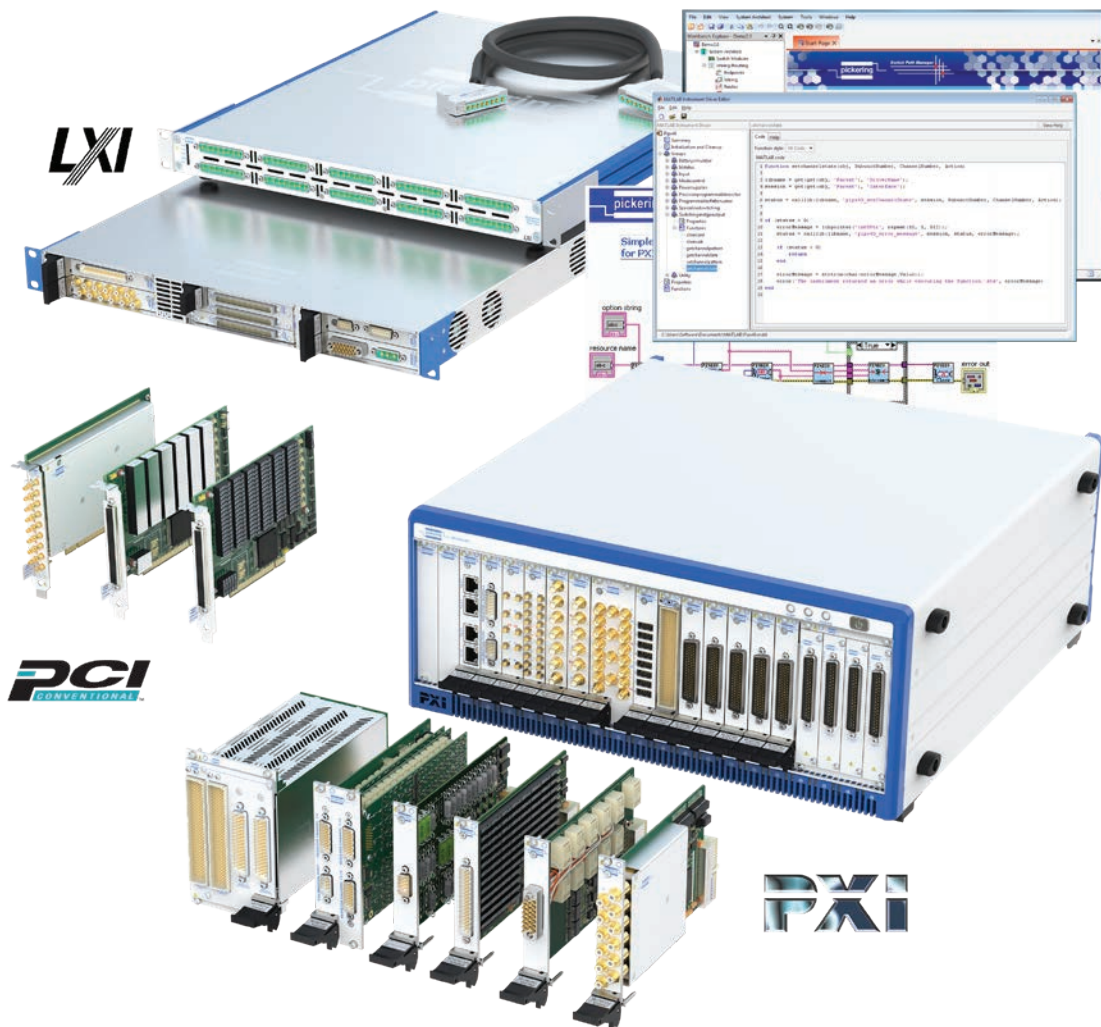


Software User Manual

- Driver Selection, Installation & Test
- Using a PCI/PXI Card or LXI Device
- The IVI Driver - pi40iv
- Programming Environments
- Support Software



© COPYRIGHT (2023) PICKERING INTERFACES. ALL RIGHTS RESERVED.

No part of this publication may be reproduced, transmitted, transcribed, translated or stored in any form, or by any means without the written permission of Pickering Interfaces.

Technical details contained within this publication are subject to change without notice.



ISO 9001
Reg No. FM38792

TECHNICAL SUPPORT

For Technical Support please contact Pickering Interfaces either by phone, the website or via e-mail.

WARRANTY

All products manufactured by Pickering Interfaces are warranted against defective materials and workmanship for a period of three years, excluding programmable power supplies, from the date of delivery to the original purchaser. Any product found to be defective within this period will, at the discretion of Pickering Interfaces be repaired or replaced.

Products serviced and repaired outside of the warranty period are warranted for ninety days.

Extended warranty and service are available. Please contact Pickering Interfaces by phone, the website or via e-mail.

ENVIRONMENTAL POLICY

Pickering Interfaces operates under an environmental management system similar to ISO 14001.

Pickering Interfaces strives to fulfil all relevant environmental laws and regulations and reduce wastes and releases to the environment. Pickering Interfaces aims to design and operate products in a way that protects the environment and the health and safety of its employees, customers and the public. Pickering Interfaces endeavours to develop and manufacture products that can be produced, distributed, used and recycled, or disposed of, in a safe and environmentally friendly manner.

