計測器が全て IEEE488.2 や SCPI 仕様に準拠しているわけではありません。計測器をスキャンする多くのユーティリティ(NI-MAX、Agilent Connection Expert、KI-VISA Instrument Explorer 等)が*IDN?クエリを投げる為、それに反応させる為だけに*IDN?クエリを実装した計測器も多数存在します。

14-1 FormattedIO488 コンポーネント・オブジェクトの作成

FormattedIO488 コンポーネントのオブジェクトは、他の IO コンポーネントのようにリソース・マネージャから作成するのではなく、リソース・マネージャを作成するときと同じように new 演算子を使って単独に作成します。

```
// リソース・マネージャの作成
IResourceManager rm = new ResourceManagerClass();

//フォーマット IO488 の作成
IFormattedIO488 fmtIO = new FormattedIO488Class();

// VISA セッションのオープン
fmtIO.IO = (IMessage)rm.Open("GPIB0::3::INSTR", AccessMode.NO_LOCK, 0, "");
```

FormattedIO488 コンポーネントを操作する時の COM インターフェースは、IFormattedIO488 です。ここでは変数 fmtIO でそれを参照します。

VISA セッションをオープンする際は、受取変数を別に用意するのではなく IFormattedIO488 の IO プロパティで受けるようにします。これは FormattedIO488 のオブジェクトが計測器 IO の矛先を知るのに必要な設定です。

以降は、IMessage インターフェースのメソッドを使うのではなく、IFormattedIO488 インターフェースのメソッドを使って計測器と通信します。(勿論 IO プロパティ、つまり IMessage インターフェースを通じて通常の WriteString/ReadString 等を使用する事は可能です。)

14-2 コマンドとレスポンスの送受信

ここでは順を追って、コマンドの送受信の方法を説明していきます。

WriteString

単純にコマンド文字列を送出します。

```
// 単純なコマンド文字列の送信
fmtIO.WriteString("OUTP 1", true); //フラッシュ指定付なので改行コードの指定は不要
```

IMessage インターフェースにも同名のメソッドがありますが IFormattedIO488 では構文が違います。

void IFormattedIO488.WriteString(string *data*, bool *flushAndEND*);

WriteString メソッドでは、*data*(送出したいコマンド文字列)の他に、flushAndEND パラメータが追加されています。これを true に指定すると、このメソッドはコマンド文字列を計測器に直ちに送信します。その際、ターミネーション・キャラクタ (TerminationCharacter プロパティで指定中のコード、通常は 0x0A)及びターミネーション・マーク(GPIB の EOI 信号もしくは USBTMC/VXI-11/HiSLIP での相当コード)が自動的に付加されます。false にした場合は、この時点ではコマンド文字列は送出されず、バッファリングされたまま保留されます。

バッファリングされたまま送出されていないコマンドは、FlushWrite メソッドを明示的に呼び出すことで一気に送出する事ができます。

先ほどの例を、バッファリングとフラッシュの 2 段階に分けた書き方をすると次のようになります。

```
// 単純なコマンド文字列の送信
fmtIO.WriteString("OUTP 1", false);
fmtIO.FlushWrite(true); //バッファ送出、コマンド文字列は終端される
```

ここで、FlushWrite メソッドには sendEND パラメータが付きます。

void IFormattedIO488.FlushWrite (bool *sendEND*);

FlushWrite メソッドは、そのパラメータに関わらずフラッシュ処理(溜まっているバッファ内容の送出)を行います。sendEND パラメータが true の場合は、ターミネーション・キャラクタ LF(0x0A)及びターミネーション・マーク(GPIB の EOI 信号もしくは USBTMC/VXI-11/HiSLIP での相当コード)が自動的ターミネーション・キャラクタとして LF(0x0A)を送出しますが、false の場合はそれらを一切送出しません。

```
// 単純なコマンド文字列の送信
fmtIO.WriteString("OUTP 1", false);
fmtIO.FlushWrite(false); //バッファ送出、コマンド文字列は終端されない
```

この例の場合は、コマンドが終端されないため、送出された"OUTP 1"というコマンド文字列は、計測器側では恐らく処理されないでしょう。従って、ASCII 文字列ベースのコマンドを送出する際は、実用的には必ず終端させる必要があります。

但し、応用的な使い方をした場合、次のような例は期待通りに動作します。

```
//セミコロン区切りの複合コマンドを断片でバッファリング(計測器が SCPI 複合構文に対応する事)
fmtIO.WriteString(":VOLT 30;", false); //最後にセミコロンが付く
fmtIO.WriteString(":CURR 1.0;", false); //最後にセミコロンが付く
fmtIO.WriteString(":OUTP 1", false);

//バッファ内容を一気に送出
fmtIO.FlushWrite(true);
```

この例では、各コマンドが一行ずつ送出されるのではなく、SCPI の複合コマンドの形式で送られています。(セミコロンがセパレータに使われている事、コマンドの記述が最上位ノードからの文脈に強制されている点に注意。)

これは次のような記述と等価な動作です。

```
fmtIO.WriteString(":VOLT 30;:CURR 1.0;:OUTP 1", true);
```

ReadString

単純にレスポンスを読み込み、そのまま文字列として返します。

```
// クエリを送信してそのレスポンスを読み込む
fmtIO.WriteString("*IDN?", true);
string strIDN = fmtIO.ReadString();
```

string IFormattedIO488.ReadString();

IMessage インターフェースの同名メソッドと違い、受信最大バイト数を指定する引数がありません。(受信最大バイト数は FormattedIO488 によって自動的に決定されますが、詳細は後述します。)

WriteNumber

単純な(配列でない)スカラ量のパラメータを送る場合に使用します。

```
void IFormattedIO488.WriteNumber(
  object data,
  IEEEASCIIType type,
  bool flushAndEND
);
```

object には数値変数(又は直値)などのオブジェクトを渡します。type には該当するデータ型を指定します。flushAndEnd が true の場合はターミネーション・キャラクタ(及びターミネーション・マーク)と共にすぐに送出を行い、false の場合はバッファリングだけ行います。

但し、コマンドとセットで送れるわけではなく、あくまでもパラメータ部分のみです。先行して送っている"VOLT"コマンドの後ろにスペースが空いている点に注意して下さい。SCPI/IEEE488.2 計測器では、コマンドとパラメータの間に空白が必要です。

```
// DOUBLE 型パラメータを送信する
fmtIO.WriteString( "VOLT ", false);
fmtIO.WriteNumber( 50.1, IEEEASCIIType.ASCIIType_R8, true);
```

例では、最終的に"VOLT 50.1\n"が送出されます。

ReadNumber

単純な(配列でない)スカラ量のレスポンスを受け取る場合に使用します。

```
object IFormattedIO488.ReadNumber(
  IEEEASCIIType type,
  bool flushToEND
);
```

type には予期されるデータ型を指定します。flushToEnd が true の場合はターミネーション条件に遭遇する(ターミネーション・キャラクタ又はターミネーション・マークを受信)するまで、待ち続けます。

```
// クエリを送信
fmtIO.WriteString("MEAS:VOLT?", true);

// DOUBLE 型レスポンスを受信し直接 DOUBLE 型変数に格納する
double dMeasVolt =
     (double)fmtIO.ReadNumber(IEEEASCIIType.ASCIIType_R8, true);
```

WriteList

リストとして事前に列挙された複数のパラメータを送る場合に使用します。パラメータ同士のセパレータも任意に指定できます。

```
void IFormattedIO488.WriteList(
  ref object data,
  IEEEASCIIType type,
  string listSeparator,
  bool flushAndEND
);
```

object には数値や文字列を格納した配列を渡します。type には該当するデータ型を指定します。listSeparator には、送出する際のセパレータ文字を指定します。送出するコマンドにもよりますが多くの場合はカンマかスペースでしょう。flushAndEnd が true の場合はターミネーション・キャラクタ(及びターミネーション・マーク)と共にすぐに送出を行い、false の場合はバッファリングだけ行います。

但し、コマンドとセットで送れるわけではなく、あくまでもパラメータ部分のみです。先行して送っている"SYST:TIME"コマンドの後ろにスペースが空いている点に注意して下さい。SCPI/IEEE488.2 計測

器では、コマンドとパラメータの間に空白が必要です。また、セパレータに 2 種類以上の文字を同時に指定する事はできません。

```
// リストの送信
short[] lst = { 23, 59, 45, };
Object o = lst;
fmtIO.WriteString("SYST:TIME ", false);
fmtIO.WriteList( ref o, IEEEASCIIType.ASCIIType_I2, ",", true);
```

この例では、"SYST:TIME 23,59,45\n"が送出されます。

ReadList

一定のセパレータで区切られた複合レスポンスを分解して配列として返します。

```
object IFormattedIO488.ReadList(
  IEEEASCIIType type,
  string listSeparator
);
```

type には予期されるデータ型を指定します。listSeparator には、レスポンスとして予期されるセパレータ文字を指定します。複数フィールドを(カンマ区切りで)返すクエリに対してはカンマを、単一レスポンスを返すクエリを複数結合させた複合クエリの場合にはセミコロンを、それぞれ指定するとよいでしょう。またこれらの条件が合成されたケース(レスポンスのセパレータがカンマとセミコロンの混在の時)では、listSeparator に",;"のような複数を指定する事もできます。
(WriteList メソッドでは複数指定はエラーになります。)

```
// レスポンスのリストを分解する
fmtIO.WriteString("*IDN?", true);
String[] s = (String[])fmtIO.ReadList(IEEEASCIIType.ASCIIType_BSTR, ",");

//受け取り配列を System.Array 型で受け取っても良い
//Array a = (Array)fmtIO.ReadList(IEEEASCIIType.ASCIIType_BSTR, ",");
```

この例では、"*IDN?"クエリに対するレスポンス(機種情報を示す 4 つのフィールドがある)を、配列の各要素に分解して格納します。例えば、s[0]には計測器ベンダー名、s[1]には機種名、といった具合です。複数の数値型フィールドを返すような場合(複数の計測データを照会するような場合)であれば、受け取り配列を double[]にしてデータ型指定を ASCIIType_R8 (8 バイト型の実数、つまり double)にすれば動作します。

WriteIEEEBlock

これは SCPI/IEEE488.2 仕様で規定化されている任意ブロック・データを扱うものです。ブロック・データは、ASCII と対比した言い方として、BINARY データ(但し 2 進数という意味ではなく)と表現される場合もあります。WriteIEEEBlock メソッドは、WriteNumber や WriteList と違って、コマンド部分とパラメータ部分の両方を同時指定する事が出来ます。

```
void IFormattedIO488.WriteIEEEBlock(
  string command,
  object data,
  bool flushAndEND

);
```

command は送出したいコマンドを指定します。data は配列を指定します。flushAndEnd が true の場合はターミネーション・キャラクタ(及びターミネーション・マーク)と共にすぐに送出を行い、false の場合はバッファリングだけ行います。

Notes:

IEEE488.2 ブロック・データは、通常の ASCII パラメータとは違った形式になっています。抽象的に表現すると、下記のようになります。

#<digit_width><length_info><block_data>

#0<block_data>ブロック・データは必ず#で始まります。続く<digit_width>は 1 桁の数字(1~9)になります。これは、その後ろに来る<length_info>の表現上の桁数を示しています。例えば先頭が#4 で始まった場合、digit_width=4 という意味になり、これは<length_info>が 4 桁で表現されている事を意味します。

例 1(固定長ブロック・データ):

#41024<データ><データ><データ>...<データ>

この例では、長さ 1024 を 4 桁の形式で宣言し、実際のデータ(バイト)が 1024 個続くと見なします。

具体的なコマンド例では、

:TRAC:DATA #41024<データ 0><データ 1><データ 2>...<データ 1023>,AUTO<LF^END>

といった感じになります。パラメータ(この例では第一パラメータがブロック、第二パラメータが AUTO というキャラクタ)自体には改行コードは付きませんが、プログラム・メッセージ全体の終端としてターミネーションされます。

<digit_width>は 1~9 の数字であると書きましたが、例外的に 0(ゼロ)の場合も存在します。これは、ヘッダ部分で明示的に全長を表現出来ない(この時点では決定できない)ようなケースです。例えば、半永久的にストリームデータを流し続けるような場合などがこれに該当します。

例 2(不定長ブロック・データ):

#0<データ><データ><データ>...<データ>

この場合<length_info>は 0 桁での表現(つまり表現しない)となり、直接ブロック・データが続きます。この形式では、不定長連続通信が継続する限りターミネーションされる事はありません。継続するのをやめたい場合には、最後の<データ>の後に LF(0x0A)とターミネーション・マーク(END)が付きます。この方式は、キャラクタではないターミネーション・マーク(END、end フラグなど)を表現できる限られた通信インターフェース(GPIB, USBTMC, VXI-11, HiSLIP 等)でのみ成立します。シリアルやソケットでは成立しません。

具体的なコマンド例では、

:TRAC:DATA #0<データ 0><データ 1><データ 2>...<データ 876><LF^END>

といった感じになります。但し、パラメータ(この例では唯一のパラメータがブロック・データ)自体に改行コードが付いているのではありません。あくまでもプログラム・メッセージ全体に対するターミネーションです。不定長パラメータは IEEE488.2 仕様書で「Indefinite Length Data(不定長データ)」に該当するため、そのパラメータより後に別のパラメータが続いたり、セミコロン区切りで別のコマンドを複合させることは禁じ手になっています。

ここで"TRAC:DATA"コマンドにブロック・データを渡す例を幾つか見ていきます。このコマンドが何をする物なのかはここでは重要ではありません。単にブロック・データを使用するコマンドの一例と考えて下さい。実際のブロック・データのバイナリイメージと照合するため、KI-VISA SPY で確認した実際のトラフィックと合わせて確認します。ここでは話を単純にするため、僅か 8 バイトのブロック・データを計測器に送信します。

```
// 送信するバイト配列
byte[] abyBlockData = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,};

// 配列を送信し LF で終端する
fmtIO.WriteIEEEBlock("TRAC:DATA", abyBlockData, false);
fmtIO.FlushWrite(true);
```

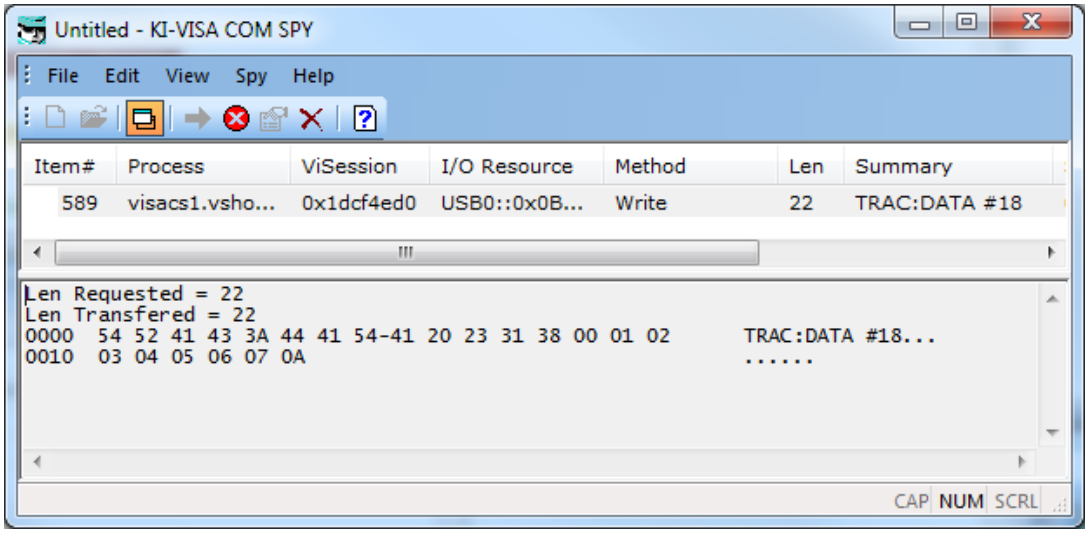

Figure 14-1 ブロック・データの送出(byte[]配列)

この例では、byte[]配列(全部で 8 バイト)を渡しています。#の後に「1 桁」で「8 バイト長」を宣言しているのがわかります。そのあとに続く 8 バイトが実際に渡された配列のイメージです。この例では最後に LF(0x0A)が付加されている点に注意して下さい。これは FlushWrite メソッドに true が渡された為にそのような動作になっています。

次は short(16 ビット整数)の例です。

```
// 送信する 16 ビット整数配列
short[] aiBlockData = { 0x0001, 0x0203, 0x0405, 0x0607, };

// short 配列を送信し LF で終端する(ビッグ・エンディアン)
fmtIO.InstrumentBigEndian = true;
fmtIO.WriteIEEEBlock("TRAC:DATA", aiBlockData, false);
fmtIO.FlushWrite(true);
```

WriteIEEEBlock メソッドは様々な配列型を引数に指定できるので、short 配列でも同じように記述できます。但し、16 ビット以上の幅を持つデータ型ではエンディアンに注意しなければなりません。エンディアンは InstrumentBigEndian プロパティで指定できます。true でビッグ、false でリトルになります。(デフォルトはビッグ。) この例ではビッグ・エンディアンを指定するので、実行すると次のようなログを観測できます。

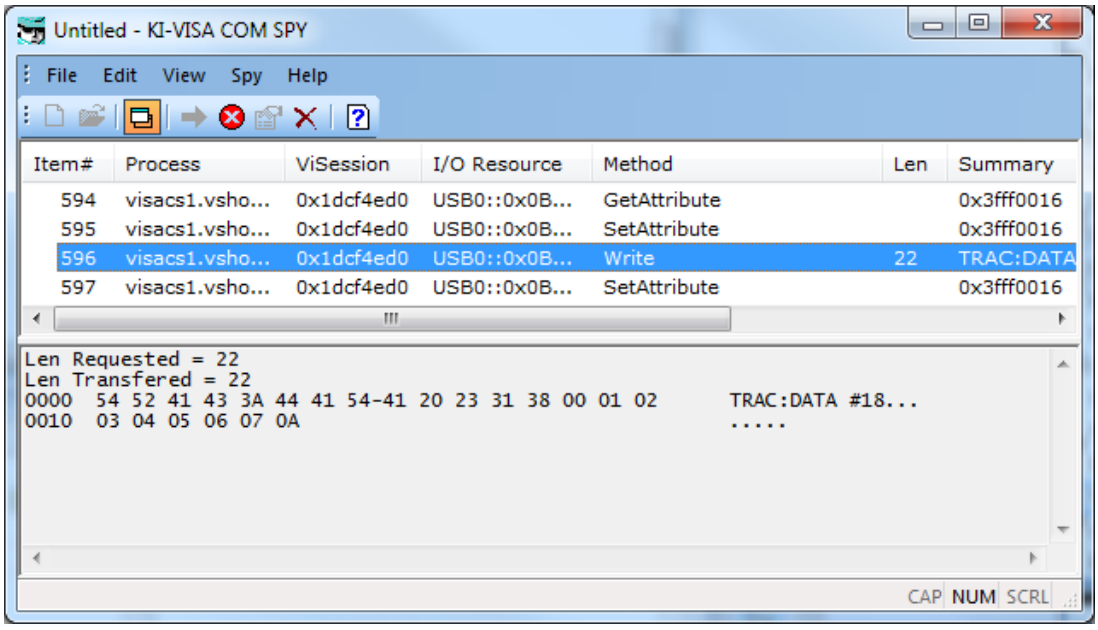

Figure 14-2 ブロック・データの送出(short[]配列、Big Endian)

リトルに指定した場合は、次のようになります。

```
//送信する 16 ビット整数配列
short[] aiBlockData = { 0x0001, 0x0203, 0x0405, 0x0607, };

// short 配列を送信し LF で終端する(リトル・エンディアン)
fmtIO.InstrumentBigEndian = false;
fmtIO.WriteIEEEBlock("TRAC:DATA", aiBlockData, false);
fmtIO.FlushWrite(true);
```

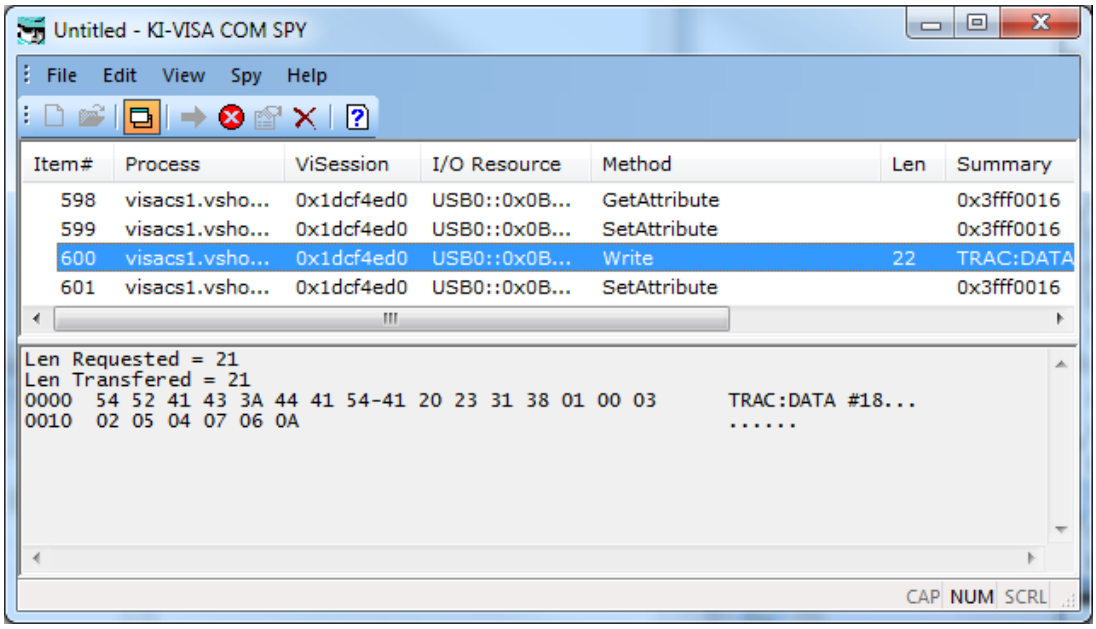

Figure 14-3 ブロック・データの送出(short[]配列、Little Endian)

Notes:

エンディアンとは、2 バイト以上の大きさを持つデータをメモリ上又は伝送経路上に並べた時、最上位バイトと最下位バイトのどちらが先に来るかを意味する言葉です。例えば UInt32 型のデータが数値 0x12345678 であった場合、メモリや伝送経路上で 12 34 56 78 という順序に並んだものをビッグ・エンディアン、78 56 45 12 という順序に並んだものをリトルエンディアンと呼びます。

コンピュータ・システムのメモリ上では、CPU のタイプによってエンディアンが決定します。モトローラ系の CPU ではビッグ、インテル系の CPU ではリトルになります。伝送経路上の場合は、複数 CPU の都合が混在するため、その伝送技術の背景になった関連技術や標準を規定した企業や業界の都合等で決まる場合が多いようです。例えば、UNIX ベースで SUN Micrososystems が大きく関与したネットワーク転送(LAN)ではビッグが標準であり、Microsoft や Intel が大きく関与した USB 関連技術ではリトルが標準です。SCPI/IEEE488.2 のような計測器インターフェースの場合は、Hewlett Packard(現 Agilent Technologies)の HP-IB という歴史的背景からビッグが標準のようです。

ReadIEEEBlock

このメソッドは SCPI/IEEE488.2 で規定されているブロック・データのレスポンスを受信します。

```
object IFormattedIO488.ReadIEEEBlock(
  IEEEBinaryType type,
  bool seekToBlock
  bool flushToEnd
);
```

type は予期されるバイナリブロックのデータ型を指定します。seekToEnd が true の場合、受信データ中のシャープ文字(#)に遭遇するまで検索を行います。(#に遭遇するまでのデータは捨てられます。) これは、受信したデータの(#で始まる)ヘッダ部分より手前に(#を含まない)ゴミが含まれていても正常動作する為の措置です。seekToEnd が false の場合はこの検索を行いませんが、受信データが#文字で始まっていなければなりません。flushToEnd が true の場合は、ターミネーション条件に遭遇するまで待ちます。

計測器から送られてくるブロック・データは、SCPI/IEEE488.2 仕様に従って長さ情報もしくは明示的な終端条件が明確化されていますが、データ形式に関する情報は含まれていません。従って、アプリケーションがデータを受信する際に、予期されるデータ形式を明示的に指定する必要があります。通常はレスポンスデータを要求するクエリの構文やそれ以前に計測器に指定してある諸条件等から形式は推測出来る筈です。(少なくとも計測器のマニュアルには明記されています。)

```
//ブロック・データを受信し、ビッグ・エンディアンの short 配列として解釈する
fmtIO.InstrumentBigEndian = true;
fmtIO.WriteString("TRAC:DATA?", true);
short[] x =
    (short[])fmtIO.ReadIEEEBlock(IEEEBinaryType.BinaryType_I2, true, true);
```

この例では、ビッグ・エンディアンを指定し、ブロック・データのレスポンスを要求する"TRAC:DATA?"クエリを送出し、そのレスポンスを受信します。ここでは、ブロック・データが short[]配列に相当すると事前に分かっている物とします。

実際の計測器が返すレスポンスと ReadIEEEBlock メソッドに指定する形式が合致しない場合は、予期しないデータで充填されている場合があるので注意して下さい。

Notes:

配列に Char, UInt16, UInt32 は使えません。これは FormattedIO488 の制約です。このコンポーネントが提供する IFormattedIO488 インターフェースは、元々Automation 互換(OLE Automation 互換)の VARIANT 型に依存している為、Automation で扱えない「符号なし 16 ビット」、「符号なし 32 ビット」、「符号付き 8 ビット」の配列は扱う事ができません。この場合は、byte, Int16, Int32 等の類似した型で便宜上宣言し、必要に応じてタイプキャストを行う必要があります。また、ブロック・データが計測器依存の構造体になっているようなケースにも対応できません。この場合も byte[]配列などに置き替えて処理する必要があります。

14-3 バッファ・サイズ

FormattedIO488 の Write/Read 関連のメソッドでは、パラメータ次第でバッファリングが行われる事を説明しました。この場合のバッファ・サイズは幾つになるでしょうか。

このバッファ・サイズは、VISA 仕様書には明確に記述されていませんが、どうやら 1000 に成っているようです。このサイズを設定するには、SetBufferSize メソッドを使用します。

```
void IFormattedIO488.SetBufferSize(
  BufferMask mask,
  int size
);
```

mask には、IO_OUT_BUF、IO_IN_BUF、又はその論理 OR を指定します。それぞれ、出力バッファ(Write 関連)、入力バッファ(Read 関連)、同時指定を意味します。

Write 系の動作で、送出したいコマンド(又はクエリ)のサイズが出力バッファ・サイズ以内に収まっている場合、flushAndEND パラメータが false を指定されている場合には、コマンドを送出せずにバッファリングのみを行います。しかし、それを連続的に動作させ、出力バッファ・サイズに空きが無くなった場合、FormattedIO488 は自動的にコマンドの送出を行います。flushAndEND が true に指定されている場合は、バッファ・サイズとは関係なくすぐにコマンドは送出されます。

Read 系の動作では、バッファ・サイズは IO コンポーネントに対する IMessage.Read メソッド呼び出しでの最大受信バイト数指定に利用されます。従って、このバッファ・サイズを極端に小さくすると、本来の受信レスポンスが尻切れ扱いになる可能性があり、結果として ReadNumber や ReadList などが期待通りの結果を返さない場合があります。。

Notes:

Write 動作で flushAndEND=true に設定した場合でも、バッファ・サイズが極端に小さい時は IO 動作が異なります。例えばバッファ・サイズを 5 にした場合、送出されるコマンドは 5 バイトずつ小分けされて最終的な IO 処理に回されます。

Write 動作でバッファ・サイズをデフォルトよりも小さくする必要があるのは、例えば計測器の RS232 インターフェースで計測器側の受信バッファが極端に小さい場合などです。この場合、計測器側のバッファオーバーランを避けるためにバッファを小さくして運転する方法が考えられます。フロー制御等が作動する場合には、この方法でコマンド送信を小刻みにする出来る場合があります。

Write 動作でバッファ・サイズをデフォルトよりも大きくする必要があるのは、長大なブロック・データを間欠ではなく一気に送出したい場合です。

Read 動作でバッファ・サイズをデフォルトよりも小さくする必要があるのは、意図的に間欠受信をするような場合です。

Read 動作でバッファ・サイズをデフォルトよりも大きくする必要があるのは、長大なブロック・データを間欠ではなく一気に受信したい場合です。

15- 数値パラメータの書式制御 (.NET Framework)

この章では.NET Framework の標準クラス・ライブラリを使用した書式制御の手法と注意点について説明します。

15-1 数値→文字列変換

計測器のコマンドに数値パラメータを渡す場合、元のデータを格納する数値型クラスの ToString()メソッドを使います。

```
//文化依存を適用しない
Thread.CurrentThread.CurrentCulture = CultureInfo.InvariantCulture;

// volt 設定値を送信する
double dVset = 13.515;
int r = msg.WriteString ( "VOLT " + dVset.ToString("0.000") + "\n");
```

ToString メソッドには書式を指定する事が可能です。上の例では、小数点以下 3 桁に書式が指定されています。指定が無い場合は自動的に行われます。(SCPI コマンドによるプログラミング・インターフェースを実装する計測器では、特に書式を指定しなくても殆どの場合正しく解釈されます。)

ここで、冒頭に書かれているカルチャー(文化)の指定に注意して下さい。.NET Framework では、ToDouble や ToString のような文字列・数値変換を行うメソッドは、期待される数値表現書式が全て OS の言語・地域の設定に依存します。しかしヨーロッパや南米で動作する PC では、小数点記号が必ずしもドット(.)ではありません。一方、計測器の通信インターフェースで受け渡しされる際の数値パラメータに小数点がつく場合、それは必ず米国式(日本も同じ)である必要があります。そのため、これらの地域は、.NET Framework のデフォルト動作では数値文字列変換が正しく動作しません。そこで文化依存動作をキャンセルするため、冒頭でカルチャーの無効化を行っています。

Notes:

カルチャーを明示的に設定しない場合、OS の言語・地域設定に依存した動作になります。

明示的に設定されたカルチャーは、現在のスレッドにのみ適用されます。計測器との IO 動作を GUI とは別のスレッドに配置するようなマルチ・スレッド・アプリケーションでは、そのスレッド毎にカルチャーの無効化を指定する必要があります。

アプリケーションの GUI が言語・地域設定を尊重したい場合(例えば南米地域で動作するアプリケーションが小数点記号に正しくカンマを表示させたい場合)、カルチャーの無効化は計測器コマンドとレスポンスの数値変換処理を行う部分に局所化する必要があります。そうでないと GUI 上での数値表現が米国式に強制されてしまいます。

前の章で説明した FormattedIO488 は、言語・地域の設定の影響は受けません。

15-2 文字列→数値変換

計測器からのレスポンスに数値表現が含まれている場合、数値型変数への変換は Convert クラスのスタティック・メソッドを使います。

```
//文化依存を適用しない
Thread.CurrentThread.CurrentCulture = CultureInfo.InvariantCulture;

//curr/volt 計測を開始しその測定値を照会する
r = msg.WriteString ( "READ:CURR?;VOLT?\n");
string rd = msg.ReadString(100);

//レスポンスをセパレータで分解
char[] splitters = new char[] { ';', ',', '\n' };
string[] asToken = rd.Split(splitters);

//分解されたトークンを数値に変換
```

```
double dCurr = Convert.ToDouble(asToken[0]);
double dVolt = Convert.ToDouble(asToken[1]);
```

この例では"READ:CURR?"及び"READ:VOLT?"の複合クエリを使って電流・電圧の計測値を照会し、その複合レスポンス(それぞれの値はセミコロン区切り)を取得します。実際のレスポンスは、例えば"+4.0010E+00;+1.2543E+01\n"のようになるかもしれません。

Split は string クラスのメソッドで、指定されたスプリッタ(セパレータ)でトークンを分解し、結果を string[]配列で返します。ここでは、セミコロン(;)、カンマ(,)、改行(LF)を同時に char[]配列で指定しています。

最後に Convert クラスのスタティック・メソッド ToDouble で double 型への変換をしています。整数を期待する場合は、ToInt16, ToInt32 などを使っても良いでしょう。

ここでもカルチャーの無効化が明示的設定されている点に注意して下さい。

16- プログラミング言語依存の補足説明(C#)

この章では C#固有の内容を説明します。このガイドブックで扱う C#のバージョンは 2008 とします。C#は.NET Framework を利用したプログラミング言語なので、ネイティブ・コードで実装された VISA COM DLL とは直接リンクされませんが、Primary Interop Assembly (通称 PIA)と呼ばれる DLL モジュールを使用して、間接的にリンクします。

16-1 コンポーネント・オブジェクトの作成方法

C#でコンポーネント・オブジェクトの作成を行うには下記サンプルのように行います。ここではリソース・マネージャ・オブジェクトを作成する場面を例に説明します。

```
using Ivi.Visa.Interop;
.
.
.
IResourceManager3 rm = new ResourceManagerClass();
```

リソース・マネージャを作成するには ResourceManagerClass クラスに対して new 演算子を使います。またプログラムの冒頭で using 擬似命令が使われている点に注意して下さい。この using 擬似命令は必須ではありませんが、もし書かない場合は、VISA COM タイプ・ライブラリが提供するネーム・スペース Ivi.Visa.Interop を全ての型名とシンボル名に明示的に書かなければなりません。

16-2 インターフェースの参照

リソース・マネージャの Open メソッドは、C#スタイルの構文として、以下のような形式で宣言されます。

```
IVisaSession IResourceManager3.Open(
    string ResourceName,
    AccessMode mode,
    int initTimeout,
    string OptionString
);
```

メソッドの戻り値は IVisaSession です。しかし実際のプログラム(アプリケーション、計測器ドライバなど)では多くの場合 IMessage 型の変数に戻り値を代入します。

```
.
.
IMessage msg;
msg = (IMessage)rm.Open("GPIB0::3::INSTR", AccessMode.NO_LOCK, 0, "");
.
.
.
msg.Close();
```

IResourceManager3.Open メソッドは IVisaSession インターフェースへの参照を返します。しかしそれを受取る変数が IMessage 型で宣言されているため、IMessage 型でタイプキャストを行います。C#では異なる COM インターフェースに対して暗黙のタイプキャストを行いません。しかし、明示的なタイプキャストを行うと、水面下では指定されたインターフェース型への問い合わせが行われます。"GPIB0::3::INSTR"のような INSTR リソースでオープンされた VISA セッションは IMessage インターフェースを必ずサポートしているため、結果的に IMessage インターフェースへの問い合わせが成功し、変数 *msg* にそのインターフェース参照が格納されます。

16-3 配列の扱い

C#での配列の扱いは.NET Framework で用意された System.Array の機能を使います。配列は Array 型で記述する方法と、型が判っている場合は[]を用いて記述する方法があります。例えばリソース・マネージャの FindRsrc()メソッドは、有効な VISA アドレスを文字列配列で返します。

```
System.Array a = rm.FindRsrc("?*INSTR");

string[] s = rm.FindRsrc("?*INSTR");

//どちらの記述も正しい
```

これ以外にも、複数の計測値を問い合わせるようなメソッドが double 配列を返すような場合も似たような方法で記述できます。

16-4 エラー・コードの原則

計測器を制御するアプリケーションの場合、実行時に計測器と正常に通信できたかどうかを確認するのは重要な事です。どんなに堅牢に設計・実装されたアプリケーションでも、実行時に計測器のパワーが落ちていたり通信ケーブルが外れていれば、計測器との通信は正常に行えません。

VISA では全てのメソッド呼び出しとプロパティへのアクセスに対して一貫したエラー・コード体系を持っています。VISA C API 及び VISA COM API の全てのファンクションは、その戻り値として 32 ビット・コードを使用します。

Notes:

VISA C API では、各関数の戻り値は全て ViStatus 型 (=int 32 ビット整数型)と定義されています。一方 VISA COM API では、COM の標準に合わせて全てのメソッド(及びプロパティ・アクセス・メソッド)の戻り値は HRESULT 型(=int 32 ビット整数型)と定義されています。VISA C API と VISA COM API の戻り値はコードのビット割り当てが若干異なりますが、同じ意味のステータス・コードは相互変換可能な関係にあります。本書は VISA COM API を中心とした説明をしていますので、HRESULT 型のステータス・コードを前提にして説明します。

C#では long は 64 ビット整数型、int は 32 ビット整数型です。従って C#の場合には、HRESULT は int 型と等価であると解釈して下さい。

一般に COM メソッド呼び出し時とプロパティ・アクセス時のステータス・コード(HRESULT)は次のように分類されます。VISA COM もその例外ではありません。

Table 16-1 HRESULT の大まかな分類

値の範囲	**説明**
HRESULT=0	成功
HRESULT>0	警告
HRESULT<0	エラー

HRESULT がマイナスの場合は、.NET Framework は COM 例外を発生します。VISA COM API のメソッドを呼び出した結果、或いはプロパティにアクセスした結果として COM 例外が発生した場合、通常は、計測器との通信が失敗したか、呼び出しのパラメータに問題があったか、或いは VISA の動作状態がその呼び出しを拒否した場合などが考えれます。

16-5 物理的な戻り値

C#では、COM インターフェースを通じてメソッドを呼び出したりプロパティにアクセスした場合、物理的な戻り値である HRESULT をプログラムから直接扱う事はできません。例えば、既に説明したリソース・マネージャの Open メソッドは、物理的には次のような形で、IDL 言語(COM インターフェース

のタイプ・ライブラリ・ソース)で記述されています。C#ユーザが一般に考えているメソッドの宣言とはかなり違いますが、次に示す IDL 言語で書かれた COM メソッドの宣言と完全に等価なものです。

```
HRESULT Open(
    [in] BSTR ResourceName
    [in] AccessMode mode,
    [in] long initTimeout,
    [in] BSTR OptionString
    [out,retval] IVisaSession** vi
);
```

ここでは物理的な戻り値が HRESULT であり、C#では「戻り値」であると信じられていた IVisaSession への参照が(二重参照、ポインタへのポインタとして)最終パラメータになっています。ここで最終パラメータに[out,retval]という方向属性が与えられている事に注意して下さい。out は、このパラメータがリソース・マネージャ・オブジェクトから呼び出し元(アプリケーション又は計測器ドライバ側)に向かって返される事を意味します。そして retval はこの戻り値が COM メソッドとしての表面的な戻り値である事を意味します。C#での COM メソッドは、このような「表面的な」構文で扱われます。そして物理的な戻り値である HRESULT は水面下の処理にまわされるのです。

もうひとつ例を見てみましょう。

```
HRESULT get_Timeout(
    [out,retval] long* pVal
);

HRESULT put_Timeout(
    [in] long newVal
);
```

これは、IMessage インターフェースで使われる Timeout プロパティの構文を物理的に表現したものです。プロパティのとは結局のところ get_又は put_をプレフィックスに持つメソッドに過ぎません。get_メソッドだけが提供される場合、そのプロパティはリード・オンリーになります。逆に put_メソッドだけが提供される場合、そのプロパティはライト・オンリーとなります。ここでも物理的な戻り値は HRESULT になっており、C#からは水面下の処理になります。

このように、全ての COM メソッド(プロパティの get_メソッド、put_メソッドを含む)は必ず物理的な戻り値が HRESULT になっています。そして C#では、メソッドの戻り値がマイナスになった場合は COM 例外として処理されるように設計されています。具体的なエラー原因は Microsoft で一般的に定義されているものと、VISA COM のような特定コンポーネントで拡張定義しているものとがあります。

> Notes:
>
> IDL 言語で記述される long 型は C++言語と同じ 32 ビットですが、C#では int(同じく 32 ビット整数)型になります。
>
> VISA COM で定義されている HRESULT コードの詳細は、KI-VISA オンライン・ヘルプ又は本書の付録に掲載されています。

16-6 エラー・トラッピング

C#ではメソッドの呼び出し直後に HRESULT の値が水面下でチェックされ、マイナスの値になっている場合は COM 例外を発生します。COM 例外を捕獲するには try/catch ステートメントを使います。

```
try {
    msg = (IMessage)rm.Open( "GPIB0::3::INSTR", AccessMode.NO_LOCK, 0, "");
    .
    .
    r = msg.WriteString( "CURR 20.0\n");
    r = msg.WriteString( "VOLT 5.5\n");
    .
    .
    .
}
catch( System.Runtime.InteropServices.ComException ex) {
    MessageBox.Show(
```

```
        exp.Message, "error 0x" + Convert.ToString(ex.ErrorCode, 16));
    .
    .
    .
}
finally {
    msg.Close();
    .
}
```

try ブロック内で呼び出された全ての COM メソッド呼び出し及びプロパティのアクセスでは、.NET Framework によって HRESULT がチェックされます。HRESULT がマイナスの場合、.NET Framework は COM 例外を発生させ、catch ブロックに処理がジャンプします。

catch ブロックでは、System.Runtime.InteropServices.ComException 型の例外オブジェクト *ex* を通じてエラー(COM 例外)の原因を探る事ができます。

System.Runtime.InteropServices.ComException 型で定義されているもので特に有用な情報は、ErrorCode プロパティと Message プロパティです。それぞれ HRESULT コード及びエラー・メッセージ文字列を返します。

Notes:

try/catch ブロックを使わずにプログラムを実行すると、例外発生時にデバッガがそれを捕獲します。デバッガを使用せずにスタンド・アロンで実行された場合はプログラムがクラッシュする事になり、プログラムは強制終了します。最終的なアプリケーションでは必ずエラー・トラッピングを行い、プログラムが強制終了される事のないようにして下さい。

finally は必ずしも必要ありませんが、Close()メソッドの呼び出しのように、成功・失敗にかかわらず最終的には呼び出さなければならないような処理があれば、使うと便利です。

16-7 イベント・シンクの作成方法

サービス・リクエスト・イベントなどで COM のイベント・コールバック機能を使う場合、VISA COM ではイベント・シンクを作成しなければなりません。イベント・シンクとは、そのイベントが要求するコールバック・インターフェース(この場合 IEventHandler インターフェース)を装備する、任意のコンポーネント・オブジェクトです。そのためには IEventHandler インターフェースを実装するコンポーネント・クラスを作成しなければなりません。

イベント・シンクの作成

Visual C#統合環境のメニューバーから **Project | Add Class** を選択して下さい。**Add New Item** ダイアログが表示されるので、**Templates** から **Clsss** を選択します。また生成されるクラスの名前を MyEventSink.cs としておきましょう。そして **OK** ボタンをクリックすると MyEventSink クラスが自動生成されプロジェクトに追加されます。

そしてクラス・モジュール MyEventSink.cs には以下のような内容を記述します。

```
using System;
using Ivi.Visa.Interop;

namespace VisaCom_cs2  ←この部分は実際にはプロジェクト名になる
{
   /// <summary>
   /// Summary description for MyEventSink.
   /// </summary>
   public class MyEventSink : IEventHandler
   {
       public MyEventSink()
       {
       }
```

```
        public  void HandleEvent(
             IEventManager vi,
             IEvent ev,
             int userHandle)
        {
             IMessage msg = (IMessage)vi;

             short stb;
             stb = msg.ReadSTB();

             // TODO: Add your own job depending on the stb value
        }
    }
}
```

MyEventSink クラスは IEventHandler インターフェースから派生したクラスとして作成します。IEventHandler には唯一のメソッドとして HandleEvent があります。このメソッドは VISA が用意するものではなく、作成するアプリケーション側で用意する責任があります。それはアプリケーションが呼び出すメソッドではなく、VISA がアプリケーションに向かって(逆方向に)呼び出すメソッドだからです。

16-8 ターゲット OS の設定

.NET Framework で動作するアプリケーションは、動作対象の OS をビルド時に指定することが出来ます。Visual Studio 2008 のメニューバーから **Project➔ *<プロジェクト名>* Properties...**を選択し **Build** タブを選択すると **Platform Target** という項目があるので、ターゲットを指定します。

ターゲットの指定と動作 OS との関係を下の表に示します。ターゲットの指定と動作 OS との関係を下の表に示します。新規プロジェクトを作成した直後のデフォルト指定は Any CPU です。

Table 16-2　ターゲット・プラットフォームの指定

ターゲット指定	**x86 OS での動作**	**x64 OS での動作**
Any CPU	32bit アプリとして動作	64bit アプリとして動作
x86	32bit アプリとして動作	32bit アプリとして WOW64 環境で動作
x64	実行不可	64bit アプリとして動作

17- プログラミング言語依存の補足説明(Visual Basic.NET)

この章では Visual Basic.NET 固有の内容を説明します。このガイドブックで扱う Visual Basic のバージョンは 2008 としますが、基本的な記述方法は 2002 以降でも同様です。Visual Basic.NET は.NET Framework を利用したプログラミング言語なので、ネイティブ・コードで実装された VISA COM DLL とは直接リンクされませんが、Primary Interop Assembly (通称 PIA)と呼ばれる DLL モジュールを使用して、間接的にリンクします。

17-1 コンポーネント・オブジェクトの作成方法

Visual Basic.NET でコンポーネント・オブジェクトの作成を行うには下記サンプルのように行います。ここではリソース・マネージャ・オブジェクトを作成する場面を例に説明します。

```
Imports Ivi.Visa.Interop
.
.
.
Dim rm As IResourceManager3 = New ResourceManager()
```

リソース・マネージャを作成するには、ResourceManager クラスに対して New 演算子を使います。またプログラムの冒頭で Imports 擬似命令が使われている点に注意して下さい。この Imports 擬似命令は必須ではありませんが、もし書かない場合は VISA COM タイプ・ライブラリが提供するネーム・スペース Ivi.Visa.Interop を全ての型名とシンボル名に明示的に書かなければなりません。

17-2 インターフェースの参照

リソース・マネージャの Open メソッドは、Visual Basic.NET から見ると、以下のような形式で宣言されています。

```
Function Open(
    ResourceName As String,
    [mode As AccessMode,]
    [initTimeout As Integer,]
    [OptionString As String]
) As IVisaSession
```

メソッドの戻り値は IVisaSession です。しかし実際のプログラム(アプリケーション、計測器ドライバなど)では多くの場合 IMessage 型の変数に戻り値を代入します。

```
.
.
Dim msg As IMessage
msg = rm.Open( "GPIB0::3::INSTR", AccessMode.NO_LOCK, 0, "")
.
.
.
msg.Close()
```

Open メソッドは IVisaSession インターフェースへの参照を返します。しかしそれを受取る変数が IMessage 型で宣言されているため、IMessage 型へ暗黙のタイプキャストが行われます。暗黙的であれ明示的であれ、タイプキャストを行うと、水面下では指定されたインターフェース型への問い合わせが行われます。"GPIB0::3::INSTR"のような INSTR リソースでオープンされた VISA セッションは IMessage インターフェースを必ずサポートしているため、結果的に IMessage インターフェースへの問い合わせが成功し、変数 *msg* にそのインターフェース参照が格納されます。

17-3 配列の扱い

Visual Basic.NET での配列の扱いは.NET Framework で用意された System.Array の機能を使います。配列は Array 型で記述する方法と、型が判っている場合は()を用いて記述する方法があります。例えばリソース・マネージャの FindRsrc()メソッドは、有効な VISA アドレスを文字列配列で返します。

```
Dim a As System.Array = rm.FindRsrc("?*INSTR")

Dim s As String() = rm.FindRsrc("?*INSTR")

//どちらの記述も正しい
```

これ以外にも、複数の計測値を問い合わせるようなメソッドが Double 配列を返すような場合も似たような方法で記述できます。

17-4 エラー・コードの原則

計測器を制御するアプリケーションの場合、実行時に計測器と正常に通信できたかどうかを確認するのは重要な事です。どんなに堅牢に設計・実装されたアプリケーションでも、実行時に計測器のパワーが落ちていたり通信ケーブルが外れていれば、計測器との通信は正常に行えません。

VISA では全てのメソッド呼び出しとプロパティへのアクセスに対して一貫したエラー・コード体系を持っています。VISA C API 及び VISA COM API の全てのファンクションは、その戻り値として 32 ビット・コードを使用します。

Notes:

VISA C API では、各関数の戻り値は全て ViStatus 型 (=Long 32 ビット整数型)と定義されています。一方 VISA COM API では、COM の標準に合わせて全てのメソッド(及びプロパティ・アクセス・メソッド)の戻り値は HRESULT 型(=Long 32 ビット整数型)と定義されています。VISA C API と VISA COM API の戻り値はコードのビット割り当てが若干異なりますが、同じ意味のステータス・コードは相互変換可能な関係にあります。本書は VISA COM API を中心とした説明をしていますので、HRESULT 型のステータス・コードを前提にして説明します。

Visual Basic.NET では Long は 64 ビット整数型、Integer は 32 ビット整数型です。従って Visual Basic.NET の場合には、HRESULT は Integer 型と等価であると解釈して下さい。

一般に COM メソッド呼び出し時とプロパティ・アクセス時のステータス・コード(HRESULT)は次のように分類されます。VISA COM もその例外ではありません。

Table 17-1 HRESULT の大まかな分類

値の範囲	**説明**
HRESULT=0	成功
HRESULT>0	警告
HRESULT<0	エラー

HRESULT がマイナスの場合は、.NET Framework は COM 例外を発生します。VISA COM API のメソッドを呼び出した結果、或いはプロパティにアクセスした結果として COM 例外が発生した場合、通常は、計測器との通信が失敗したか、呼び出しのパラメータに問題があったか、或いは VISA の動作状態がその呼び出しを拒否した場合などが考えれます。

17-5 物理的な戻り値

Visual Basic.NET では、COM インターフェースを通じてメソッドを呼び出したりプロパティにアクセスした場合、物理的な戻り値である HRESULT をプログラムから直接扱う事はできません。例えば、既に説明したリソース・マネージャの Open メソッドは、物理的には次のような形で、IDL 言語(COM イ

ンターフェースのタイプ・ライブラリ・ソース)で記述されています。Visual Basic.NET ユーザが一般に考えているメソッドの宣言とはかなり違いますが、次に示す IDL 言語で書かれた COM メソッドの宣言と完全に等価なものです。

```
HRESULT Open(
    [in] BSTR ResourceName
    [in] AccessMode mode,
    [in] long initTimeout,
    [in] BSTR OptionString
    [out,retval] IVisaSession** vi
);
```

ここでは物理的な戻り値が HRESULT であり、Visual Basic.NET では「戻り値」であると信じられていた IVisaSession への参照が(二重参照、ポインタへのポインタとして)最終パラメータになっています。ここで最終パラメータに[out,retval]という方向属性が与えられている事に注意して下さい。out は、このパラメータがリソース・マネージャ・オブジェクトから呼び出し元(アプリケーション又は計測器ドライバ側)に向かって返される事を意味します。そして retval はこの戻り値が COM メソッドとしての表面的な戻り値である事を意味します。Visual Basic.NET での COM メソッドは、このような「表面的な」構文で扱われます。そして物理的な戻り値である HRESULT は水面下の処理にまわされるのです。

もうひとつ例を見てみましょう。

```
HRESULT get_Timeout(
    [out,retval] long* pVal
);

HRESULT put_Timeout(
    [in] long newVal
);
```

これは、IMessage インターフェースで使われる Timeout プロパティの構文を物理的に表現したものです。プロパティのとは結局のところ get_又は put_をプレフィックスに持つメソッドに過ぎません。get_メソッドだけが提供される場合、そのプロパティはリード・オンリーになります。逆に put_メソッドだけが提供される場合、そのプロパティはライト・オンリーとなります。ここでも物理的な戻り値は HRESULT になっており、Visual Basic.NET からは水面下の処理になります。

このように、全ての COM メソッド(プロパティの get_メソッド、put_メソッドを含む)は必ず物理的な戻り値が HRESULT になっています。そして Visual Basic.NET では、メソッドの戻り値がマイナスになった場合はランタイム・エラーとして処理されるように設計されています。具体的なエラー原因は Microsoft で一般的に定義されているものと、VISA COM のような特定コンポーネントで拡張定義しているものとがあります。

> Notes:
>
> IDL 言語で記述される long 型は C++言語と同じ 32 ビットですが、Visual Basic.NET では Integer(同じく 32 ビット整数)型になります。
>
> VISA COM で定義されている HRESULT コードの詳細は、KI-VISA オンライン・ヘルプ又は本書の付録に掲載されています。

17-6 エラー・トラッピング

Visual Basic.NET ではメソッドの呼び出し直後に HRESULT の値が水面下でチェックされ、マイナスの値になっている場合は COM 例外を発生します。COM 例外を捕獲するには Try/Catch ステートメントを使います。

```
Try
    msg = rm.Open( "GPIB0::3::INSTR", AccessMode.NO_LOCK, 0, "")
    .
    .
    r = msg.WriteString( "CURR 20.0" & vbLf)
    r = msg.WriteString( "VOLT 5.5" & vbLf)
    .
    .
```

```
    .
Catch ex As System.Runtime.InteropServices.ComException
    MessageBox.Show(_
        ex.Message, "error 0x" + Convert.ToString(ex.ErrorCode, 16))
    .
    .
    .
Finally
    msg.Close()
    .
    .
End Try
```

Notes:

Visual Basic.NET では従来の Visual Basic と同様の On Error Goto ステートメントを使うことができます。しかし、Try/Catch ステートメントを使ったほうが処理ロジックをより明確に表現する事ができるので、こちらの書き方を推奨します。また、このアプローチは、C#言語でもほとんど同じように記述できます。Visual Basic.NET は言語的には BASIC 言語ですが、動作プラットフォームや処理ロジックなどのスタイルは、むしろ同じ.NET フレーム環境の仲間である C#に近いのです。

Try ブロック内で呼び出された全ての COM メソッド呼び出し及びプロパティのアクセスでは、.NET Framework によって HRESULT がチェックされます。HRESULT がマイナスの場合、.NET Framework は COM 例外を発生させ、Catch ブロックに処理がジャンプします。

Catch ブロックでは、System.Runtime.InteropServices.ComException 型の例外オブジェクト *ex* を通じてエラー(COM 例外)の原因を探る事ができます。

System.Runtime.InteropServices.ComException 型で定義されているもので特に有用な情報は、ErrorCode プロパティと Message プロパティです。それぞれ HRESULT コード及びエラー・メッセージ文字列を返します。

Notes:

Try/Catch ブロックを使わずにプログラムを実行すると、例外発生時にデバッガがそれを捕獲します。デバッガを使用せずにスタンド・アロンで実行された場合はプログラムがクラッシュする事になり、プログラムは強制終了します。最終的なアプリケーションでは必ずエラー・トラッピングを行い、プログラムが強制終了される事のないようにして下さい。

Finally は必ずしも必要ありませんが、Close()メソッドの呼び出しのように、成功・失敗にかかわらず最終的には呼び出さなければならないような処理があれば、使うと便利です。

17-7 イベント・シンクの作成方法

サービス・リクエスト・イベントなどで COM のイベント・コールバック機能を使う場合、VISA COM ではイベント・シンクを作成しなければなりません。イベント・シンクとは、そのイベントが要求するコールバック・インターフェース(この場合 IEventHandler インターフェース)を装備する、任意のコンポーネント・オブジェクトです。そのためには IEventHandler インターフェースを実装するコンポーネント・クラスを作成しなければなりません。

イベント・シンクの作成

Visual Basic.NET 統合環境のメニューバーから **Project | Add Class** を選択して下さい。**Add New Item** ダイアログが表示されるので、**Templates** から **Clsss** を選択します。また生成されるクラスの名前を MyEventSink.vb としておきましょう。そして **OK** ボタンをクリックすると MyEventSink クラスが自動生成されプロジェクトに追加されます。

そしてクラス・モジュール MyEventSink.vb には以下のような内容を記述します。

```
Imports Ivi.Visa.Interop

Public Class MyEventSink
```



```
    Implements IEventHandler

    Private Sub HandleEvent( _
       ByVal vi As IEventManager, _
       ByVal ev As IEvent, _
       ByVal userHandle As Integer) Implements IEventHandler.HandleEvent

        Dim msg As IMessage
        msg = vi
        Dim stb As Short
        stb = msg.ReadSTB  'ステータス・バイトを取得する

    End Sub

    ' TODO: Add your own job depending on the stb value

End Class
```

MyEventSink クラスは IEventHandler インターフェースを実装する必要があるため、Implements キーワードを使用します。IEventHandler には唯一の追加メソッドとして HandleEvent があります。このメソッドは VISA が用意するものではなく、作成するアプリケーション側で用意する責任があります。それはアプリケーションが呼び出すメソッドではなく、VISA がアプリケーションに向かって(逆方向に)呼び出すメソッドだからです。

17-8 ターゲット OS の設定

.NET Framework で動作するアプリケーションは、動作対象の OS をビルド時に指定することが出来ます。Visual Studio 2008 のメニューバーから **Project➔ *<プロジェクト名>* Properties...**を選択し **Compile** タブを選択すると **Advanced Compile Options...**ボタンがあるのでそれをクリックして **Advanced Compiler Settings** ダイアログを表示します。その中に **Target CPU** という項目があるので、ターゲットを指定します。

ターゲットの指定と動作 OS との関係を下の表に示します。新規プロジェクトを作成した直後のデフォルト指定は Any CPU です。

Table 17-2　ターゲット・プラットフォームの指定

ターゲット指定	**x86 OS での動作**	**x64 OS での動作**
Any CPU	32bit アプリとして動作	64bit アプリとして動作
x86	32bit アプリとして動作	32bit アプリとして WOW64 環境で動作
x64	実行不可	64bit アプリとして動作

18- プログラミング言語依存の補足説明(Visual Basic 6.0)

この章では Visual Basic 6.0(以下 VB6 と省略)固有の内容を説明します。VB6 でのプログラミング手法はそのまま Excel VBA マクロにも適用できます。VB6 は.NET ではなく旧来のネイティブコードを生成するツールなので、VISA COM DLL とは直接リンクされます。Primary Interop Assembly は使用しません。

18-1 コンポーネント・オブジェクトの作成方法

VB6 ではコンポーネント・オブジェクトの作成を行うには下記サンプルのように行います。ここではリソース・マネージャ・オブジェクトを作成する場面を例に説明します。

```
Dim rm As IResourceManager3

Set rm As New ResourceManager
```

リソース・マネージャを作成するには、ResourceManager クラスに対して New 演算子を使います。

18-2 インターフェースの参照

リソース・マネージャの Open メソッドは VB6 から見ると、以下のような形式で宣言されています。

```
Function Open(
    ResourceName As String,
    [mode As AccessMode,]
    [initTimeout As Long,]
    [OptionString As String]
) As IVisaSession
```

メソッドの戻り値は IVisaSession です。しかし実際のプログラム(アプリケーション、計測器ドライバなど)では多くの場合 IMessage 型の変数に戻り値を代入します。

```
.
.
Dim msg As IMessage
Set msg = rm.Open("GPIB0::3::INSTR", NO_LOCK, 0, "")
.
.
.
msg.Close()
```

Open メソッドは一旦 IVisaSession インターフェースへの参照を返します。しかしそれを受取る変数が IMessage 型で宣言されているため、IMessage 型への暗黙のタイプキャストが行われます。COM インターフェースを変数で受ける場合、VB6 では必ず Set 文が使われます。
"GPIB0::3::INSTR"のような INSTR リソースでオープンされた VISA セッションは IMessage インターフェースを必ずサポートしているため、結果的に IMessage インターフェースへの問い合わせが成功し、変数 *msg* にそのインターフェース参照が格納されます

18-3 配列の扱い

VB6 での配列は、一般的な COM で使われる配列の実装と全く同じものです。(VB6 自体が COM 技術によって成り立っているので当然と言えば当然です。)例えばリソース・マネージャの FindRsrc メソッドが返す配列は、通常の文字列配列として扱う事ができます。

```
Dim r() As String
r = rm.FindRsrc("?*INSTR")
```

18-4 エラー・コードの原則

計測器を制御するアプリケーションの場合、実行時に計測器と正常に通信できたかどうかを確認するのは重要な事です。どんなに堅牢に設計・実装されたアプリケーションでも、実行時に計測器のパワーが落ちていたり GPIB ケーブルが外れていれば、計測器との通信は正常に行えません。

VISA では全てのメソッド呼び出しとプロパティへのアクセスに対して一貫したエラー・コード体系を持っています。VISA C API 及び VISA COM API の全てのファンクションは、その戻り値として 32 ビット・コードを使用します。

> Notes:
>
> VISA C API では、各関数の戻り値は全て ViStatus 型 (=Long 型)と定義されています。一方 VISA COM API では、COM の標準に合わせて全てのメソッド(及びプロパティ・アクセス・メソッド)の戻り値は HRESULT 型 (=Long 型)と定義されています。VISA C API と VISA COM API の戻り値はコードのビット割り当てが若干異なりますが、同じ意味のステータス・コードは相互変換可能な関係にあります。本書は VISA COM API を中心とした説明をしていますので、HRESULT 型のステータス・コードを前提にして説明します。

一般に COM メソッド呼び出し時とプロパティ・アクセス時のステータス・コード(HRESULT)は次のように分類されます。VISA COM もその例外ではありません。

Table 18-1　　HRESULT の大まかな分類

値の範囲	説明
HRESULT=0	成功
HRESULT>0	警告
HRESULT<0	エラー

HRESULT がマイナスの場合は、VB6 はランタイム・エラーを発生します。VISA COM API のメソッドを呼び出した結果、或いはプロパティにアクセスした結果としてランタイム・エラーが発生した場合、通常は、計測器との通信が失敗したか、呼び出しのパラメータに問題があったか、或いは VISA の動作状態がその呼び出しを拒否した場合などが考えれます。

18-5 物理的な戻り値

VB6 では、COM インターフェースを通じてメソッドを呼び出したりプロパティにアクセスした場合、物理的な戻り値である HRESULT をプログラムから直接扱う事はできません。例えば、既に説明したリソース・マネージャの Open メソッドは物理的には次のような形で、IDL 言語(COM インターフェースのタイプ・ライブラリ・ソース)で記述されています。VB6 ユーザが一般に考えているメソッドの宣言とはかなり違いますが、次に示す IDL 言語で書かれた COM メソッドの宣言と完全に等価なものです。

```
HRESULT Open(
    [in] BSTR ResourceName
    [in] AccessMode mode,
    [in] long initTimeout,
    [in] BSTR OptionString
    [out,retval] IVisaSession** vi
);
```

ここでは物理的な戻り値が HRESULT であり、VB6 では「戻り値」であると信じられていた IVisaSession への参照が(二重参照、ポインタへのポインタとして)最終パラメータになっています。ここで最終パラメータに[out,retval]という方向属性が与えられている事に注意して下さい。out は、このパラメータがリソース・マネージャ・オブジェクトから呼び出し元(アプリケーション又は計測器ドライバ側)に向かって返される事を意味します。そして retval はこの戻り値が COM メソッドとしての表面的な戻り値である事を意味します。VB6 での COM メソッドは、このような「表面的な」構文で扱われます。そして物理的な戻り値である HRESULT は水面下の処理にまわされるのです。

もうひとつ例を見てみましょう。

```
HRESULT get_Timeout(
   [out,retval] long* pVal
);
HRESULT put_Timeout(
   [in] long newVal
);
```

これは、IMessage インターフェースで使われる Timeout プロパティの構文を物理的に表現したものです。プロパティのとは結局のところ get_又は put_をプレフィックスに持つメソッドに過ぎません。get_メソッドだけが提供される場合、そのプロパティはリード・オンリーになります。逆に put_メソッドだけが提供される場合、そのプロパティはライト・オンリーとなります。ここでも物理的な戻り値は HRESULT になっており、VB6 からは水面下の処理になります。

このように、全ての COM メソッド(プロパティの get_メソッド、put_メソッドを含む)は必ず物理的な戻り値が HRESULT になっています。そして VB6 では、メソッドの戻り値がマイナスになった場合はランタイム・エラーとして処理されるように設計されています。具体的なエラー原因は Windows で一般的に定義されているものと、VISA COM のような特定コンポーネントで拡張定義しているものとがあります。

Notes:

VISA COM で定義されている HRESULT コードの詳細は、KI-VISA オンライン・ヘルプ又は本書の付録に掲載されています。

18-6 エラー・トラッピング

Visual Basic ではメソッドの呼び出し直後に HRESULT の値が水面下でチェックされ、マイナスの値になっている場合はランタイム・エラーを発生します。ランタイム・エラーを捕獲するには On Error Goto ステートメントを使います。

```
    On Error Goto IOEXP
    r = msg.WriteString( "CURR 20.0" & vbLf)
    r = msg.WriteString( "VOLT 5.5" & vbLf)
    .
    .
    .
IOEXP:
    MsgBox Err.Number, Err.Description
    .
    .
```

On Error Goto ステートメントで予めエラー・ジャンプ先を指定すると、ランタイム・エラー発生時にその場所へプログラムがジャンプします。ランタイム・エラーの発生は抑制したいがどこにもジャンプさせたくない場合は、On Error Resume Next を使います。エラー・トラッピング自体をやめたい場合は On Error Goto 0 を使います。エラー発生時は、Visual Basic で定義済みの Err オブジェクトを使って、エラー・コード(HRESULT)や詳細説明(文字列)を取得する事が出来ます。

Notes:

エラー・トラッピングを無効にしたまま(On Error Goto 0 の状態)でプログラムを実行しランタイム・エラーが発生すると、VB6 の統合環境(インタプリタ環境)と Excel VBA 環境では、デバッガがプログラムを一時停止しエラーが発生した場所のソースコードを表示します。しかし VB6 でスタンド・アロン EXE を生成しそれを実行中にランタイム・エラーが発生すると、エラー発生を示すメッセージ・ボックスが表示されたあと、プログラムは強制終了します。最終的なアプリケーションでは必ずエラー・トラッピングを行い、プログラムが強制終了される事のないようにして下さい。

18-7 イベント・シンクの作成方法

サービス・リクエスト・イベントなどでイベント・コールバック機能を使う場合、VISA COM ではイベント・シンクを作成しなければなりません。イベント・シンクとは、そのイベントが要求するコールバック・インターフェース(この場合 IEventHandler インターフェース)を装備する、任意のコンポーネント・オブジェクトです。そのためには IEventHandler インターフェースを実装するコンポーネント・クラスを作成しなければなりません。

イベント・シンクの作成

VB6 の場合は、メニューバーから **Project | Add Class Module** を選択して下さい。Excel VBA の場合は、メニューバーから **Insert | Class Module** を選択して下さい。いずれの場合もクラス・モジュールが追加されます。

追加されたクラス・モジュールの Name プロパティを、MyEventSink としましょう。つまり今から作成するのは MyEventSink というコンポーネント・クラスです。Excel の場合は、Instancing プロパティがありますが、Private にしておいて下さい。そしてクラス・モジュールには以下のような内容を記述します。

```
Option Explicit
Implements VisaComLib.IEventHandler

Private Sub IEventHandler_HandleEvent( _
  ByVal vi As IEventManager, ByVal ev As IEvent, ByVal userHandle As Long)

    On Error Resume Next

    Dim msg As VisaComLib.IMessage
    Dim stb As Integer
    Set msg = vi
    stb = msg.ReadSTB()

    ' TODO: Add your own job depending on the stb value

End Sub
```

Implements キーワードは VB6 で利用可能なステートメントで、クラス・モジュールで特定の COM インターフェースを実装する際に使用します。このサンプルでは、このコンポーネント・クラス (MyEventSink 型)に IEventHandler インターフェースを実装するという事を宣言しています。IEventHandler には唯一の追加メソッドとして HandleEvent があります。このメソッドは VISA が用意するものではなく、作成するリケーション側で用意する責任があります。それはアプリケーションが呼び出すメソッドではなく、VISA がアプリケーションに向かって(逆方向に)呼び出すメソッドだからです。

VB でインターフェース・メソッドを実装する場合、関数名として *<interfacename>_<methodname>*のように、インターフェース名とメソッド名をアンダースコア("_")で連結します。

Notes

VISA COM のイベントコールバック機構はマルチスレッドで動作するため、VISA からのコールバックの呼び出しは GUI のスレッドとは別の作業スレッドから行われます。VB6 の言語とランタイム・ライブラリ自体はマルチスレッドをサポートしますが、VB6 統合環境はそれをサポートしません。その為、コールバック関数の中をデバッグしようとすると確実に統合環境がクラッシュします。また、デバッグを伴わないコンパイル済みコードであっても、作業スレッド内から GUI を操作すると殆どの場合クラッシュします。

19- プログラミング言語依存の補足説明(C++/CLI)

この章では C++/CLI(.NET Framework で動作するマネージド版 C++)固有の内容を説明します。このガイドブックで扱う C++/CLI のバージョンは VC++2008 とします。C++/CLI は.NET Framework を利用したプログラミング言語なので、ネイティブ・コードで実装された VISA COM DLL とは直接リンクされませんが、Primary Interop Assembly (通称 PIA)と呼ばれる DLL モジュールを使用して、間接的にリンクします。

19-1 コンポーネント・オブジェクトの作成方法

C++/CLI でコンポーネント・オブジェクトの作成を行うには下記サンプルのように行います。ここではリソース・マネージャ・オブジェクトを作成する場面を例に説明します。

```
#include "stdafx.h"
.
.
using namespace Ivi::Visa::Interop;
.
.
IResourceManager3^ rm = gcnew ResourceManagerClass();
```

リソース・マネージャを作成するには ResourceManagerClass クラスに対して gcnew 演算子を使います。またプログラムの冒頭で using namespace 擬似命令が使われている点に注意して下さい。この using namespace 擬似命令は必須ではありませんが、もし書かない場合は、VISA COM タイプ・ライブラリが提供するネーム・スペース Ivi::Visa::Interop を全ての型名とシンボル名に明示的に書かなければなりません。

Notes:

同じ C++言語であっても、従来のアンマネージド版 C++(MFC や ATL を使う C++言語)とマネージド版の C++/CLI では、ポインタ演算子や new 演算子の扱い方が根本的に違います。C++/CLI では原則的にアスタリスク(*)を使ったポインタ表現は使わず、キャレット(^)を使います。キャレットはマネージドヒープへのポインタを指すものです。また、.NET アセンブリによるオブジェクトを作成する場合は、マネージドヒープへの割り当てを行う gcnew 演算子を使います。

19-2 インターフェースの参照

リソース・マネージャの Open メソッドは、C++/CLI スタイルの構文として、以下のような形式で宣言されます。

```
IVisaSession^ IResourceManager3::Open(
    System::String^ ResourceName,
    AccessMode mode,
    int initTimeout,
    System::String^ OptionString
);
```

メソッドの戻り値は IVisaSession^です。しかし実際のプログラム(アプリケーション、計測器ドライバなど)では多くの場合 IMessage^型の変数に戻り値を代入します。

```
.
.
IMessage^ msg =
    (IMessage^)rm->Open( "GPIB0::3::INSTR", AccessMode::NO_LOCK, 0, "");
.
.
msg->Close();
```

IResourceManager3::Open メソッドは IVisaSession インターフェースへの参照を返します。しかしそれを受取る変数が IMessage^型で宣言されているため、IMessage^型でタイプキャストを行います。C++/CLI では異なる COM インターフェースに対して暗黙のタイプキャストを行いません。しかし、明示的なタイプキャストを行うと、水面下では指定されたインターフェース型への問い合わせが行われます。"GPIB0::3::INSTR"のような INSTR リソースでオープンされた VISA セッションは IMessage インターフェースを必ずサポートしているため、結果的に IMessage インターフェースへの問い合わせが成功し、変数 *msg* にそのインターフェース参照が格納されます。

19-3 配列の扱い

C++/CLI での配列の扱いは.NET Framework で用意された System::Array の機能を使います。配列は System::Array 型で記述する方法と、型が判っている場合は array<>を用いて記述する方法があります。例えばリソース・マネージャの FindRsrc()メソッドは、有効な VISA アドレスを文字列配列で返します。

```
System::Array^ a = rm->FindRsrc( "?*INSTR");

array<String^>^ s =  rm->FindRsrc( "?*INSTR");

//どちらの記述も正しい
```

これ以外にも、複数の計測値を問い合わせるようなメソッドが double 配列を返すような場合も似たような方法で記述できます。

19-4 エラー・コードの原則

計測器を制御するアプリケーションの場合、実行時に計測器と正常に通信できたかどうかを確認するのは重要な事です。どんなに堅牢に設計・実装されたアプリケーションでも、実行時に計測器のパワーが落ちていたり通信ケーブルが外れていれば、計測器との通信は正常に行えません。

VISA では全てのメソッド呼び出しとプロパティへのアクセスに対して一貫したエラー・コード体系を持っています。VISA C API 及び VISA COM API の全てのファンクションは、その戻り値として 32 ビット・コードを使用します。

Notes:

VISA C API では、各関数の戻り値は全て ViStatus 型 (=int 32 ビット整数型)と定義されています。一方 VISA COM API では、COM の標準に合わせて全てのメソッド(及びプロパティ・アクセス・メソッド)の戻り値は HRESULT 型(=int 32 ビット整数型)と定義されています。VISA C API と VISA COM API の戻り値はコードのビット割り当てが若干異なりますが、同じ意味のステータス・コードは相互変換可能な関係にあります。本書は VISA COM API を中心とした説明をしていますので、HRESULT 型のステータス・コードを前提にして説明します。

C++/CLI では long/int 共に 32 ビット整数型でが、IDL で定義された COM インターフェース上の long 型は C++/CLI では int と扱われるようです。HRESULT は int 型と等価であると解釈して下さい。

一般に COM メソッド呼び出し時とプロパティ・アクセス時のステータス・コード(HRESULT)は次のように分類されます。VISA COM もその例外ではありません。

Table 19-1　　HRESULT の大まかな分類

値の範囲	**説明**
HRESULT=0	成功
HRESULT>0	警告
HRESULT<0	エラー

HRESULT がマイナスの場合は、.NET Framework は COM 例外を発生します。VISA COM API のメソッドを呼び出した結果、或いはプロパティにアクセスした結果として COM 例外が発生した場合、通

常は、計測器との通信が失敗したか、呼び出しのパラメータに問題があったか、或いは VISA の動作状態がその呼び出しを拒否した場合などが考えれます。

19-5 物理的な戻り値

C++/CLI では、COM インターフェースを通じてメソッドを呼び出したりプロパティにアクセスした場合、物理的な戻り値である HRESULT をプログラムから直接扱う事はできません。例えば、既に説明したリソース・マネージャの Open メソッドは、物理的には次のような形で、IDL 言語(COM インターフェースのタイプ・ライブラリ・ソース)で記述されています。C++/CLI ユーザが一般に考えているメソッドの宣言とはかなり違いますが、次に示す IDL 言語で書かれた COM メソッドの宣言と完全に等価なものです。

```
HRESULT Open(
    [in] BSTR ResourceName
    [in] AccessMode mode,
    [in] long initTimeout,
    [in] BSTR OptionString
    [out,retval] IVisaSession** vi
);
```

ここでは物理的な戻り値が HRESULT であり、C++/CLI では「戻り値」であると信じられていた IVisaSession への参照が(二重参照、ポインタへのポインタとして)最終パラメータになっています。ここで最終パラメータに[out,retval]という方向属性が与えられている事に注意して下さい。out は、このパラメータがリソース・マネージャ・オブジェクトから呼び出し元(アプリケーション又は計測器ドライバ側)に向かって返される事を意味します。そして retval はこの戻り値が COM メソッドとしての表面的な戻り値である事を意味します。C++/CLI での COM メソッドは、このような「表面的な」構文で扱われます。そして物理的な戻り値である HRESULT は水面下の処理にまわされるのです。

もうひとつ例を見てみましょう。

```
HRESULT get_Timeout(
    [out,retval] long* pVal
);

HRESULT put_Timeout(
    [in] long newVal
);
```

これは、IMessage インターフェースで使われる Timeout プロパティの構文を物理的に表現したものです。プロパティのとは結局のところ get_又は put_をプレフィックスに持つメソッドに過ぎません。get_メソッドだけが提供される場合、そのプロパティはリード・オンリーになります。逆に put_メソッドだけが提供される場合、そのプロパティはライト・オンリーとなります。ここでも物理的な戻り値は HRESULT になっており、C++/CLI からは水面下の処理になります。

このように、全ての COM メソッド(プロパティの get_メソッド、put_メソッドを含む)は必ず物理的な戻り値が HRESULT になっています。そして C++/CLI では、メソッドの戻り値がマイナスになった場合は COM 例外として処理されるように設計されています。具体的なエラー原因は Microsoft で一般的に定義されているものと、VISA COM のような特定コンポーネントで拡張定義しているものとがあります。

> Notes:
>
> VISA COM で定義されている HRESULT コードの詳細は、KI-VISA オンライン・ヘルプ又は本書の付録に掲載されています。

19-6 エラー・トラッピング

C++/CLI ではメソッドの呼び出し直後に HRESULT の値が水面下でチェックされ、マイナスの値になっている場合は COM 例外を発生します。COM 例外を捕獲するには try/catch ステートメントを使います。

```
try {
```

```
    msg =
        (IMessage^)rm->Open( "GPIB0::3::INSTR", AccessMode::NO_LOCK, 0, "");

    ...
    r = msg->WriteString( "CURR 20.0\n");
    r = msg->WriteString( "VOLT 5.5\n");
    ...
}
catch( System::Runtime::InteropServices::COMException^ ex) {
    MessageBox::Show(
    ex->Message, "error 0x" + Convert::ToString(ex-    >ErrorCode, 16));
}

finally {
    msg->Close();
}
```

try ブロック内で呼び出された全ての COM メソッド呼び出し及びプロパティのアクセスでは、.NET Framework によって HRESULT がチェックされます。HRESULT がマイナスの場合、.NET Framework は COM 例外を発生させ、catch ブロックに処理がジャンプします。

catch ブロックでは、System::Runtime::InteropServices::ComException 型の例外オブジェクト *ex* を通じてエラー(COM 例外)の原因を探る事ができます。

System::Runtime::InteropServices::ComException 型で定義されているもので特に有用な情報は、ErrorCode プロパティと Message プロパティです。それぞれ HRESULT コード及びエラー・メッセージ文字列を返します。

Notes:

try/catch ブロックを使わずにプログラムを実行すると、例外発生時にデバッガがそれを捕獲します。デバッガを使用せずにスタンド・アロンで実行された場合はプログラムがクラッシュする事になり、プログラムは強制終了します。最終的なアプリケーションでは必ずエラー・トラッピングを行い、プログラムが強制終了される事のないようにして下さい。

finally は必ずしも必要ありませんが、Close()メソッドの呼び出しのように、成功・失敗にかかわらず最終的には呼び出さなければならないような処理があれば、使うと便利です。

19-7 イベント・シンクの作成方法

サービス・リクエスト・イベントなどで COM のイベント・コールバック機能を使う場合、VISA COM ではイベント・シンクを作成しなければなりません。イベント・シンクとは、そのイベントが要求するコールバック・インターフェース(この場合 IEventHandler インターフェース)を装備する、任意のコンポーネント・オブジェクトです。そのためには IEventHandler インターフェースを実装するコンポーネント・クラスを作成しなければなりません。

イベント・シンクの作成

C++/CLI 統合環境のメニューバーから **Project | Add Class** を選択して下さい。**Add Class** ダイアログが表示されるので、**C++**から **C++ Class** を選択し **Add** します。すると **Generic C++ Class Wizard** が表示されるので、下記のようにして CMyEventSink クラスを作成します。ベースクラスに IEventHandler を指定し、インラインで生成しましょう。

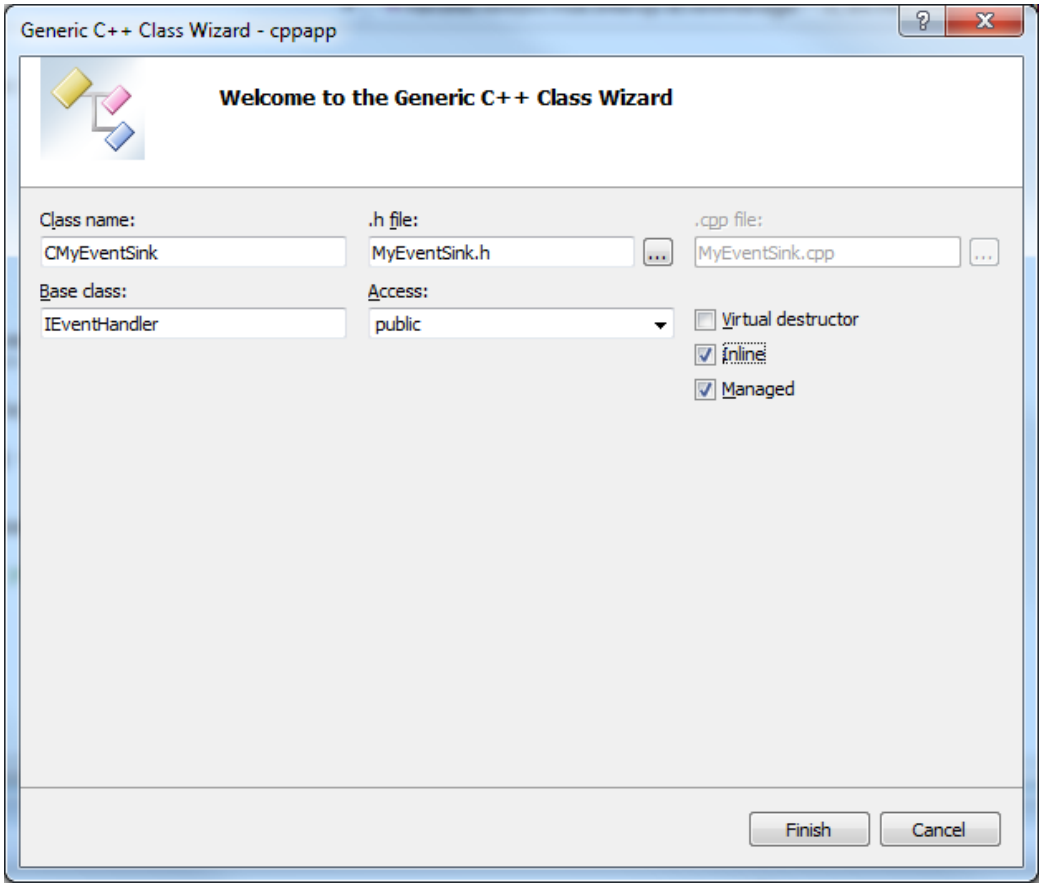

Figure 19-1 イベント・シンク用のクラスを作成する

すると、MyEventSink.h(インラインなので.cpp ファイルは無い)が生成されます。

そして下記のようにクラスを記述します。

```
#pragma once

using namespace Ivi::Visa::Interop;

ref class CMyEventSink : public IEventHandler
{
public:

    CMyEventSink(void)
    {
    }

    virtual void HandleEvent(
             IEventManager^ vi,
             IEvent^ ev,
             int userHandle)
    {
         IMessage^ msg = (IMessage^)vi;

         short stb;
         stb = msg->ReadSTB();

        // TODO: Add your own job depending on the stb value
    }

};
```

CMyEventSink クラスは IEventHandler インターフェースから派生したクラスとして作成します。IEventHandler には唯一のメソッドとして HandleEvent があります。このメソッドは VISA が用意するものではなく、作成するアプリケーション側で用意する責任があります。それはアプリケーションが呼び出すメソッドではなく、VISA がアプリケーションに向かって(逆方向に)呼び出すメソッドだからです。

19-8 ターゲット OS の設定

C++/CLI でのターゲット・プラットフォーム指定は少し複雑です。C++/CLI はマネージド・コードと P/Invoke によるアンマネージド・コードの混在を許しているからです。ここでは 2 箇所の設定が必要になります。

1 つは MSIL に関する設定です。Visual Studio 2008 のメニューバーから **Project➔ *<プロジェクト名>* Properties...**を選択し **General** タブの中にある **Common Language Runtime Support** という項目に注目します。ここでは MSIL の動作条件を細かく指定でき、**/clr**、**/clr:pure**、**/clr:safe** の選択肢があります。(/clr:oldSyntax は C++/CLI の構文をコンパイルできないので事実上選択不可です。) safe を選択するとアプリケーションから P/Invoke 機能(ネイティブ DLL、あるいは Win32 API を直接呼び出す機能)を利用できないので注意してください。

もうひとつは **Configuration Manager** によるターゲット・プラットフォームの設定です。**General** タブが表示されている状態で、**Configuration Manager** ボタンをクリックし、**Platform** の選択肢から新規に **x64** 又は **Win32** を作成して適用します。

ターゲットの指定と動作 OS との関係を下の表に示します。新規プロジェクトを作成した直後のデフォルト指定は/clr の Win32 ターゲットです。

Table 19-2　ターゲット・プラットフォームの指定

MSIL とターゲットの指定	**x86 OS での動作**	**x64 OS での動作**
/clr、又は/clr:pure Win32 ターゲット	32bit アプリとして動作	32bit アプリとして WOW64 環境で動作
/clr、又は/clr:pure x64 ターゲット	実行不可	64bit アプリとして動作
/clr:safe ターゲット不問	32bit アプリとして動作	64bit アプリとして動作

20- プログラミング言語依存の補足説明(Visual C++ 2008)

この章ではネイティブ版 Visual C++(MFC, ATL 等を使う場合など)固有の内容を説明します。このガイドブックで扱う Visual C++のバージョンは 2008 とします。マネージドバージョンの Visual C++については C++/CLI での説明を参照して下さい。

Visual C++では、COM メソッド及びプロパティを、その物理的な実装に忠実にプリミティブに扱う事が可能です。しかし、多くのアプリケーション・プログラムでは、より便利に使えるスマート・ポインタを使います。本書でもプリミティブな COM のアクセスではなくスマート・ポインタを使うことを前提に説明します。

20-1 COM ラッパ・モジュールの生成

「5-4 Visual C++ 2008(アンマネージド)での参照設定」でも説明しましたが、Visual C++では #import 擬似命令をプログラムの冒頭に書くことでタイプ・ライブラリを生成します。

```
#import "C:/Program Files (x86)/IVI Foundation/VISA/VisaCom/GlobMgr.DLL"
   no_namespace named_guids

(改行せず一行で書いて下さい。)
```

GlobMgr.DLL の実際のロケーションに合わせてパスの指定を変更して下さい。このファイルは通常<VXIPNPPATH>/VisaCom デイレクトリに置かれています。プロジェクトの stdafx.h ファイルに記述するのが良いでしょう。

#import 擬似命令は、DLL 又は TLB ファイルからタイプ・ライブラリを抽出し、拡張子.tlh と.tli の 2 個のラッパ・モジュールを C++ソース・ファイルとして生成します。これらのファイルには、メソッドやプロパティを簡単にアクセスするのに都合の良い、スマート・ポインタが定義されています。

例えば、IMessage 型は Visual C++プログラムの中では(水面下での処理が一切介在しない)生のインターフェース型です。従って、

```
IMessage* pMsg = NULL; //生のインターフェース・ポインタ
```

などように宣言するのですが、生のインターフェース型であるために、参照カウントを調整する AddRef、Release メソッドやインターフェースを問い合わせる QueryInterface の自動呼び出し(水面下での呼び出し)は一切行われません。ところがスマート・ポインタ IMessagePtr 型で COM インターフェースを宣言すると、これらの呼び出しを自動化する事ができます。スマート・ポインタとはこのような COM インターフェース固有の水面下での処理をサポートするものです。

```
IMessagePtr spMsg;   //スマート・ポインタはアスタリスクを付けずに宣言する
```

20-2 COM を使うための宣言

COM を使用する場合、COM オブジェクトを作成するプロセスは必ず CoInitialize または CoInitializeEx 関数(Win32 API)を呼び出さなければなりません。また終了時に CoUninitialize を呼び出さなければなりません。MFC アプリケーションでオートメーション・サポートを有効にして Wizard コードを生成させた場合は、*<CYourApp>*::InitInstance で AfxOleInit が呼び出されるようになっています。これは CoInitialize と同じ効果があります。そうでない場合は CoInitialize(Ex)/CoUninitialize 呼び出しを忘れずに追加して下さい。これを忘れるとリソース・マネージャ・オブジェクトを作成できません。簡単な動作実験をするためのコンソール・アプリケーションなどを書く場合は、忘れがちなので特に注意して下さい。

Notes:

OleInitialize と CoInitialize はどちらもアプリケーションを STA(Single Threaded Apartment)として初期化します。しかし VISA COM ライブラリはスレッディング・モデルを「both」として実装しているので、複数スレッド間で VISA セッションを使いまわす場合は、アプリケーションが MTA(Multi Threaded Apartment)として初期化されるように、下記のように呼び出して下さい。アプリケーションが MTA として動作する場合、VISA COM ライブラリも MTA として動作するので、この場合スレッド間で COM インターフェースを共有する場合でもスレッド･マーシャリングは必要ありません。

```
CoInitializeEx(NULL, COINIT_MULTITHREADED);
```

20-3 コンポーネント・オブジェクトの作成方法

Visual C++では、コンポーネント・オブジェクトの作成にはスマート・ポインタの CreateInstance()メソッドを使います。ここではリソース・マネージャ・オブジェクトを作成する場面を例に説明します。

```
HRESULT hr;
IResourceManager3Ptr spRM;
hr = spRM.CreateInstance( CLSID_ResourceManager);
```

ここでは、スマート・ポインタの CreateInstance()メソッドを使ってリソース・マネージャを作成しています。

20-4 インターフェースの参照

リソース・マネージャの Open メソッドは、例えば VB6 のような環境では、以下のような形式で宣言されているように見えます。C++で生のインターフェース・ポインタを使った場合は、「20-8 物理的な戻り値」で説明されるような宣言なのですが、スマート・ポインタを利用する事により、あたかも VB6 のような見かけ上の構文で利用する事ができます。

```
Function Open(
    ResourceName As String,
    [mode As AccessMode,]
    [initTimeout As Long,]
    [OptionString As String]
) As IVisaSession
```

メソッドの戻り値は IVisaSession です。しかし実際のプログラム(アプリケーション、計測器ドライバなど)では多くの場合 IMessage 型の変数に戻り値を代入します。

```
IMessagePtr spMsg;
spMsg = spRM->Open( L"GPIB0::3::INSTR", NO_LOCK, 0, L"");
.
.
.
spMsg->Close();
```

Open メソッドは一旦 IVisaSession インターフェースへの参照を返します。しかしそれをスマート・ポインタ IMessagePtr 型で宣言されている変数 *spMsg* で受取ります。スマート・ポインタ変数に代入を行う場合、IVisaSession インターフェース(Open で返されたインターフェース参照)の QueryInterface が水面下で呼び出されます。その際 IID_IMessage というインターフェース識別子(実際には GUID 値)を渡します。全ての COM インターフェースは間接的又は直接的に IUnknwon インターフェースから派生し、その結果 QueryInterface、AddRef、Release という 3 つのメソッドを装備しています。IVisaSession もその例外では有りません。

"GPIB0::3::INSTR"のような INSTR リソースでオープンされた VISA セッションは IMessage インターフェースを必ずサポートしているため、結果的に IID_IMessage を指定された QueryInterface 呼び出しは成功し、最終的に IMessage インターフェースへの参照が返されて、変数 *spMsg* に格納されるのです。これはスマート・ポインタの代入演算子(=)がオーバーロードされており、QueryInterface を水面下で呼び出すように仕組まれているからなのです。

Notes:

IUnknown には 3 つのメソッドが定義されています。QueryInterface はインターフェースの問合せをします。AddRef は参照カウンタの増加、Release メソッドは参照カウンタの現象を行います。全ての COM オブジェクトは最低でも IUnknown から派生したインターフェースを装備しているので、これら全てのメソッドを必ず装備しているといえます。これら 3 つのメソッドを全てを装備していない限り、COM オブジェクトとは言えません。

COM オブジェクトの寿命は参照カウンタによって管理されます。参照カウンタは、通常 COM オブジェクト自身によって管理され、AddRef で増加、Release で減少します。オブジェクトが QueryInterface 呼び出しを受けそれが成功すると、オブジェクトは AddRef を連鎖的に 1 回だけ呼び出します。オブジェクトが Release 呼び出しを受け、その結果参照カウントがゼロのなった時、オブジェクトは自分自身を破壊します。

Visual C++で#import 擬似命令を使用した場合に生成される COM ラッパ(スマート・ポインタ)では、代入演算子(=)が QueryInterface にマップされています。代入演算子はまた、既に保持している有効なインターフェース・ポインタがあれば QueryInterface を呼び出す寸前に Release を呼び出します。Release 呼び出しは、スマート・ポインタのデストラクタからも自動的に呼び出されます。

20-5 文字列の扱い

C++環境では、COM インターフェースをアクセスする際に扱われる文字列は全て UNICODE になります。従ってリテラル(直値文字列)を記述する場合は、L"*IDN?"のように引用符の手前に L が付きます。これは、アプリケーションが UNICODE ビルドであっても ANSI ビルドであっても変わる事はありません。

Notes:

COM で文字列パラメータや戻り値を受け渡す場合、UNICODE 文字の配列ではなく BSTR 型を使います。BSTR 型は見かけ上は wchar_t*と typedef されていますが、実際には SysAllocString や SysFreeString などの API を使って生成・廃棄を行わなければなりません。

Visual C++では BSTR 型をラップする_bstr_t クラスがあります。これは BSTR のメモリ管理を水面下で処理してくれるだけでなく、UNICODE 文字列と ANSI 文字列の相互変換を行えるように、オーバーロードされた代入演算子やタイプ・キャスト演算子を実装しています。

Visual C++ 2008 では、通常新規作成したプロジェクトは UNICODE ビルドが指定されています。アプリケーションを Windows95/98/Me で動作させる需要が殆ど無くなった現在、ANSI ビルドを選択する必要性は殆どないでしょう。

20-6 配列の扱い

C++環境では COM 互換の配列を扱うには少々テクニックが必要です。COM インターフェースがメソッドの引数や戻り値あるいはプロパティで使用する配列は SAFEARRAY と呼ばれるものであり、C/C++言語のそれとは大きく異なるからです。リソース・マネージャの FindRsrc メソッドが返す文字列配列(SAFEARRAY)を例に説明しましょう。

```
// Search for valid visa resource strings
//
.
.
SAFEARRAY* pSA = NULL;
try {
    pSA = spRM->FindRsrc( L"?*INSTR");
}
catch(...) {
}

if( pSA) {
```

```
        BSTR*  rgElems = NULL;
        ::SafeArrayAccessData( pSA, (PVOID*)&rgElems);
        ASSERT( rgElems);

        LONG lLBound, lUBound;
        ::SafeArrayGetLBound( pSA, 1, &lLBound);
        ::SafeArrayGetUBound( pSA, 1, &lUBound);

        for( long lNdx=lLBound; lNdx<=lUBound; lNdx++) {
             _bstr_t strFound = rgElems[lNdx];

             //この時点で strFound は文字列を 1 個保持している

        }

        ::SafeArrayUnaccessData( pSA);
        ::SafeArrayDestroyData( pSA);
   }
```

FindRsrc メソッドから文字列配列を受取るには、一旦 SAFEARRAY 型ポインタ変数 pSA を使います。更に SafeArrayAccessData、SafeArrayGetLBound、SafeArrayGetUBound、SafeArrayUnaccessData、SafeArrayDestroyData の各 API 関数を駆使して内部エレメントにアクセスします。

20-7 エラー・コードの原則

計測器を制御するアプリケーションの場合、実行時に計測器と正常に通信できたかどうかを確認するのは重要な事です。どんなに堅牢に設計・実装されたアプリケーションでも、実行時に計測器のパワーが落ちていたり GPIB ケーブルが外れていれば、計測器との通信は正常に行えません。

VISA では全てのメソッド呼び出しとプロパティへのアクセスに対して一貫したエラー・コード体系を持っています。VISA C API 及び VISA COM API の全てのファンクションは、その戻り値として 32 ビット・コードを使用します。

Notes:

VISA C API では、各関数の戻り値は全て ViStatus 型 (=Long 型)と定義されています。一方 VISA COM API では、COM の標準に合わせて全てのメソッド(及びプロパティ・アクセス・メソッド)の戻り値は HRESULT 型 (=Long 型)と定義されています。VISA C API と VISA COM API の戻り値はコードのビット割り当てが若干異なりますが、同じ意味のステータス・コードは相互変換可能な関係にあります。本書は VISA COM API を中心とした説明をしていますので、HRESULT 型のステータス・コードを前提にして説明します。

一般に COM メソッド呼び出し時とプロパティ・アクセス時のステータス・コード(HRESULT)は次のように分類されます。VISA COM もその例外ではありません。

Table 20-1　　HRESULT の大まかな分類

値の範囲	**説明**
HRESULT=0	成功
HRESULT>0	警告
HRESULT<0	エラー

HRESULT がマイナスの場合は、スマート・ポインタのラッパー・コードは COM 例外を発生します。VISA COM API のメソッドを呼び出した結果、或いはプロパティにアクセスした結果として COM 例外が発生した場合、通常は、計測器との通信が失敗したか、呼び出しのパラメータに問題があったか、或いは VISA の動作状態がその呼び出しを拒否した場合などが考えれます。

20-8 物理的な戻り値

C++スマートポインターでは、COM インターフェースを通じてメソッドを呼び出したりプロパティにアクセスした場合、物理的な戻り値である HRESULT をプログラムから直接扱う事はできません。例えば、既に説明したリソース・マネージャの Open メソッドは、物理的には次のように宣言されています。スマート・ポインタを通じてアクセスする場合の見かけ上の宣言とはかなり違いますが、次に示す IDL 言語で書かれた COM メソッドの宣言と完全に等価なものです。

```
HRESULT Open(
    [in] BSTR ResourceName
    [in] AccessMode mode,
    [in] long initTimeout,
    [in] BSTR OptionString
    [out,retval] IVisaSession** vi
);
```

ここでは物理的な戻り値が HRESULT であり、スマート・ポインタを使用した場合には「戻り値」であると信じられていた IVisaSession への参照が(二重参照、ポインタへのポインタとして)最終パラメータになっています。ここで最終パラメータに[out,retval]という方向属性が与えられている事に注意して下さい。out は、このパラメータがリソース・マネージャ・オブジェクトから呼び出し元(アプリケーション又は計測器ドライバ側)に向かって返される事を意味します。そして retval はこの戻り値が COM メソッドとしての表面的な戻り値である事を意味します。C++でスマート・ポインタを使用した場合の COM メソッドは、このような「表面的な」構文で扱われます。そして物理的な戻り値である HRESULT は水面下の処理にまわされるのです。

もうひとつ例を見てみましょう。

```
HRESULT get_Timeout(
    [out,retval] long* pVal
);

HRESULT put_Timeout(
    [in] long newVal
);
```

これは、IMessage インターフェースで使われる Timeout プロパティの構文を物理的に表現したものです。プロパティのとは結局のところ get_又は put_をプレフィックスに持つメソッドに過ぎません。get_メソッドだけが提供される場合、そのプロパティはリード・オンリーになります。逆に put_メソッドだけが提供される場合、そのプロパティはライト・オンリーとなります。ここでも物理的な戻り値は HRESULT になっており、スマート・ポインタを利用した場合は水面下の処理になります。

このように、全ての COM メソッド(プロパティの get_メソッド、put_メソッドを含む)は必ず物理的な戻り値が HRESULT になっています。そしてスマート・ポインタのラッパ・コードでは、メソッドの戻り値がマイナスになった場合はランタイム・エラーとして処理されるように設計されています。具体的なエラー原因は Microsoft で一般的に定義されているものと、VISA COM のような特定コンポーネントで拡張定義しているものとがあります。

> Notes:
>
> VISA COM で定義されている HRESULT コードの詳細は、KI-VISA オンライン・ヘルプ又は本書の付録に掲載されています。

20-9 エラー・トラッピング

C++でスマート・ポインタを利用する場合はメソッドの呼び出し直後に HRESULT の値が水面下でチェックされ、マイナスの値になっている場合は COM 例外が発生します。例外を捕獲するには C++の try/catch ブロックを使います。

```
try {
    r = spmsg->WriteString( L"CURR 20.0\n");
    r = spmsg->WriteString( L"VOLT 5.5\n");
    .
    .
    .
```

```
}
catch( _com_error e) {
   //エラー・コードとメッセージは_com_error クラスのメンバー関数から取得できる
   HRESULT hr = e.Error();
   _bstr_t strDescription = e.Description();

   //TODO: ...
}
```

Notes:

try/catch ブロックを使わずにプログラムを実行し COM 例外が発生すると、プログラムは「処理されない例外が発生した」として強制終了させられます。最終的なアプリケーションでは必ずエラー・トラッピングを行い、プログラムが強制終了される事のないようにして下さい。

20-10 イベント・シンクの作成方法

サービス・リクエスト・イベントなどでイベント・コールバック機能を使う場合、VISA COM ではイベント・シンクを作成しなければなりません。イベント・シンクとは、そのイベントが要求するコールバック・インターフェース(この場合 IEventHandler インターフェース)を装備する、任意のコンポーネント・オブジェクトです。そのためには IEventHandler インターフェースを実装するコンポーネント・クラスを作成しなければなりません。

イベント・シンクの作成

イベント・シンクを作成するには、C++クラスを 1 個作成する必要があります。その C++クラスで IEventHandler インターフェースを実装することになります。C++環境で C++クラス・モジュールを生成する方法はプロジェクトのタイプによって異なりますが、ここでは手作業で MyEventSink.H ファイルを作成し、すべてインライン記述した場合を想定します。作成するのは CMyEventSink クラスとしましょう。

ここで作成するイベントシンクは、これも立派な COM クラスです(DLL 等に独立に配置しているわけではありませんが)。従って、COM クラスとしての実装の原則に従わなければなりません。

CMyEventSink のベースクラスになっている IEventHandler は、元々IUnknown を継承した派生インターフェースです。全ての COM クラスの原点になる IUnknown には、元々QueryInterface, AddRef, Release の 3 つのメソッド(仮想メンバー関数)が設計されています。そしてそこから派生した IEventHandler には、追加メソッドとして HandleEvent が設計されています。これら合計 4 つのメソッドは全て純仮想関数(pure virtual method)として記述されているため、そこからの最終的な派生クラス CMyEventSink は、これらを全て現実に実装する必要があります。(そうしないとビルドが通りません。)

IUnknown からの 3 つのメソッドの実装は全ての COM クラスに共通の部分なのでここでは詳細は省略します。HandleEvent メソッドは、raw_HandleEvent 関数に既にリネームされている点に注意して下さい。stdafx.h に記述した#import 文では、GlobMgr.dll のタイプ・ライブラリから.tlh/.tli を自動生成しました。しかしこの時にスマートポインタを使用するための諸条件を与えているため、(IUnknown の 3 つのメソッドを除いた残りの)元々のメソッド名はスマートポインターでの使用にあてがわれているため、生のインターフェース記述として raw_として雛型を生成しています。ここでのイベントシンクの実装は生の COM インターフェースを直接扱う必要があるため、raw_HandleEvent 関数を実装することになります。

```
#include "stdafx.h"

class CMyEventSink : public IEventHandler
{
protected:
   DWORD m_dwRef;
public:
    CMyEventSink()    {m_dwRef = 0;}
    ~CMyEventSink()   {}
```



```
public:
// Implementation

    // IUnknown
     HRESULT __stdcall QueryInterface( REFIID riid, void** ppvObject)
    {
        if( (riid == IID_IEventHandler) || (riid == IID_IUnknown))      {
            *ppvObject = (IUnknown*)this;
            AddRef();
            return S_OK;
        }
        return E_NOINTERFACE;
    }
     ULONG __stdcall AddRef()
    {
        return ::InterlockedIncrement( (LONG*)&m_dwRef);
    }
     ULONG __stdcall Release()
    {
        ::InterlockedDecrement( (LONG*)&m_dwRef);
        if( m_dwRef == 0){
            delete this;
            return 0;
        }
        return m_dwRef;
    }




    // IEventHandler
     HRESULT __stdcall raw_HandleEvent(
        IEventManager* pVi,
        IEvent* pEvent,
        long userHandle
       )
     {

        IMessagePtr spMsg = pVi;
        short stb = spMsg->ReadSTB();

        return S_OK;
     }
};
```

20-11 ターゲット OS の設定

アンマネージド版 C++でのターゲット・プラットフォーム指定は次のようにします。

Visual Studio 2008 のメニューバーから **Project➔ *<プロジェクト名>* Properties...**を選択し **General** タブを選択します。さらに、**Configuration Manager** ボタンをクリックし、**Platform** の選択肢から新規に **x64** 又は **Win32** を作成して適用します。(デフォルトで Win32 は作られています。)

ターゲットの指定と動作 OS との関係を下の表に示します。新規プロジェクトを作成した直後のデフォルト指定は Win33 ターゲットです。

Table 20-2　ターゲット・プラットフォームの指定

ターゲットの指定	**x86 OS での動作**	**x64 OS での動作**
Win32 ターゲット	32bit アプリとして動作	32bit アプリとして WOW64 環境で動作
x64 ターゲット	実行不可	64bit アプリとして動作

21- プログラミング言語依存の補足説明(C++Builder XE2)

この章では C++Builder 固有の内容を説明します。このガイドブックで扱う C++Builder のバージョンは XE2 とします。

C++Builder では、COM メソッド及びプロパティを、その物理的な実装に忠実にプリミティブに扱う事が可能です。しかし、多くのアプリケーション・プログラムでは、より便利に使えるスマート・ポインタを使います。本書でもプリミティブな COM のアクセスではなくスマート・ポインタを使うことを前提に説明します。

21-1 COM ラッパ・モジュールの生成

C++Builder では COM ラッパ・モジュールの生成を外部コマンドライン・ツール、TLIBIMP.exe に頼っています(Visual C++に似た#import は推奨されていません)。

コマンド・プロンプトを起動し、アプリケーションのソースが置かれた作業ディレクトリ内で下記コマンドを実行して下さい。この操作で生成される VisaComLib_TLB.cpp 及び同.h を使います。

```
tlibimp -C "C:/Program Files (x86)/IVI Foundation/VISA/VisaCom/GlobMgr.DLL"
```

プロジェクトに VisaComLib_TLB.cpp を追加して下さい。更にアプリケーションのソース・コード上に、下記インクルード文を追加して下さい。

```
#include <ComObj.hpp>
#include "VisaComLib_TLB.h"
```

TLIBIMP.exe が生成したファイルには、メソッドやプロパティを簡単にアクセスするのに都合の良い、スマート・ポインタが定義されています。

例えば、IMessage 型は C++Builder プログラムの中では(水面下での処理が一切介在しない)生のインターフェース型です。従って、

```
IMessage* pMsg = NULL; //生のインターフェース・ポインタ
```

などように宣言するのですが、生のインターフェース型であるために、参照カウントを調整する AddRef、Release メソッドやインターフェースを問い合わせる QueryInterface の自動呼び出し(水面下での呼び出し)は一切行われません。ところがスマート・ポインタ IMessagePtr 型で COM インターフェースを宣言すると、これらの呼び出しを自動化する事ができます。スマート・ポインタとはこのような COM インターフェース固有の水面下での処理をサポートするものです。

```
IMessagePtr spMsg;    //スマート・ポインタはアスタリスクを付けずに宣言する
```

21-2 COM を使うための宣言

COM を使用する場合、COM オブジェクトを作成するプロセスは必ず CoInitialize または CoInitializeEx 関数(Win32 API)を呼び出さなければなりません。また終了時に CoUninitialize を呼び出さなければなりません。VCL アプリケーションでオートメーションが既にサポートされている場合は、同様の処理がフレームワーク内で行われている場合があります。そうでない場合は CoInitialize(Ex)/CoUninitialize 呼び出しを忘れずに追加して下さい。

これを忘れるとリソース・マネージャ・オブジェクトを作成できません。簡単な動作実験をするためのコンソール・アプリケーションなどを書く場合は、忘れがちなので特に注意して下さい。

21-3 コンポーネント・オブジェクトの作成方法

C++Builder では、コンポーネント・オブジェクトの作成にはスマート・ポインタの CreateInstance()メソッドを使います。ここではリソース・マネージャ・オブジェクトを作成する場面を例に説明します。

```
HRESULT hr;
IResourceManager3Ptr spRM;
hr = spRM.CreateInstance( CLSID_ResourceManager);
```

ここでは、スマート・ポインタの CreateInstance()メソッドを使ってリソース・マネージャを作成しています。

21-4 インターフェースの参照

リソース・マネージャの Open メソッドは、例えば VB6 のような環境では、以下のような形式で宣言されているように見えます。

```
Function Open(
    ResourceName As String,
    [mode As AccessMode,]
    [initTimeout As Long,]
    [OptionString As String]
) As IVisaSession
```

しかし C++Builder でスマート・ポインタを利用する場合は、生の COM インターフェースが剥き出しになります (Visual C++のスマート・ポインタとは事情が異なる)。生の COM インターフェースとは次に示す IDL 言語で書かれた物です。

```
HRESULT Open(
    [in] BSTR ResourceName
    [in] AccessMode mode,
    [in] long initTimeout,
    [in] BSTR OptionString
    [out,retval] IVisaSession** vi
);
```

BSTR は C++Builder 上では wchar_t* (UNICODE 文字列へのポインタ)として見えます。メソッドの戻り値は常に HRESULT であり、Open メソッドで取得したい VISA セッションへの参照(COM インターフェース・ポインタ)は戻り値ではなく最終パラメータでの二重参照を通じて行われます。

```
IVisaSessionPtr spVi;
IMessagePtr spMsg;

hr = spRM->Open(
    L"TCPIP::192.168.1.3::INSTR", AccessMode::NO_LOCK, 0, L"", &spVi);

hr = spVi->QueryInterface(IID_IMessage, (void**)&spMsg);
hr = spMsg->Close();
```

リソース・マネージャ Open メソッドは、最終パラメータ(IVisaSession ポインタの参照渡し)を通じて一旦 IVisaSession インターフェースへの参照を返します。しかし IVisaSession インターフ

ェースはメッセージ・ベースの I/O を直接行えないので、この例では IMessage インターフェースを必要とします。

Open メソッドの最終パラメータはシンタックス上 IVisaSession**型なので、IMessage**型でそれを直接受け取ることは出来ません。その為一旦 IVisaSessionPtr 型の spVi でそれを受け、そのあと明示的に QueryInterface()で IMessage インターフェースを照会します。

Notes:

C++Builder のスマート・ポインタではプロパティへのアクセスは変数のように扱うことは出来ず、全てアクセッサ・メソッド(get_又は set_の付いた関数)で扱います。同様に、値を返すメソッドも、その受け渡しを最終パラメータでの参照渡しを通じて行います。これは全てのメソッドが物理的に HRESULT を返すように作られており、C++Builder ではその物理的なシンタックスが剥き出しになっているからです。Visual C++のスマート・ポインタとはこの点が異なります。

IUnknown には 3 つのメソッドが定義されています。QueryInterface はインターフェースの問合せをします。AddRef は参照カウンタの増加、Release メソッドは参照カウンタの現象を行います。全ての COM オブジェクトは最低でも IUnknown から派生したインターフェースを装備しているので、これら全てのメソッドを必ず装備しているといえます。これら 3 つのメソッドを全てを装備していない限り、COM オブジェクトとは言えません。

COM オブジェクトの寿命は参照カウンタによって管理されます。参照カウンタは、通常 COM オブジェクト自身によって管理され、AddRef で増加、Release で減少します。オブジェクトが QueryInterface 呼び出しを受けそれが成功すると、オブジェクトは AddRef を連鎖的に 1 回だけ呼び出します。オブジェクトが Release 呼び出しを受け、その結果参照カウントがゼロのなった時、オブジェクトは自分自身を破壊します。C++Builder のスマート。ポインタは Visual C++よりは薄いラッパですが、参照カウントの管理に関しては自動化します。

21-5 文字列の扱い

C++環境では、COM インターフェースをアクセスする際に扱われる文字列は全て UNICODE になります。従ってリテラル(直値文字列)を記述する場合は、L"*IDN?"のように引用符の手前に L が付きます。これは、アプリケーションが UNICODE ビルドであっても ANSI ビルドであっても変わる事はありません。

Notes:

COM で文字列パラメータや戻り値を受け渡す場合、UNICODE 文字の配列ではなく BSTR 型を使います。BSTR 型は見かけ上は wchar_t*と typedef されていますが、実際には SysAllocString や SysFreeString などの API を使って生成・廃棄を行わなければなりません。

21-6 配列の扱い

C++Builder 環境では COM 互換の配列を扱うには少々テクニックが必要です。COM インターフェースがメソッドの引数や戻り値あるいはプロパティで使用する配列は SAFEARRAY と呼ばれるものであり、C/C++言語のそれとは大きく異なるからです。リソース・マネージャの FindRsrc メソッドが返す文字列配列(SAFEARRAY)を例に説明しましょう。

```
// Search for valid visa resource strings
//
.
.
SAFEARRAY* pSA = NULL;
if( pSA) {   try {
    hr = spRM->FindRsrc( L"?*INSTR", &pSA);
}
catch(...) {
}

if( pSA) {
    BSTR*  rgElems = NULL;
    ::SafeArrayAccessData( pSA, (PVOID*)&rgElems);
```

```
        LONG lLBound, lUBound;
        ::SafeArrayGetLBound( pSA, 1, &lLBound);
        ::SafeArrayGetUBound( pSA, 1, &lUBound);

        for( long lNdx=lLBound; lNdx<=lUBound; lNdx++) {
             String strFound = rgElems[lNdx];

             //この時点で strFound は文字列を 1 個保持している

        }

        ::SafeArrayUnaccessData( pSA);
        ::SafeArrayDestroyData( pSA);
    }
```

FindRsrc メソッドから文字列配列を受取るには、一旦 SAFEARRAY 型ポインタ変数 pSA を使います。更に SafeArrayAccessData、SafeArrayGetLBound、SafeArrayGetUBound、SafeArrayUnaccessData、SafeArrayDestroyData の各 API 関数を駆使して内部エレメントにアクセスします。

21-7 エラー・コードの原則

計測器を制御するアプリケーションの場合、実行時に計測器と正常に通信できたかどうかを確認するのは重要な事です。どんなに堅牢に設計・実装されたアプリケーションでも、実行時に計測器のパワーが落ちていたり GPIB ケーブルが外れていれば、計測器との通信は正常に行えません。

VISA では全てのメソッド呼び出しとプロパティへのアクセスに対して一貫したエラー・コード体系を持っています。VISA C API 及び VISA COM API の全てのファンクションは、その戻り値として 32 ビット・コードを使用します。

Notes:

VISA C API では、各関数の戻り値は全て ViStatus 型 (=Long 型)と定義されています。一方 VISA COM API では、COM の標準に合わせて全てのメソッド(及びプロパティ・アクセス・メソッド)の戻り値は HRESULT 型 (=Long 型)と定義されています。VISA C API と VISA COM API の戻り値はコードのビット割り当てが若干異なりますが、同じ意味のステータス・コードは相互変換可能な関係にあります。本書は VISA COM API を中心とした説明をしていますので、HRESULT 型のステータス・コードを前提にして説明します。

一般に COM メソッド呼び出し時とプロパティ・アクセス時のステータス・コード(HRESULT)は次のように分類されます。VISA COM もその例外ではありません。

Table 21-1　　HRESULT の大まかな分類

値の範囲	**説明**
HRESULT=0	成功
HRESULT>0	警告
HRESULT<0	エラー

HRESULT がマイナスの場合は、通常は、計測器との通信が失敗したか、呼び出しのパラメータに問題があったか、或いは VISA の動作状態がその呼び出しを拒否した場合などが考えれます。

Notes:

Visual C++のスマート・ポインタと異なり、C++Builder では HRESULT がマイナスであっても C++言語としての COM 例外は投げられません。各々のメソッド呼び出しのあと、HRESULT の値を IF 文等で評価する必要があります。

Notes:

VISA COM で定義されている HRESULT コードの詳細は、KI-VISA オンライン・ヘルプ又は本書の付録に掲載されています。

21-8 エラー・トラッピング

C++Builder のスマート・ポインタを利用する場合はメソッドの呼び出し直後に、必要に応じて HRESULT の値を評価する必要があります。HRESULT がマイナスの場合はエラーを意味します。下記の例では、HRESULT がマイナスだった場合に、EOleSysError 型の例外オブジェクトを自力で投げるように記述しています。try ブロックから投げた例外を捕獲するには catch ブロックを使います。

```
try {
    .
   long lenWritten;
   hr = spMsg->WriteString( L"CURR 20.0\n", &lenWritten);
   if( FAILED(hr))
       throw( EOleSysError(NULL, hr, 0));
   hr = spMsg->WriteString( L"VOLT 5.5\n", &lenWritten);
   if( FAILED(hr))
       throw( EOleSysError(NULL, hr, 0));    .
    .
    .
}
catch( EOleSysError& e) {
   wchar_t msg[256];
   wsprintf( msg, L"Error = 0x%08x", e.ErrorCode);
   //TODO: ...
}
```

21-9 イベント・シンクの作成方法

サービス・リクエスト・イベントなどでイベント・コールバック機能を使う場合、VISA COM ではイベント・シンクを作成しなければなりません。イベント・シンクとは、そのイベントが要求するコールバック・インターフェース(この場合 IEventHandler インターフェース)を装備する、任意のコンポーネント・オブジェクトです。そのためには IEventHandler インターフェースを実装するコンポーネント・クラスを作成しなければなりません。

イベント・シンクの作成

イベント・シンクを作成するには、C++クラスを 1 個作成する必要があります。その C++クラスで IEventHandler インターフェースを実装することになります。C++環境で C++クラス・モジュールを生成する方法はプロジェクトのタイプによって異なりますが、ここでは手作業で MyEventSink.H ファイルを作成し、すべてインライン記述した場合を想定します。作成するのは CMyEventSink クラスとしましょう。

ここで作成するイベントシンクは、これも立派な COM クラスです(DLL 等に独立に配置しているわけではありませんが)。従って、COM クラスとしての実装の原則に従わなければなりません。

TMyEventSink のベースクラスになっている IEventHandler は、元々IUnknown を継承した派生インターフェースです。全ての COM クラスの原点になる IUnknown には、元々QueryInterface, AddRef, Release の 3 つのメソッド(仮想メンバー関数)が設計されています。そしてそこから派生した IEventHandler には、追加メソッドとして HandleEvent が設計されています。これら合計 4 つのメソッドは全て純仮想関数(pure virtual method)として記述されているため、そこからの最終的な派生クラス TMyEventSink は、これらを全て現実に実装する必要があります。(そうしないとビルドが通りません。)

幸い C++Builder には IUnknown の 3 つのメソッドを実装済みのデリゲート・クラス TInterfacedObject 型が用意されています。

```
#include <ComObj.hpp>
#include "VisaComLib_TLB.h"

class TMyEventSink : public TInterfacedObject, IEventHandler
```

```
{
public:
// Implementation

    // IUnknown
    // (simply delegates to TInterfacedObject)
    HRESULT __stdcall QueryInterface( REFIID riid, void** ppv)
    {
        return TInterfacedObject::QueryInterface(riid, (void*)ppv);
    }
    ULONG __stdcall AddRef()
    {
         return TInterfacedObject::_AddRef();
    }
    ULONG __stdcall Release()
    {
     return TInterfacedObject::_Release();
    }

    // IEventHandler
    HRESULT __stdcall HandleEvent(
       IEventManager* pVi,
        IEvent* pEvent,
        long userHandle
      )
    {

        IMessagePtr spMsg = pVi;
        short stb;
        HRESULT hr = spMsg->ReadSTB( &stb);

        return S_OK;
    }
};
```

22- プログラミング言語依存の補足説明(Delphi XE2)

この章では Delphi 固有の内容を説明します。このガイドブックで扱う Delphi のバージョンは XE2 とします。

Delphi では、COM メソッド及びプロパティを、その物理的な実装に忠実にプリミティブに扱う事が可能です。

22-1 COM ラッパ・モジュールの生成

Delphi では COM ラッパ・モジュールの生成を外部コマンドライン・ツール、TLIBIMP.exe に頼っています。コマンド・プロンプトを起動し、アプリケーションのソースが置かれた作業ディレクトリ内で下記コマンドを実行して下さい。この操作で生成される VisaComLib_TLB.pas を使います。

```
tlibimp -P "C:/Program Files (x86)/IVI Foundation/VISA/VisaCom/GlobMgr.DLL"
```

プロジェクトに VisaComLib_TLB.pas を追加して下さいすると、アプリケーションのソース・コード上の uses ブロックに VisaComLib_TLB.pas が自動追加されます。更に uses ブロックに、下記の Pascal ユニットを追加して下さい。

```
ComObj, Windows,
```

TLIBIMP.exe が生成したファイルには、メソッドやプロパティを簡単にアクセスするのに都合の良いラッパ・クラスが定義されています。例えば、下記は IMessage 型のラッパ・クラスで変数を宣言する一例です。

```
var
   msg : IMessage;
...
```

I で始まる型名は、C/C++言語では生のインターフェース型のため、参照カウントを調整する AddRef、Release メソッドやインターフェースを問い合わせる QueryInterface の自動呼び出し(水面下での呼び出し)は一切行われませんでした。しかし Delphi での型は、生のインターフェース・ポインタではなくラッパ・クラスです。その為これら参照カウントに関する処理は自動化されています。また C/C++のようにポインタ型で宣言するのではなく、実体型のように変数宣言します。

22-2 コンポーネント・オブジェクトの作成方法

Delphi では、コンポーネント・オブジェクトの作成には COM ラッパ・クラスの Create()メソッドを使います。ここではリソース・マネージャ・オブジェクトを作成する場面を例に説明します。

```
var
  rm : IResourceManager3;

...
  rm:= CoResourceManager.Create();
```

ここでは、ラッパ・クラスの Create()メソッドを使ってリソース・マネージャを作成しています。

22-3 インターフェースの参照

リソース・マネージャの Open メソッドは、例えば VB6 のような環境では、以下のような形式で宣言されているように見えます。

```
Function Open(
   ResourceName As String,
   [mode As AccessMode,]
   [initTimeout As Long,]
   [OptionString As String]
) As IVisaSession
```

しかし Delphi でラッパ・クラスを利用する場合は、生の COM インターフェースが剥き出しになります。生の COM インターフェースとは次に示す IDL 言語で書かれた物です。

```
HRESULT Open(
   [in] BSTR ResourceName
   [in] AccessMode mode,
   [in] long initTimeout,
   [in] BSTR OptionString
   [out,retval] IVisaSession** vi
);
```

BSTR は Delphi 上では WideString として、AccessMode は列挙型として、それぞれ見えます。メソッドの戻り値は常に HRESULT であり、Open メソッドで取得したい VISA セッションへの参照(COM インターフェース・ポインタ)は戻り値ではなく最終パラメータでの参照を通じて行われます。

```
var
  hr : HRESULT;
  rm : IResourceManager3;
  vi : IVisaSession;
  msg : IMessage;
    ...
begin
    ...
    hr:= rm.Open( 'TCPIP::192.168.1.3::INSTR', NO_LOCK, 0, '', vi);
    hr:= vi.QueryInterface(IID_IMessage, msg);

    hr:= msg.Close();
    ...
end.
```

リソース・マネージャ Open メソッドは、最終パラメータ(IVisaSession の参照渡し)を通じて一旦 IVisaSession インターフェースへの参照を返します。しかし IVisaSession インターフェースはメッセージ・ベースの I/O を直接行えないので、この例では IMessage インターフェースを必要とします。

Open メソッドの最終パラメータはシンタックス上 IVisaSession への参照なので、IMessage 型でそれを直接受け取ることは出来ません。その為一旦 IVisaSession 型の vi でそれを受け、そのあと明示的に QueryInterface()で IMessage インターフェースを照会します。

Notes:

Delphi の COM ラッパ・クラスではプロパティへのアクセスは変数のように扱うことは出来ず、全てアクセッサ・メソッド(get_又は set_の付いた関数)で扱います。同様に、値を返すメソッドも、その受け渡しを最終パラメータでの参照渡しを通じて行います。これは全てのメソッドが物理的に HRESULT を返すように作られており、Delphi ではその物理的なシンタックスが剥き出しになっているからです。

IUnknown には 3 つのメソッドが定義されています。QueryInterface はインターフェースの問合せをします。AddRef は参照カウンタの増加、Release メソッドは参照カウンタの現象を行います。全ての COM オブジ

ェクトは最低でも IUnknown から派生したインターフェースを装備しているので、これら全てのメソッドを必ず装備しているといえます。これら 3 つのメソッドを全てを装備していない限り、COM オブジェクトとは言えません。

COM オブジェクトの寿命は参照カウンタによって管理されます。参照カウンタは、通常 COM オブジェクト自身によって管理され、AddRef で増加、Release で減少します。オブジェクトが QueryInterface 呼び出しを受けそれが成功すると、オブジェクトは AddRef を連鎖的に 1 回だけ呼び出します。オブジェクトが Release 呼び出しを受け、その結果参照カウントがゼロのなった時、オブジェクトは自分自身を破壊します。Delphi の COM ラッパ・クラスは C++Builder の場合と同様薄いラッパですが、参照カウントの管理に関しては自動化します。

22-4 文字列の扱い

Delphi 環境では、COM インターフェースをアクセスする際に扱われる文字列は全て UNICODE(WideString)になります。

22-5 配列の扱い

Delphi 環境では COM 互換の配列を扱うには少々テクニックが必要です。COM インターフェースがメソッドの引数や戻り値あるいはプロパティで使用する配列は SAFEARRAY と呼ばれるものであり、Delphi のそれとは大きく異なるからです。リソース・マネージャの FindRsrc メソッドが返す文字列配列(SAFEARRAY)を例に説明しましょう。

```
var
  hr : HRESULT;
  rm : IResourceManager3;
  {vi : IVisaSession;
  msg : IMessage;}

  sa : PSafeArray;
  rgElems : ^PWideChar;
  found: PWideChar;
  lLBound : longint;
  lUBound : longint;
  ndx: longint;

begin
    .
    .
    { Search for valid visa resource strings}
    hr:= rm.FindRsrc( '?*INSTR', sa);

    if sa <> nil then
    begin
      SafeArrayAccessData( sa, Pointer(rgElems));

      SafeArrayGetLBound( sa, 1, lLBound);
      SafeArrayGetUBound( sa, 1, lUBound);

      for ndx:= lLBound to lUBound do
      begin
        found:= rgElems^;
        //この時点で found は文字列を 1 個保持している

        Inc( rgElems);
      end;

      SafeArrayUnaccessData( sa);
      SafeArrayDestroyData( sa);
    end;
```

SafeArray 型を使用するため、uses 節に ActiveX ユニットを追加して下さい。FindRsrc メソッドから文字列配列を受取るには、一旦 SageArray 型ポインタ変数 sa を使います。更に

SafeArrayAccessData、SafeArrayGetLBound、SafeArrayGetUBound、SafeArrayUnaccessData、SafeArrayDestroyData の各 API 関数を駆使して内部エレメントにアクセスします。

22-6 エラー・コードの原則

計測器を制御するアプリケーションの場合、実行時に計測器と正常に通信できたかどうかを確認するのは重要な事です。どんなに堅牢に設計・実装されたアプリケーションでも、実行時に計測器のパワーが落ちていたり GPIB ケーブルが外れていれば、計測器との通信は正常に行えません。

VISA では全てのメソッド呼び出しとプロパティへのアクセスに対して一貫したエラー・コード体系を持っています。VISA C API 及び VISA COM API の全てのファンクションは、その戻り値として 32 ビット・コードを使用します。

Notes:

VISA C API では、各関数の戻り値は全て ViStatus 型 (=Long 型)と定義されています。一方 VISA COM API では、COM の標準に合わせて全てのメソッド(及びプロパティ・アクセス・メソッド)の戻り値は HRESULT 型 (=Long 型)と定義されています。VISA C API と VISA COM API の戻り値はコードのビット割り当てが若干異なりますが、同じ意味のステータス・コードは相互変換可能な関係にあります。本書は VISA COM API を中心とした説明をしていますので、HRESULT 型のステータス・コードを前提にして説明します。

一般に COM メソッド呼び出し時とプロパティ・アクセス時のステータス・コード(HRESULT)は次のように分類されます。VISA COM もその例外ではありません。

Table 22-1　HRESULT の大まかな分類

値の範囲	**説明**
HRESULT=0	成功
HRESULT>0	警告
HRESULT<0	エラー

HRESULT がマイナスの場合は、通常は、計測器との通信が失敗したか、呼び出しのパラメータに問題があったか、或いは VISA の動作状態がその呼び出しを拒否した場合などが考えれます。

Notes:

Delphi では HRESULT がマイナスであっても EOleException のような COM 例外は投げられません。各々のメソッド呼び出しのあと、HRESULT の値を IF 文等で評価する必要があります。

Notes:

VISA COM で定義されている HRESULT コードの詳細は、KI-VISA オンライン・ヘルプ又は本書の付録に掲載されています。

22-7 エラー・トラッピング

Delphi の COM ラッパ・クラスを利用する場合はメソッドの呼び出し直後に、必要に応じて HRESULT の値を評価する必要があります。HRESULT がマイナスの場合はエラーを意味します。下記の例では、HRESULT がマイナスだった場合に、EOleSysError 型の例外オブジェクトを自力で投げるように記述しています。try ブロックから投げた例外を捕獲するには except ブロックを使います。

```
var
   ...
  lenWritten: longint;
   ...

begin
  try
   ...

    hr:= msg.WriteString( 'CURR 20.0'#10, lenWritten);
```



```
    if( FAILED(hr)) then
      raise EOleSysError.Create('', hr, 0);

    hr:= msg.WriteString( 'VOLT 5.5'#10, lenWritten);
    if( FAILED(hr)) then
      raise EOleSysError.Create('', hr, 0);

   ...

  except
    on E: EOleSysError do
      Writeln( Format( 'Error:0x%08x', [E.ErrorCode]));
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
```

22-8 イベント・シンクの作成方法

サービス・リクエスト・イベントなどでイベント・コールバック機能を使う場合、VISA COM ではイベント・シンクを作成しなければなりません。イベント・シンクとは、そのイベントが要求するコールバック・インターフェース(この場合 IEventHandler インターフェース)を装備する、任意のコンポーネント・オブジェクトです。そのためには IEventHandler インターフェースを実装するコンポーネント・クラスを作成しなければなりません。

イベント・シンクの作成

イベント・シンクを作成するには、Delphi クラスを 1 個作成する必要があります。その Delphi クラスで IEventHandler インターフェースを実装することになります。Delphi 環境でクラス・モジュールを生成する方法はプロジェクトのタイプによって異なりますが、ここでは手作業で MyEventSink.pas ファイルを作成する例を示します。作成するのは TMyEventSink クラスとしましょう。

ここで作成するイベントシンクは、これも立派な COM クラスです(DLL 等に独立に配置しているわけではありませんが)。従って、COM クラスとしての実装の原則に従わなければなりません。

TMyEventSink のベースクラスになっている IEventHandler は、元々IUnknown を継承した派生インターフェースです。全ての COM クラスの原点になる IUnknown には、元々QueryInterface, AddRef, Release の 3 つのメソッド(仮想メンバー関数)が設計されています。そしてそこから派生した IEventHandler には、追加メソッドとして HandleEvent が設計されています。これら合計 4 つのメソッドは全て純仮想関数(pure virtual method)として記述されているため、そこからの最終的な派生クラス TMyEventSink は、これらを全て現実に実装する必要があります。(そうしないとビルドが通りません。)

幸い Delphi には IUnknown の 3 つのメソッドを実装済みのデリゲート・クラス TInterfacedObject 型が用意されています。

```
unit MyEventSink;

interface

uses
  System.SysUtils,
  ComObj,
  Windows,
  ActiveX,
  VisaComLib_TLB;

type
  TMyEventSink = class( TInterfacedObject, IEventHandler)
  public
     // IUnknown
```



```
    // (already implemented by TInterfacedObject)

    // IEventHandler
   function HandleEvent(
       const vi : IEventManager;
       const event : IEvent;
       userHandle : long
     ): HRESULT; stdcall;

end;

implementation

   // IEventHandler
  function TMyEventSink.HandleEvent(
       const vi : IEventManager;
       const event : IEvent;
       userHandle : long
     ): HRESULT; stdcall;
  var
       msg : IMessage;
       stb : short;
    hr : HRESULT;
  begin
       msg:= vi as IMEssage;
       hr:= msg.ReadSTB( stb);

       Result:= hr;
   end;

end.
```

23- IVI Configuration

ここでは KI-VISA に付属するもう一つのユーティリティ、IVI Configuration Utility の使い方を説明します。

Notes:

IVI Configuration Utility は VISA ライブラリを使う上での必須ツールではありません。IVI 計測器ドライバを「計測器クラス」のインターフェースで使用する際に、「仮想インストルメント」を設定するためのツールです。

この章での記述は、2003 年版 IVI ドライバガイドブック(IVI Configuration Utility 編、現在は廃刊)からの移植です。

23-1 ユーティリティの起動

Start ボタン➔Programs➔Kikusui IO Software➔KI-VISA(x64)メニューから、**IVI Config Utility** を起動する事ができます。起動直後の画面は下の図のようになっています。画面左側はツリー･ビューになっていて、2 つの階層が表示されています。上側は **Software Modules** で、この階層には現在インストールされている IVI-COM 計測器ドライバが全て表示されます。弊社製以外の、例えば Agilent Technologies 製や Tektronix 製の IVI-COM 計測器ドライバも表示されます。下側は **Logical Names** で、ここには利用可能な全ての仮想インストルメントが表示されます。下図の例では、**Kikusui4800** というロジカル･ネームで識別される仮想インストルメントが設定されています。

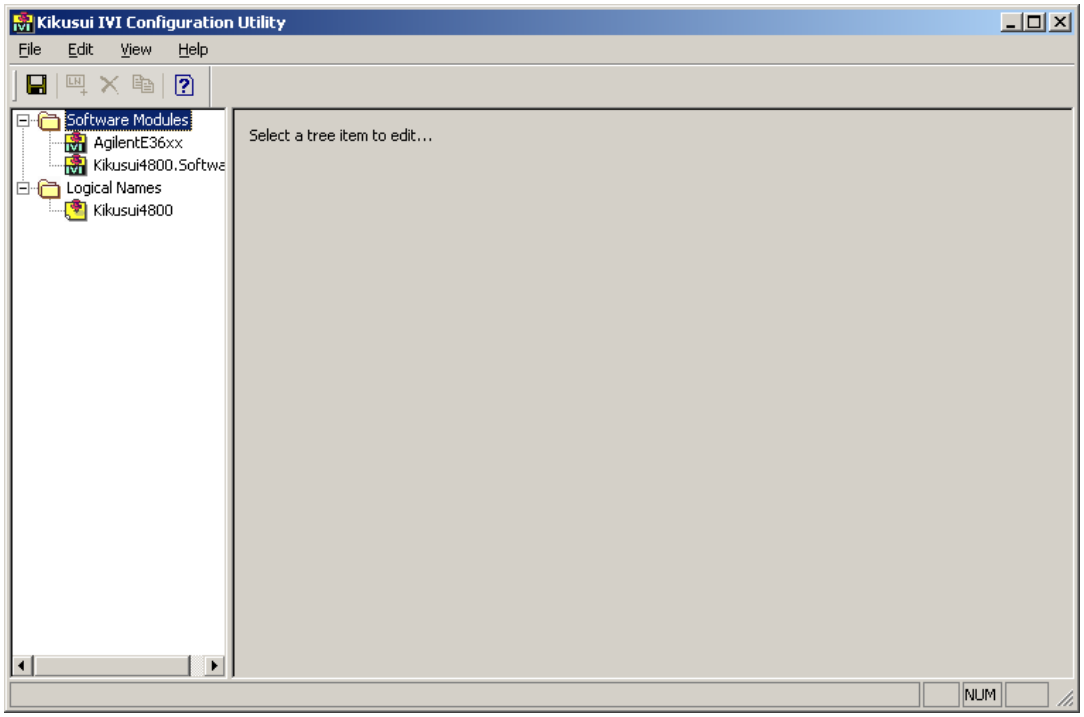

Figure 23-1 Kikusui IVI Config Utility メイン･スクリーン

ロジカル･ネーム Kikusui4800 は、Kikusui4800 IVI-COM ドライバをインストールした時に作成されるデフォルトの仮想インストルメントです。これをカスタマイズしても良いのですが、ここでは新規に作成する例を示します。

Notes:

IVI-COM 計測器ドライバをセットアップすると、デフォルトの仮想インストルメント(ロジカル･ネーム)が作成される場合があります。これは仮想インストルメントの作成をする際のサンプルとして用意されています。ロジカル･ネームを変更して利用しても良いでしょう。

デフォルトの仮想インストルメントは、IVI-COM ドライバのベンダーやバージョンによっては提供されない場合があります。

23-2 ロジカル･ネームの追加

ツリー表示の上で **Logical Names** と書かれた部分を右クリックすると、コンテキスト･メニューが表示されるので、**Add Logical Name...**を選択します。すると **Add Logical Name** ダイアログが表示されます。

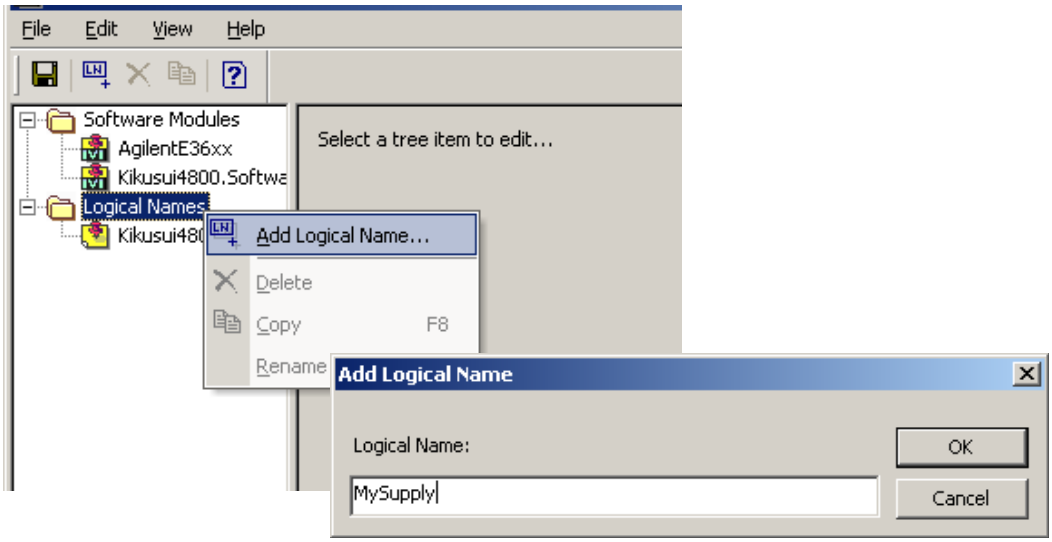

Figure 23-2 Logical Name の追加

ここでは新規作成したい仮想インストルメントの名前を指定します。ロジカル･ネームに使用できる文字は、英数字(A..Z, a..z, 0..9)、アンダースコア(_)、及び感嘆符(!)のみです。また、既に存在するロジカル･ネームは使用できません。ロジカル･ネームはケース･センシティブ(大文字･小文字の区別あり)です。

ここでは、例として MySupply という名前を指定します。すると、この名前が **Logical Names** に追加されて表示されます。ロジカル･ネームを与えて仮想インストルメントを作成したら、**Logical Name**、**Driver Session**、**Hardware Asset**、**Virtual Names**、のタブ･ページで具体的な設定を行います。

23-3 タブ･ページでの設定

Logical Name タブ

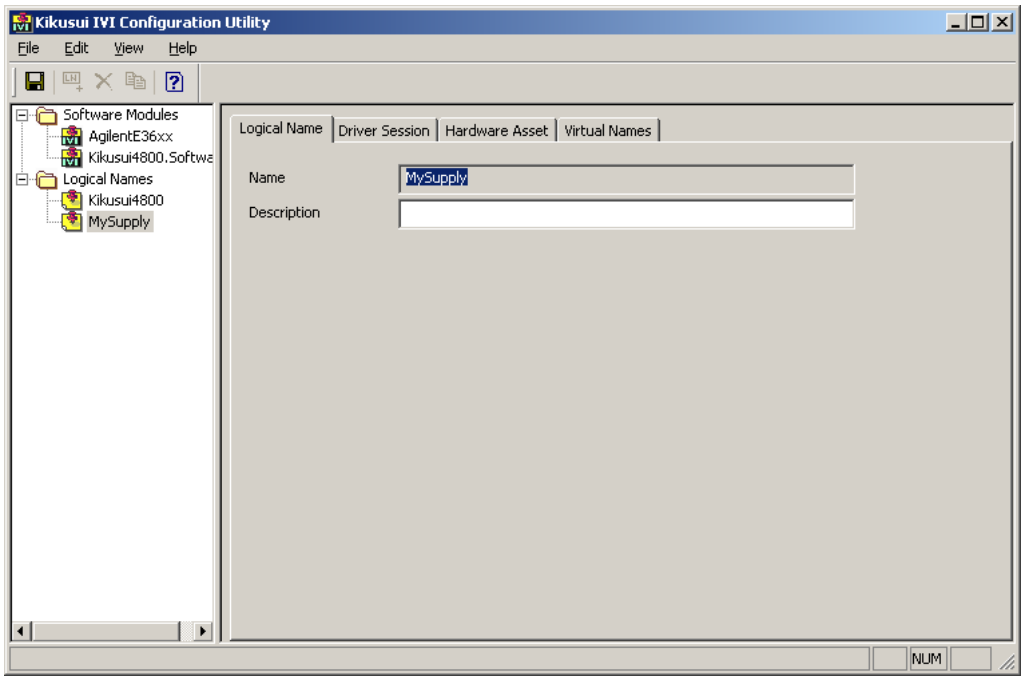

Figure 23-3 Logical Name タブ

このページで設定できるのは **Description** だけです。機能上の特別な意味は無いのでここでは省略します。空白のまま放置して構いません。

Driver Session タブ

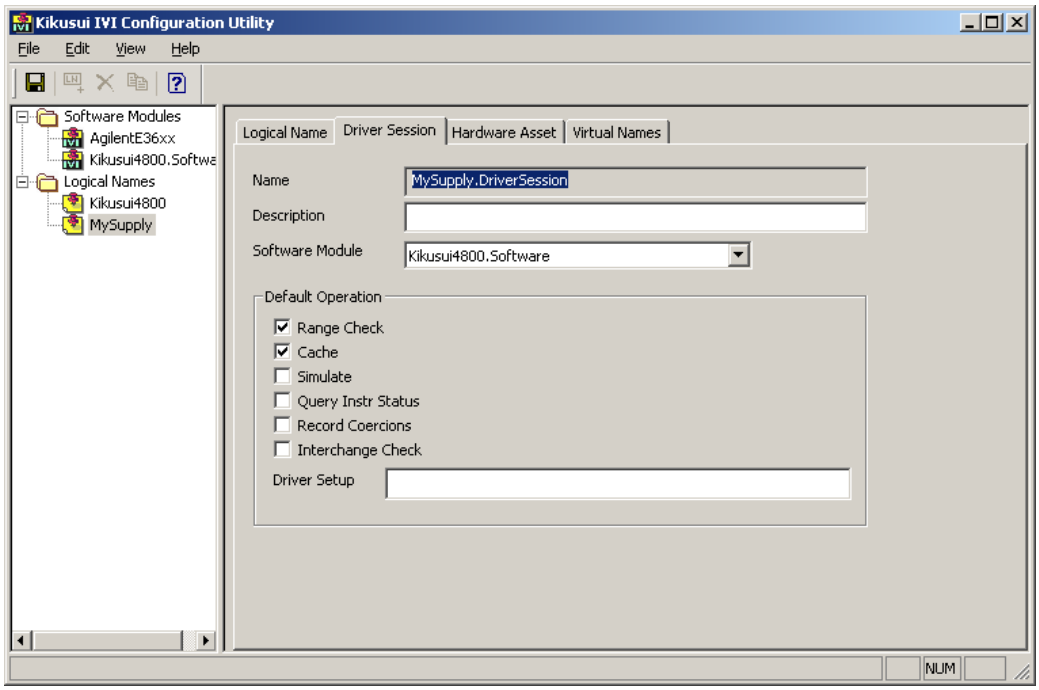

Figure 23-4 Driver Session タブ

Description は機能上の特別な意味は無いので空白のまま放置して構いません。**Software Module** では、コンボボックスで提示されるソフトウェア・モジュールを 1 つ選択します。これは、この仮想インストルメントをどの計測器ドライバでホストするかを決定する重要な設定です。ここでは Kikusui4800.Software でホストしてもらうので、それを選択してください。選択肢として提示されたもの以外は指定できません。

Figure 23-5 Software Module の選択

Default Operation では、**Range Check**, **Cache**、**Simulate**、**Query Instr Status**、**Record Coercions**、**Interchange Check**、の ON/OFF 設定、及び **Driver Setting** 文字列を設定します。ここで設定した内容は、アプリケーションが `Initialize` メソッドを呼び出したときのデフォルト設定となります。

Figure 23-6 Default Operation の設定

Range Check が有効になっている場合、ドライバはプロパティの put_アクセスにおける入力値とメソッド呼び出しのパラメータ値をチェックします。この機能は、計測器に対して範囲外の値を送る前の事前チェックとして行われます。アプリケーションのデバッグ時に役に立ちます。

Cache が有効になっている場合、ドライバは不必要な I/O を排除します。たとえば、もし或る値が計測器に既に設定されている場合、計測器に同じ設定値を再度送ることは冗長です。無駄な I/O を排除することでアプリケーションのパフォーマンスが向上します。

Simulate が有効になっている場合、ドライバは 計測器への I/O を行わず、出力パラメータにシミュレートされた値を返します。この機能はアプリケーション開発で実際の計測器が準備できていない場合に役に立ちます。

Query Instr Status が有効になっている場合、ドライバは計測器への I/O を行う各メソッド呼び出し或いはプロパティへのアクセスの後に計測器への問い合わせを行います。計測器がエラーを報告した場合、`ErrorQuery` メソッドを使用して計測器からエラーメッセージを取得することができます。アプリケーションのデバッグ時に役に立ちますが、開発が完了したら無効に設定すればよいでしょう。

Record Coercions が有効になっている場合、ドライバは LONG と DOUBLE 値に関してコアーション(値の強制)記録のリストを作成します。ドライバがコアーション記録をサポートしない場合は、この設定は無視されます。

Interchange Check が有効になっている場合、ドライバは インターチェンジャビリティ警告の記録を作成します。ドライバがインターチェンジャビリティのチェックをサポートしない場合は、この設定は無視されます。

Hardware Asset タブ

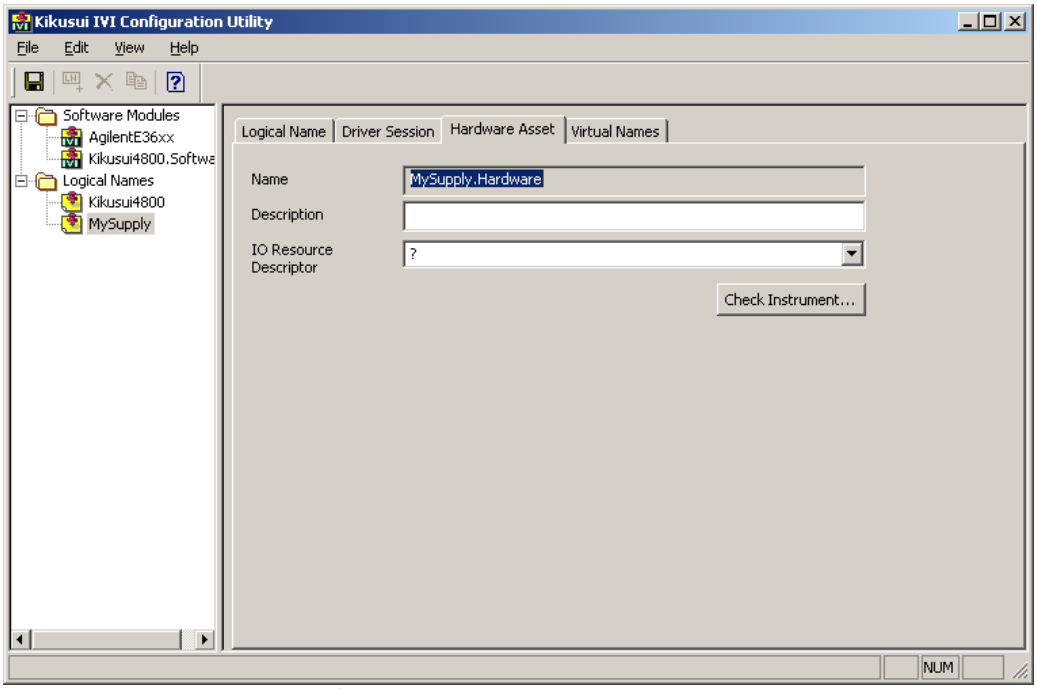

Figure 23-7 Hardware Asset タブ

Description は機能上の特別な意味は無いので空白のまま放置して構いません。**IO Resource Descriptor** のコンボボックスには現在使用可能な VISA I/O リソースが提示されます。選択肢の中から適切なものを選んで下さい。接続されていない計測器や TCP/IP 上の計測器などは、リストに表示されません。この場合には直接 VISA リソースを入力してください。アプリケーションがロジカル・ネームをパラメータにして `Initialize` メソッドを呼び出したとき、ここで設定された VISA リソースを使用して計測器との I/O を行います。

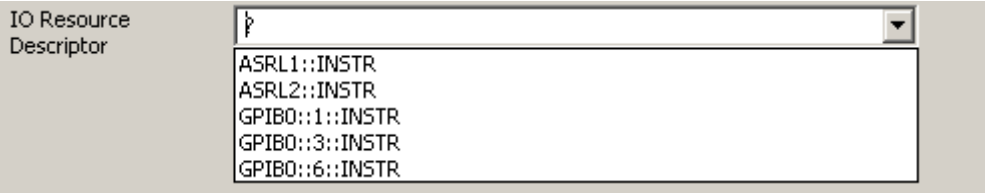

Figure 23-8 IO Resource Descriptor の選択

Check Instrument ボタンをクリックすると、VISA COM ライブラリを使用した簡単な I/O テストが行えます。

Virtual Name タブ

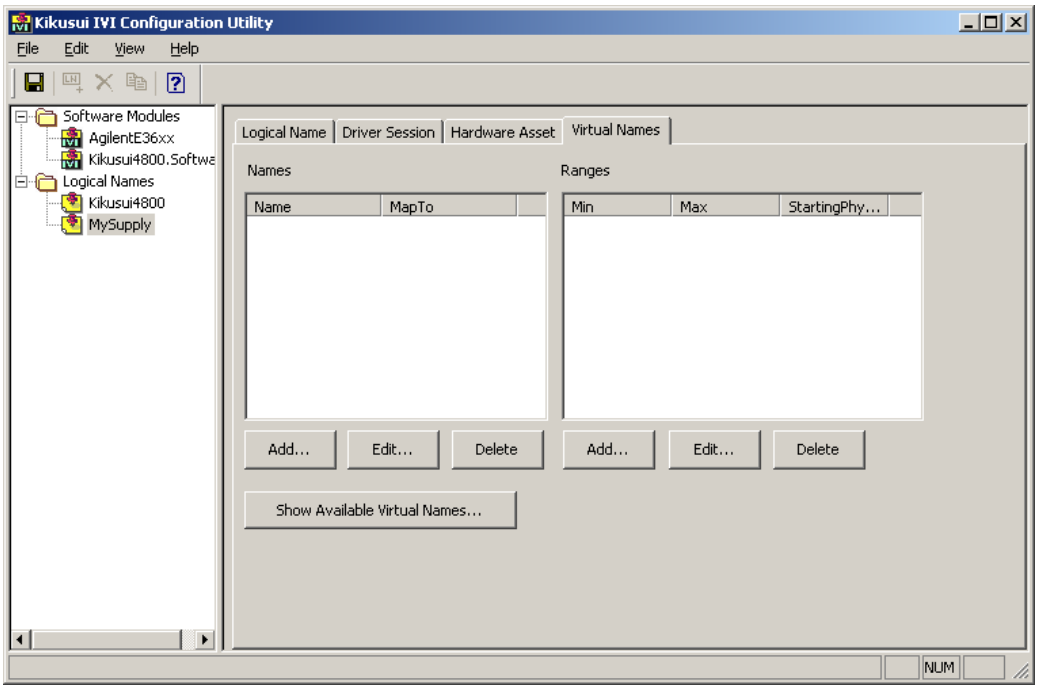

Figure 23-9 Virtual Names タブ

ここでは、バーチャル･ネームとフィジカル・ネームのマッピングを行います。これは、リピーテッド・キャパビリティを使用する場合の、オブジェクトの識別に使われます

リピーテッド・キャパビリティとは、全く同じ或いは類似する複数のオブジェクトを配列又はコンテナのように扱う概念です。例えば IviDCPwr クラスに属する計測器ドライバでは、出力チャンネルを複数装備したマルチ・トラック DC 電源装置を前提に設計されています。別の例では、IviScope クラスに属する計測器ドライバは、複数のトレース・チャンネルを装備するオシロスコープを前提に設計されています。このように、類似するオブジェクトを複数保持する場合にはリピーテッド･キャパビリティを仕様する事が IVI 仕様で推奨されています。

IviDCPwr クラスに属する Kikusui4800 IVI-COM ドライバでは、"N5!C1"のようなルールの名前で出力オブジェクトを識別しますが、このような名前は計測器ドライバ独自のものです。このようなドライバ独自の名前をフィジカル・ネームと言います。しかしインターチェンジャビリティ機能を利用するアプリケーションから特定ドライバに依存したフィジカル・ネームを使うわけにはいきません。そこでバーチャル・ネームを作成し、それがどのフィジカル・ネームにマップされるかを設定する必要があります。

Names 側にある Add ボタンをクリックすると、Virtual Name ダイアログが表示されます。

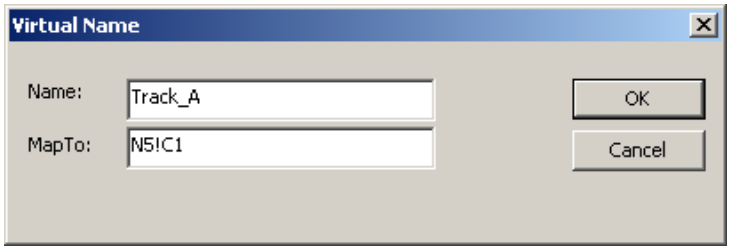

Figure 23-10 Virtual Name ダイアログ

ここでは例として、**Name** の項目に Track_A、**MapTo** の項目に N5!C1 とタイプします。OK ボタンをクリックするとバーチャル・ネームが 1 つ出来上がりました。

この設定により、アプリケーションがリピーテッド・キャパビリティを通じて"Track_A"のオブジェクトを参照しようとすると、計測器ドライバはそのマップ先である"N5!C1"を指定されたと解釈します。これにより、計測器ドライバ内で"N5!C1"という名前で存在するオブジェクトを、"Track_A"という名前で参照できるわけです。

同様の手順で必要な数だけマップを作成してください。同じ名前を Name に複数回指定する事はできません。全てのバーチャル・ネームは一意的な名前でなければなりません。しかし、異なるバーチャル・ネームが同じフィジカル・ネームにマップする事は構いません。これは同じフィジカル･ネームに対して、複数のバーチャル･ネームでエイリアス(別名)を作った事になります。

また、全てのフィジカル･ネームに対して対応するバーチャル･ネームを作成する必要はありません。例えば、Kikusui4800 IVI-COM ドライバでは"N30!C4"は有効なフィジカル･ネームですが、アプリケーションがそのオブジェクトを参照しないのであれば、それにマップされるバーチャル･ネームを作る必要はありません。

現在利用可能なバーチャル･ネームとそのマップ先フィジカル･ネームを全て表示するには、**Show Available Virtual Names** ボタンをクリックします。

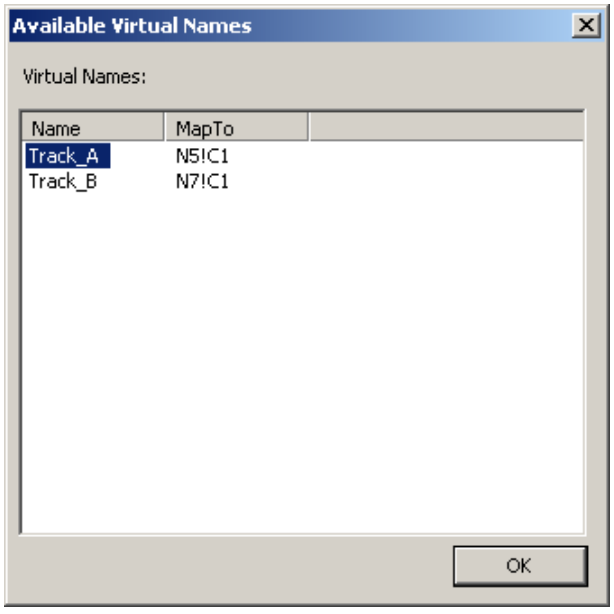

Figure 23-11 Show Available Virtual Names ダイアログ

マップ先として利用可能なフィジカル･ネームは、**Driver Session** タブの **Software Module** コンボボックスで選択されたソフトウェア･モジュール(計測器ドライバ・ソフトウェア)によって全く異なります。その情報を得るには、計測器ドライバのオンライン・ヘルプや Readme 文書を参照するか、または各言語別に用意されたプログラミング・ガイドブックに示されるサンプル･コードを使って取得してください。

23-4 設定の保存

仮想インストルメントの設定が完了したら、**File | Save** メニューで設定を保存してください。

24- 付録(C) VISA COM Quick Reference

各インターフェースのメソッドとプロパティの概略を示します。詳細については KI-VISA オンラインヘルプを参照して下さい。下に各メソッドとプロパティに使用される型の言語別表記法の違いを示します。

Table 24-1　型の説明

表記	**VB.NET (managed)**	**C# (managed)**	**C++/CLI (managed)**	**VB6/VBA (unmanaged)**	**C++ (unmanaged)**
Byte (8bit)	Byte	byte	byte	Byte	BYTE
Integer (16bit)	Short	short	short	Integer	short
Long (32bit)	Integer	int	int	Integer	short
Double (64bit)	Double	double	doube	Double	double
String	System.String	System.String	System::String	String	BSTR
Boolean (16bit)	Boolean	Boolean	Boolean	Boolean	VARIANT_BOOL
IXxxxx (interface)	IXxxxx	IXxxxx	IXxxxx^	IXxxxx	IXxxxxPtr

24-1 IVisaSession インターフェース

Table 24-2　IVisaSession インターフェースのメソッド

メソッド	**説明**
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする
Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる

Table 24-3 IVisaSession インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値(ms)を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す

24-2 IBaseMessage インターフェース

Table 24-4 IBaseMassage インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする
Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる
AssertTrigger	デバイス・トリガ(GET コマンド)を送信する
Clear	セレクテッド・デバイス・クリア(SDC コマンド)を送信する
ReadSTB	シリアル・ポーリングを実行する

Table 24-5 IBaseMassage インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
IOProtocol	IOProtocol (W/R)	IO プロトコルを設定・取得する
SendEndEnabled	Boolean (W/R) デフォルト TRUE	メッセージの最終バイト送信時に END インジケータを付けるかどうかを設定・取得する
TerminationCharacter	Byte (W/R) デフォルト hex 0A	メッセージ送信時又は受信時に使われるターミネーション・キャラクタを設定・取得する
TerminationCharacterEnabled	Boolean (R) デフォルト FALSE	メッセージ受信時にターミネーション・キャラクタの考慮をするかどうかを設定・取得する

24-3 IMessage インターフェース

Table 24-6 IMassage インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする

Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる
AssertTrigger	デバイス・トリガ（GET コマンド）を送信する
Clear	セレクテッド・デバイス・クリア(SDC コマンド)を送信する
ReadSTB	シリアル・ポーリングを実行する
Write	配列で指定されたメッセージを同期モードで送信する
Read	メッセージを同期モードで受信し配列に格納する
WriteString	文字列で指定されたメッセージを同期モードで送信する
ReadString	メッセージを同期モードで受信し文字列に格納する

Table 24-7 IMassage インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
IOProtocol	IOProtocol (W/R) デフォルト PROT_NORMAL	IO プロトコルを設定・取得する
SendEndEnabled	Boolean (W/R) デフォルト TRUE	メッセージの最終バイト送信時に END インジケータを付けるかどうかを設定・取得する
TerminationCharacter	Byte (W/R) デフォルト hex 0A	メッセージ送信時又は受信時に使われるターミネーション・キャラクタを設定・取得する
TerminationCharacterEnabled	Boolean (R) デフォルト FALSE	メッセージ受信時にターミネーション・キャラクタの考慮をするかどうかを設定・取得する

24-4 IAsyncMessage インターフェース

Table 24-8 IAsyncMassage インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する

LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする
Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる
AssertTrigger	デバイス・トリガ（GET コマンド）を送信する
Clear	セレクテッド・デバイス・クリア(SDC コマンド)を送信する
ReadSTB	シリアル・ポーリングを実行する
Write	配列で指定されたメッセージを非同期モードで送信する
Read	メッセージを非同期モードで受信する
WriteString	文字列で指定されたメッセージを非同期モードで送信する
Terminate	非同期モードの IO 動作を中止する

Table 24-9 IAsyncMassage インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
IOProtocol	IOProtocol (W/R) デフォルト PROT_NORMAL	IO プロトコルを設定・取得する
SendEndEnabled	Boolean (W/R) デフォルト TRUE	メッセージの最終バイト送信時に END インジケータを付けるかどうかを設定・取得する
TerminationCharacter	Byte (W/R) デフォルト hex 0A	メッセージ送信時又は受信時に使われるターミネーション・キャラクタを設定・取得する
TerminationCharacterEnabled	Boolean (R) デフォルト FALSE	メッセージ受信時にターミネーション・キャラクタの考慮をするかどうかを設定・取得する

24-5 IHislipInstr インターフェース

Table 24-10 IHislipInstr インターフェースのメソッド

メソッド	説明
ControlREN	リモート・ローカルを操作する

Table 24-11 IHislipInstr インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
MaxMessage	Long (W/R)	最大メッセージサイズを設定又は取得する
OverlappedEnabled	Boolean (W/R)	HiSLIP オーバーラップモードにするかどうかを設定又は取得する
ProtocolVersion	Long (R)	計測器と VISA でネゴシエートされた HiSLIP プロトコルのバージョンを返す

24-6 IGpib インターフェース

Table 24-12 IGpib インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする
Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる
ControlREN	リモート・ローカルを操作する

Table 24-13 IGpib インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
PrimaryAddress	Integer (W/R)	プライマリ・アドレスを設定・取得する
RENState	LineState (R)	REN ラインの状態を取得する
RepeatAddressingEnabled	Boolean (W/R) デフォルト TRUE	メッセージの送受信時にアドレッシングを毎回行う
SecondaryAddress	Integer (W/R)	セカンダリ・アドレスを設定・取得する
UnaddressingEnabled	Boolean (W/R) デフォルト FALSE	メッセージの送受信後にアドレッシングの解除(UNT/UNL)を行う

24-7 ISerial インターフェース

Table 24-14 ISerial インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする

Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる
SetBufferSize	IO バッファのサイズを設定する
Flush	IO バッファの中身を消去する

Table 24-15 ISerial インターフェースのプロパティ

プロパティ	**タイプ (アクセス・モード)**	**説明**
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
BytesAvailable	Long (R)	受信バッファに残っているデータ・バイト数を返す
BaudRate	Long (W/R) デフォルト 9600	ボーレートを設定・取得する
DataBits	Integer (W/R) デフォルト 8	ストップビットを設定・取得する
ClearToSendState	LineState (R)	CTS(Clear To Send)ラインの状態を取得する
DataCarrierDetectState	LineState (R)	DCT(Data Carrier Detect)ラインの状態を取得する
DataSetReadyState	LineState (R)	DSR(Data Set Ready)ラインの状態を取得する

DataTerminalReadyState	LineState (W/R)	DTR(Data Terminal Ready)ラインを制御する
EndIn	SerialEndConst (W/R) デフォルト ASRL_END_TERMCHAR	受信時のターミネーション条件を設定・取得する
EndOut	SerialEndConst (W/R) デフォルト ASRL_END_NONE	送信時のターミネーション条件を設定・取得する
FlowControl	SerialFlowControl (W/R) デフォルト ASRL_FLOW_NONE	フロー制御の条件を設定・取得する
Parity	SerialParity (W/R) デフォルト ASRL_PAR_NONE	パリティを設定・取得する
RingIndicatorState	LineState (R)	RI(Ring Indicator)ラインの状態を取得する
RequestToSendState	LineState (W/R)	RTS(Request To Send)ラインを制御する
StopBits	SerialStopBits (W/R) デフォルト ASRL_STOP_ONE	ストップビットを設定・取得する
ReplacementCharacter	Byte (W/R) デフォルト hex 00	パリティなどによるエラー発生時の置き換えキャラクターを設定・取得する
XONCharacter	Byte (W/R) デフォルト hex 11	X フロー制御における XON キャラクターを設定・取得する
XOFFCharacter	Byte (W/R) デフォルト hex 13	X フロー制御における XOFF キャラクターを設定・取得する

24-8 IUsb インターフェース

Table 24-16 IUsb インターフェースのメソッド

メソッド	**説明**
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする
Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる
ControlIn	ベンダー固有リクエストを送付しレスポンスを取得する
ControlOut	ベンダー固有リクエストを送付する
ControlREN	リモート・ローカルを操作する

Table 24-17 IUsb インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
ManufacturerID	Integer (R)	USB デバイスのベンダーID を返す
ManufacturerName	String (R)	USB デバイスのベンダー名を返す
ModelCode	Integer (R)	USB デバイスのプロダクト ID を返す
ModelName	String (R)	USB デバイスのプロダクト名を返す
Is4882Compliant	Boolean (R)	488.2 準拠の計測器であれば TRUE を返す
UsbSerialNumber	String (R)	USB デバイスのシリアル番号を返す
UsbInterfaceNumber	Integer (R)	USB インターフェース番号を返す
UsbMaximumInterruptSize	Integer (W/R)	USB デバイスが送信するインタラプト IN パイプのバッファ・サイズを設定・取得する
Usbprotocol	Integer (R)	USB プロトコル番号を返す

24-9 ITcpipInstr インターフェース

Table 24-18 ITcpipInstr インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする
Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる

Table 24-19 ITcpipInstr インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
Address	String (R)	デバイスの IP アドレスを返す
DeviceName	String (R)	LAN デバイス名を返す
HostName	String (R)	ホスト名を返す

24-10 ITcpipSocket インターフェース

Table 24-20 ITcpipSocket インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする
Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる
SetBufferSize	IO バッファのサイズを設定する
Flush	IO バッファの中身を消去する

Table 24-21 ITcpipSocket インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
Address	String (R)	デバイスの IP アドレスを返す
KeepAlive	Boolean (W/R)	キープアライブ機能を有効にする
NoDelay	Boolean (W/R)	Nagle アルゴリズムを有効にする
HostName	String (R)	ホスト名を返す
Port	Integer (R)	Port 番号を返す

24-11 IGpibIntfc インターフェース

Table 24-22 IGpibIntfc インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする
Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる
Command	配列で指定されたコマンドを ATN TRUE で送出する
ControlATN	ATN ラインを制御する
ControlREN	REN ラインを制御する
PassControl	指定されたアドレスに CIC(Contoller In Charge)権限を渡す
SendIFC	IFC(Interface Clear)信号を送出し、CIC 権限を取り返す

Table 24-23 IGpibIntfc インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
AddressingState	AddressingState (R)	アドレッシング・ステートを取得する
ATNState	LineState (R)	ATN ラインの状態を取得する
DevStatusByte	Byte (W/R)	ステータス・バイトを設定する
CICState	Boolean (R)	CIC(Controller In Charge)かどうかを取得する
HS488CBLLength	Integer (W/R)	HS488 のケーブル長を設定・取得する
NDACState	LineState (R)	NDAC ラインの状態を取得する
PrimaryAddress	Integer (W/R)	プライマリ・アドレスを設定・取得する
RENState	LineState (R)	REN ラインの状態を取得する
SecondaryAddress	Integer (W/R)	セカンダリ・アドレスを設定・取得する
SRQState	LineState (R)	SRQ ラインの状態を取得する

SysControllerState	Boolean (W/R)	システム・コントローラの権限を設定・取得する

24-12 IGpibIntfcMessage インターフェース

Table 24-24 IGpibIntfcMassage インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
LockRsrc	リソースをロックする
UnlockRsrc	リソースをアンロックする
Init	セッションをオープンする(リソース・マネージャの Open と同じ機能)
Close	セッションを閉じる
AssertTrigger	デバイス・トリガ(GET コマンド)を送信する
Write	配列で指定されたメッセージを同期モードで送信する
Read	メッセージを同期モードで受信し配列に格納する
WriteString	文字列で指定されたメッセージを同期モードで送信する
ReadString	メッセージを同期モードで受信し文字列に格納する

Table 24-25 IGpibIntfcMassage インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す
HardwareInterfaceName	String (R)	ハードウェア・インターフェース名を返す
HardwareInterfaceNumber	Integer (R)	ハードウェア・インターフェース番号を返す
HardwareInterfaceType	Integer (R)	ハードウェア・インターフェースのタイプ番号を返す
LockState	AccessMode (R)	現在のリソース・ロッキング状態を返す
OptionString	String (R)	設定可能な全ての属性値を文字列形式で返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
ResourceName	String (R)	リソース名を返す
SessionType	String (R)	セッション・タイプを文字列で返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
Timeout	Long (W/R) デフォルト 2000	IO タイムアウトの値を設定又は取得する
LastStatus	Long (R)	セッションに対して最後に行ったメソッド呼び出し・プロパティ・アクセスの HRESULT を返す
SendEndEnabled	Boolean (W/R) デフォルト TRUE	メッセージの最終バイト送信時に END インジケータを付けるかどうかを設定・取得する
TerminationCharacter	Byte (W/R) デフォルト hex 0A	メッセージ送信時又は受信時に使われるターミネーション・キャラクタを設定・取得する
TerminationCharacterEnabled	Boolean (R) デフォルト FALSE	メッセージ受信時にターミネーション・キャラクタの考慮をするかどうかを設定・取得する

24-13 IEvent インターフェース

Table 24-26 IEvent インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
Close	イベントを閉じる

Table 24-27 IEvent インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
type	EventType (R)	イベントのタイプを返す
CustomEventTypeNumber	Long (R)	カスタム・イベントのタイプ番号を返す

24-14 IEventIOCompletion インターフェース

Table 24-28 IEventIOCompletion インターフェースのメソッド

メソッド	説明
GetAttribute	指定された属性値を取得する
SetAttribute	指定された属性値を設定する
Close	イベントを閉じる

Table 24-29 IEventIOCompletion インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
type	EventType (R)	イベントのタイプを返す
CustomEventTypeNumber	Long (R)	カスタム・イベントのタイプ番号を返す
IOStatus	HRESULT (R)	IO 動作の結果を返す
jobID	Long (R)	ジョブ ID を返す
ReturnCount	Long (R)	IO 動作で転送されたデータ・バイト数を返す
ReadBuffer	SafeArray(Byte) (R)	IO 動作で読み込まれたデータを配列で返す
ReadBufferAsString	String (R)	IO 動作で読み込まれたデータを文字列で返す

24-15 IResourceManager3 インターフェース

Table 24-30 IResourceManager3 インターフェースのメソッド

メソッド	説明
FindRsrc	指定された正規表現にマッチするリソースを検索する
Open	指定されたリソースでセッションをオープンする
ParseRsrc	指定されたリソースが有効なものか評価する
ParseRsrcEx	指定されたリソースが有効なものか評価する

Table 24-31 IResourceManager3 インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
ComponentVersion	Long (R)	コンポーネントのバージョンを返す
Description	String (R)	説明文字列を返す
ProgID	String (R)	コンポーネントの Program ID 文字列を返す
SoftwareManufacturerID	Integer (R)	コンポーネントを作成したベンダーの ID を返す
SoftwareManufacturerName	String (R)	コンポーネントを作成したベンダー名を返す
SpecVersion	Long (R)	コンポーネントが準拠する VISA 仕様のバージョンを返す

24-16 IFormattedIO488 インターフェース

Table 24-32 IFormattedIO488 インターフェースのメソッド

メソッド	説明
WriteString	指定されたメッセージを同期モードで送信する
WriteNumber	指定された数値を書式化して同期モードで送信する
WriteList	指定された複数項目を同期モードで送信する
WriteIEEEBlock	指定された配列を IEEE488.2 ブロックデータとして同期モードで送信する
ReadString	レスポンスを同期モードで受信し文字列のまま返す
ReadNumber	レスポンスを同期モードで受信し数値として返す
ReadList	レスポンスを同期モードで受信し複数項目に分解して返す
ReadIEEEBlock	IEEE488.2 ブロックデータのレスポンスを同期モードで受信する
FlushWrite	送信用バッファをフラッシュする
FlushRead	受信用バッファをフラッシュする
SetBufferSize	送信又は受信バッファのサイズを設定する

Table 24-33 IFormattedIO488 インターフェースのプロパティ

プロパティ	タイプ (アクセス・モード)	説明
IO	IMessage (W/R)	VisaSession の IMessage インターフェースを設定又は照会する
InstrumentBigEndian	Boolean (W/R)	488.2 ブロック転送におけるエンディアン条件を設定又は照会する

25- 付録(B) VISA COM インターフェース階層

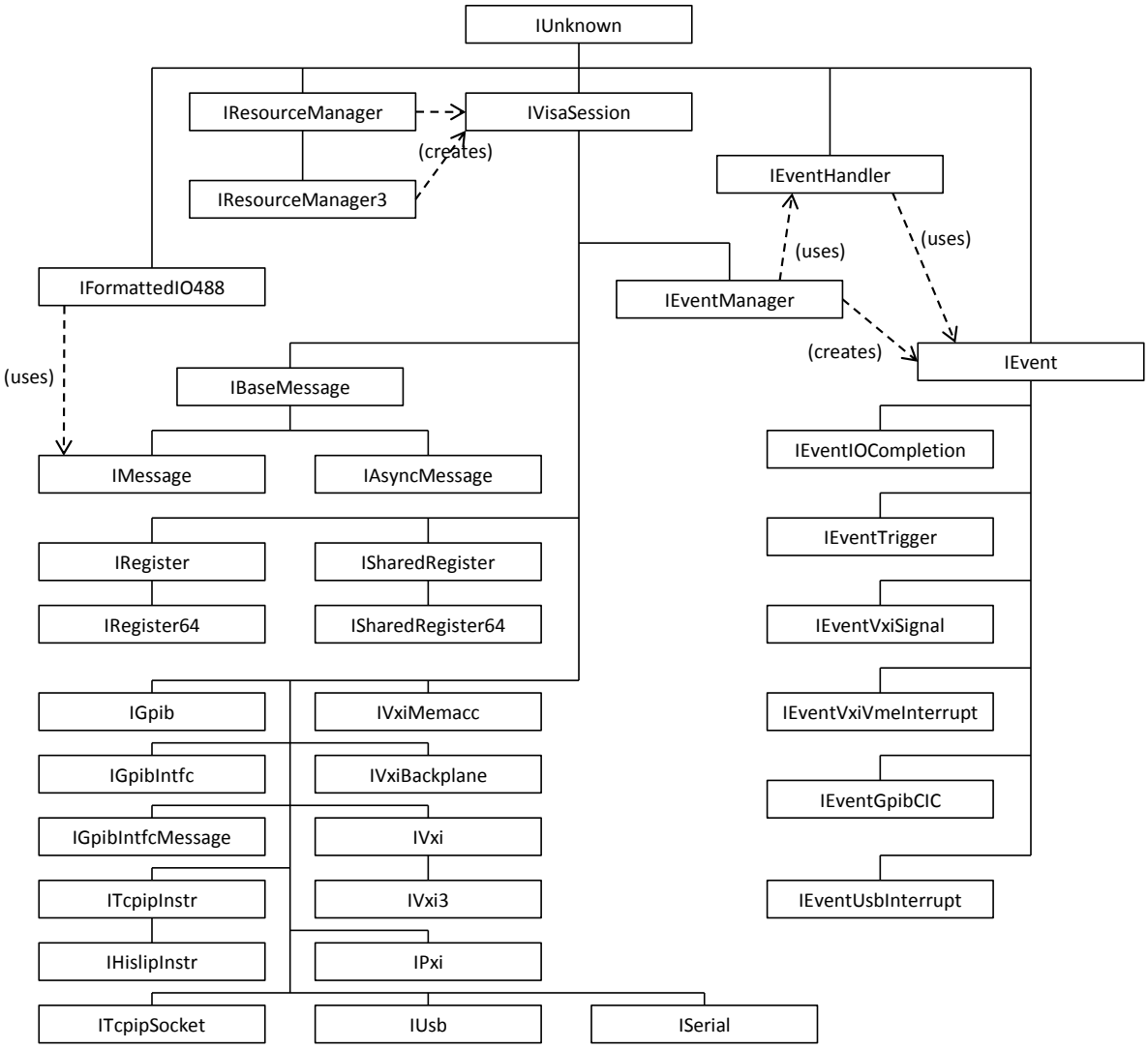

Figure 25-1 VISA COM Interface (IVI Foundation, VPP-4.3.4:VISA COM 仕様書 Rev5.0 より引用)

26- 付録(C) VISA COM API ステータス・コード

Table 26-1 VISA COM HRESULT 一覧

Symbol	Hex Value	Decimal Value	Description
E_UNEXPECTED	0x8000FFFF	-2147418113	A serious internal error occurred.
E_POINTER	0x80004003	-2147467261	Invalid pointer.
E_NOTIMPL	0x80000001	-2147483647	Not implemented.
E_FAIL	0x80004005	-2147467259	Failed.
E_INVALIDARG	0x80070057	-2147024809	One of the arguments did not contain a legal value.
S_OK	0x0	0	Operation completed successfully.
S_VISA_SUCCESS	0x0	0	Operation completed successfully
S_VISA_EVENT_EN	0x40002	262146	The event type is already enabled.
S_VISA_EVENT_DIS	0x40003	262147	The event type is already disabled.
S_VISA_QUEUE_EMPTY	0x40004	262148	The event queue is disabled.
S_VISA_TERM_CHAR	0x40005	262149	The read operation terminated due to reception of the termination character.
S_VISA_MAX_CNT	0x40006	262150	The read operation terminated due to the reception of the requested number of bytes
S_VISA_DEV_NPRESENT	0x4007d	262269	Session opened successfully, but the device at the specified address is not responding.
S_VISA_QUEUE_NEMPTY	0x40080	262272	Wait terminated successfully on receipt of an event notification. There is still at least one more event occurrence of the type specified by inEventType available for this session.
S_VISA_TRIG_MAPPED	0x4007e	262270	The path from trigSrc to trigDest is already mapped.
S_VISA_NCHAIN	0x40098	262296	Event handled successfully. Do not invoke any other handlers on this session for this event.
S_VISA_NESTED_SHARED	0x40099	262297	Specified access mode is successfully acquired, and this session has nested shared locks.
S_VISA_NESTED_EXCLUSIVE	0x4009a	262298	Call succeeded, but this session still has nested exclusive locks.
S_VISA_SYNC	0x4009b	262299	Operation performed synchronously.
S_VISA_CONFIG_NLOADED	0x40077	262263	The specified configuration either does not exist or could not be loaded; using VISA-specified defaults.
S_VISA_NULL_OBJECT	0x40082	262274	The specified object reference is uninitialized.
S_VISA_NSUP_ATTR_STATE	0x40084	262276	Although the specified attribute state is valid, it is not supported by this implementation.
S_VISA_UNKNOWN_STATUS	0x40085	262277	The status code passed to the operation could not be interpreted.
S_VISA_NSUP_BUF	0x40088	262280	The specified buffer is not supported.

E_VISA_SYSTEM_ERROR	0x80040000	-2147221504	Unknown system error (miscellaneous error).
E_VISA_INV_OBJECT	0x8004000e	-2147221490	The given session or object reference is invalid (both are the same value).
E_VISA_RSRC_LOCKED	0x8004000f	-2147221489	Specified type of lock cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested.
E_VISA_INV_EXPR	0x80040010	-2147221488	Invalid expression specified for search.
E_VISA_RSRC_NFOUND	0x80040011	-2147221487	Specified expression does not match any devices.
E_VISA_INV_RSRC_NAME	0x80040012	-2147221486	Invalid resource reference specified. Parsing error.
E_VISA_INV_ACC_MODE	0x80040013	-2147221485	Invalid access mode.
E_VISA_TMO	0x80040015	-2147221483	The operation terminated due to a timeout.
E_VISA_CLOSING_FAILED	0x80040016	-2147221482	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.
E_VISA_INV_DEGREE	0x8004001b	-2147221477	Specified degree is invalid.
E_VISA_INV_JOB_ID	0x8004001c	-2147221476	Specified job identifier is invalid.
E_VISA_NSUP_ATTR	0x8004001d	-2147221475	The specified attribute is not defined by the referenced session, event, or find list.
E_VISA_NSUP_ATTR_STATE	0x8004001e	-2147221474	The specified state of the attribute is not valid, or is not supported as defined by the session, event, or find list.
E_VISA_ATTR_READONLY	0x8004001f	-2147221473	The specified attribute is read-only.
E_VISA_INV_LOCK_TYPE	0x80040020	-2147221472	The specified type of lock is not supported by this resource.
E_VISA_INV_ACCESS_KEY	0x80040021	-2147221471	The requestedKey value passed in is not a valid access key to the specified resource.
E_VISA_INV_EVENT	0x80040026	-2147221466	Specified event type is not supported by the resource.
E_VISA_INV_MECH	0x80040027	-2147221465	Invalid mechanism specified.
E_VISA_HNDLR_NINSTALLED	0x80040028	-2147221464	A handler is not currently installed for the specified event. The session cannot be enabled for the VI_HNDLR mode of the callback mechanism.
E_VISA_INV_HNDLR_REF	0x80040029	-2147221463	The given handler reference is invalid.
E_VISA_INV_CONTEXT	0x8004002a	-2147221462	Specified event context is invalid.
E_VISA_QUEUE_OVERFLOW	0x8004002d	-2147221459	The Event queue overflowed.
E_VISA_NENABLED	0x8004002f	-2147221457	The session must be enabled for events of the specified type in order to receive them.
E_VISA_ABORT	0x80040030	-2147221456	The asynchronous job as terminated early.
E_VISA_RAW_WR_PROT_VIOL	0x80040034	-2147221452	Violation of raw write protocol occurred during transfer.
E_VISA_RAW_RD_PROT_VIOL	0x80040035	-2147221451	Violation of raw read protocol occurred during transfer.
E_VISA_OUTP_PROT_VIOL	0x80040036	-2147221450	Device reported an output protocol error during transfer.

E_VISA_INP_PROT_VIOL	0x80040037	-2147221449	Device reported an input protocol error during transfer.
E_VISA_BERR	0x80040038	-2147221448	Bus error occurred during transfer.
E_VISA_IN_PROGRESS	0x80040039	-2147221447	An operation is in progress.
E_VISA_INV_SETUP	0x8004003a	-2147221446	Some implementation-specific configuration file is corrupt or does not exist.
E_VISA_QUEUE_ERROR	0x8004003b	-2147221445	There was an event queue error.
E_VISA_ALLOC	0x8004003c	-2147221444	The system could not allocate the buffer(s) of the specified size because of insufficient system resources.
E_VISA_INV_MASK	0x8004003d	-2147221443	The system cannot set the buffer for the given mask.
E_VISA_IO	0x8004003e	-2147221442	Could not perform read/write operation because of IO error.
E_VISA_INV_FMT	0x8004003f	-2147221441	The format of the data is invalid.
E_VISA_NSUP_FMT	0x80040041	-2147221439	The format of the data is not supported.
E_VISA_LINE_IN_USE	0x80040042	-2147221438	One of the specified lines (trigSrc or trigDest) is currently in use.
E_VISA_NSUP_MODE	0x80040046	-2147221434	The specified mode is not supported by this VISA implementation.
E_VISA_SRQ_NOCCURRED	0x8004004a	-2147221430	Service request has not been received for the session.
E_VISA_INV_SPACE	0x8004004e	-2147221426	Invalid address space specified.
E_VISA_INV_OFFSET	0x80040051	-2147221423	Invalid offset specified.
E_VISA_INV_WIDTH	0x80040052	-2147221422	Specified width is not supported by this hardware.
E_VISA_NSUP_OFFSET	0x80040054	-2147221420	Specified offset is not accessible from this hardware.
E_VISA_NSUP_VAR_WIDTH	0x80040055	-2147221419	Cannot support source and destination widths that are different.
E_VISA_WINDOW_NMAPPED	0x80040057	-2147221417	The specified session is not currently mapped.
E_VISA_RESP_PENDING	0x80040059	-2147221415	A previous response is still pending, causing a multiple query error.
E_VISA_NLISTENERS	0x8004005f	-2147221409	No listeners condition is detected (both NRFD and NDAC are deasserted).
E_VISA_NCIC	0x80040060	-2147221408	The interface associated with the given vi is not currently the controller in charge.
E_VISA_NSYS_CNTLR	0x80040061	-2147221407	The interface associated with this session is not the system controller.
E_VISA_NSUP_OPER	0x80040067	-2147221401	The given resource does not support this operation.
E_VISA_INTR_PENDING	0x80040068	-2147221400	An interrupt is still pending from a previous call.
E_VISA_ASRL_PARITY	0x8004006a	-2147221398	A parity error occurred during transfer.
E_VISA_ASRL_FRAMING	0x8004006b	-2147221397	A framing error occurred during transfer.

E_VISA_ASRL_OVERRUN	0x8004006c	-2147221396	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
E_VISA_TRIG_NMAPPED	0x8004006e	-2147221394	The path from trigSrc to trigDest is not currently mapped.
E_VISA_NSUP_ALIGN_OFFSET	0x80040070	-2147221392	The specified offset is not properly aligned for the access width of the operation.
E_VISA_USER_BUF	0x80040071	-2147221391	A specified user buffer is not valid or cannot be accessed for the required size.
E_VISA_RSRC_BUSY	0x80040072	-2147221390	The resource is valid, but VISA cannot currently access it.
E_VISA_NSUP_WIDTH	0x80040076	-2147221386	Specified width is not supported by this hardware.
E_VISA_INV_PARAMETER	0x80040078	-2147221384	The value of some parameter-which parameter is not known-is invalid.
E_VISA_INV_PROT	0x80040079	-2147221383	The protocol specified is invalid.
E_VISA_INV_SIZE	0x8004007b	-2147221381	Invalid size of window specified.
E_VISA_WINDOW_MAPPED	0x80040080	-2147221376	The specified session already contains a mapped window.
E_VISA_NIMPL_OPER	0x80040081	-2147221375	The given operation is not implemented.
E_VISA_INV_LENGTH	0x80040083	-2147221373	Invalid length specified.
E_VISA_INV_MODE	0x80040091	-2147221359	The value specified by the mode parameter is invalid.
E_VISA_SESN_NLOCKED	0x8004009c	-2147221348	The current session did not have any lock on the resource.
E_VISA_MEM_NSHARED	0x8004009d	-2147221347	The device does not export any memory.
E_VISA_LIBRARY_NFOUND	0x8004009e	-2147221346	A code library required by VISA could not be located or loaded.
E_VISA_NSUP_INTR	0x8004009f	-2147221345	The interface cannot generate an interrupt on the requested level or with the requested statusID value.
E_VISA_INV_LINE	0x800400a0	-2147221344	The value specified by the line parameter is invalid.
E_VISA_FILE_ACCESS	0x800400a1	-2147221343	An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights.
E_VISA_FILE_IO	0x800400a2	-2147221342	An error occurred while accessing the specified file.
E_VISA_NSUP_LINE	0x800400a3	-2147221341	One of the specified lines (trigSrc or trigDest) is not supported by this VISA implementation.
E_VISA_NSUP_MECH	0x800400a4	-2147221340	The specified mechanism is not supported for the given event type.
E_VISA_INTF_NUM_NCONFIG	0x800400a5	-2147221339	The interface type is valid but the specified interface number is not configured.
E_VISA_CONN_LOST	0x800400a6	-2147221338	The I/O connection for the given session has been lost.

E_VISA_MACHINE_NAVAIL	0x800400a7	-2147221337	The remote machine does not exist or is not accepting any connections.
E_VISA_NPERMISSION	0x800400a8	-2147221336	Access to the resource or remote machine is denied. This is due to lack of sufficient privileges for the current user or machine.