Worldwide Technical Support and Product Information

pickeringtest.com

Pickering Interfaces Headquarters

Stephenson Road Clacton-on-Sea CO15 4NL United Kingdom

Tel: +44 (0)1255-687900

E-Mail: sales@pickeringtest.com

USA

Tel: (West) +1 541 471 0700

Tel: (East) +1 781 897 1710

E-mail: ussales@pickeringtest.com

France

Tel +33 9 72 58 77 00

E-mail frsales@pickeringtest.com

Germany

Tel: +49 89 125 953 160

E-mail: desales@pickeringtest.com

Sweden

Tel: +46 340-69 06 69

E-mail: ndsales@pickeringtest.com

Czech Republic

Tel: +420 558 987 613

E-mail: desales@pickeringtest.com

China

Tel: +86 4008 799 765

E-mail: chinasales@pickeringtest.com

Trademarks:

LabVIEW, LabWindows/CVI, Switch Executive, LabVIEW RT, VeriStand, TestStand, Measurement and Automation Explorer (MAX) are all trademarks of National Instrument Corporation.

Windows and Visual Studio are trademarks of Microsoft Corporation.

MATLAB is a trademark of The Mathworks Inc.

Switch Path Manager is a trademark of Pickering Interfaces Ltd.

Note:

This is a generic manual describing the driver options available, their installation and general principles of use. For information about a particular card refer to the manual for that card.

THIS PAGE INTENTIONALLY BLANK

CONTENTS

| | |
|---|------------|
| Section 1 - DRIVER SELECTION INSTALLATION & TEST | 1.1 |
| 1.1 Choosing a Driver..... | 1.1 |
| 1.2 Installation..... | 1.4 |
| 1.3 Testing & Diagnostics | 1.5 |
| Section 2 - USING A PCI/PXI CARD OR LXI DEVICE | 2.1 |
| 2.1 Installing a Card..... | 2.1 |
| 2.2 Card Address Aliasing | 2.3 |
| 2.3 Card Architecture | 2.8 |
| 2.4 SubUnit Types..... | 2.9 |
| 2.5 Using an LXI Product | 2.23 |
| Section 3 - IVI DRIVER PI40IV..... | 3.1 |
| 3.1 The IVI Switch Class..... | 3.1 |
| 3.2 The IVI Switch Paradigm..... | 3.1 |
| 3.3 Using the IVI C Driver..... | 3.4 |
| 3.4 Operating a Switch | 3.5 |
| 3.5 Interaction With NI MAX..... | 3.6 |
| 3.6 Example Adding PI Module Using MAX | 3.8 |
| 3.7 Using the IVI Driver in NI Switch Executive | 3.9 |
| 3.8 Using the IVI Driver in NI TestStand | 3.12 |
| Section 4 - PROGRAMMING ENVIRONMENTS | 4.1 |
| 4.1 Pickering Software Environments | 4.1 |
| 4.2 NI Software Environments..... | 4.3 |
| 4.3 Microsoft Visual Studio..... | 4.8 |
| 4.4 Pickering .NET Driver..... | 4.10 |
| 4.5 Python | 4.22 |
| 4.6 Linux..... | 4.24 |
| Section 5 - SUPPORT SOFTWARE..... | 5.1 |
| 5.1 General Soft Front Panel | 5.1 |
| 5.2 IVI Wizard | 5.3 |
| 5.3 Pickering Diagnostic Utility | 5.6 |
| Section 6 - USEFUL INFORMATION | 6.1 |
| 6.1 IVI Interchangeability | 6.1 |
| 6.2 Channel Names..... | 6.3 |
| Section 7 - SHARED MEMORY | 7.1 |
| Section 8 - PROTECTION RESET UTILITY | 8.1 |

THIS PAGE INTENTIONALLY BLANK

SECTION 1 - DRIVER SELECTION, INSTALLATION AND TEST

1.1 CHOOSING A DRIVER

Pickering Interfaces provide a selection of different programming interfaces aimed at different environments, each comprises a low level interface to the computer hardware layer and a user programming interface.

It is important to understand the basic characteristics of each driver, the capability of the programming environment and any limitations of the target environment before selecting a driver.

Three basic driver types are offered:

| Driver | Description |
|-------------|--|
| Direct IO | Does not require any additional software installation, no additional licenses. |
| VXIPnP/VISA | Requires a VISA installation which may require a license, not available for Pickering LXI devices. |
| IVI | Provides a degree of interchangeability, requires VISA and IVI installations which may require licenses. |

Each core driver has a 'C' style interface and additional wrappers to interface the core driver to common programming environments such as LabVIEW and LabWindows/CVI.

The VXIPnP/VISA system provides a number of useful features such as a mechanism to define device aliases thus permitting device location changes without recompilation of target software.

IVI provides a more standardized interface and can offer interchangeability. See a later section of this manual for an explanation of how to achieve interchangeability.

The choice of driver depends on a number of factors, primarily:

- Product type
- Programming environment
- Operating system
- Preference

1.1.1 Product Type

Not all products have a full choice of driver types available.

PC/PXI - Direct I/O Driver, VXIPnP/VISA Driver, IVI Driver
 LXI - Direct I/O Driver, IVI Driver

1.1.2 Programming Environment

All drivers may be used in a wide variety of programming environments, however some programming environments restrict the choice of drivers, for example, National Instruments Switch Executive is designed to use only IVI drivers.

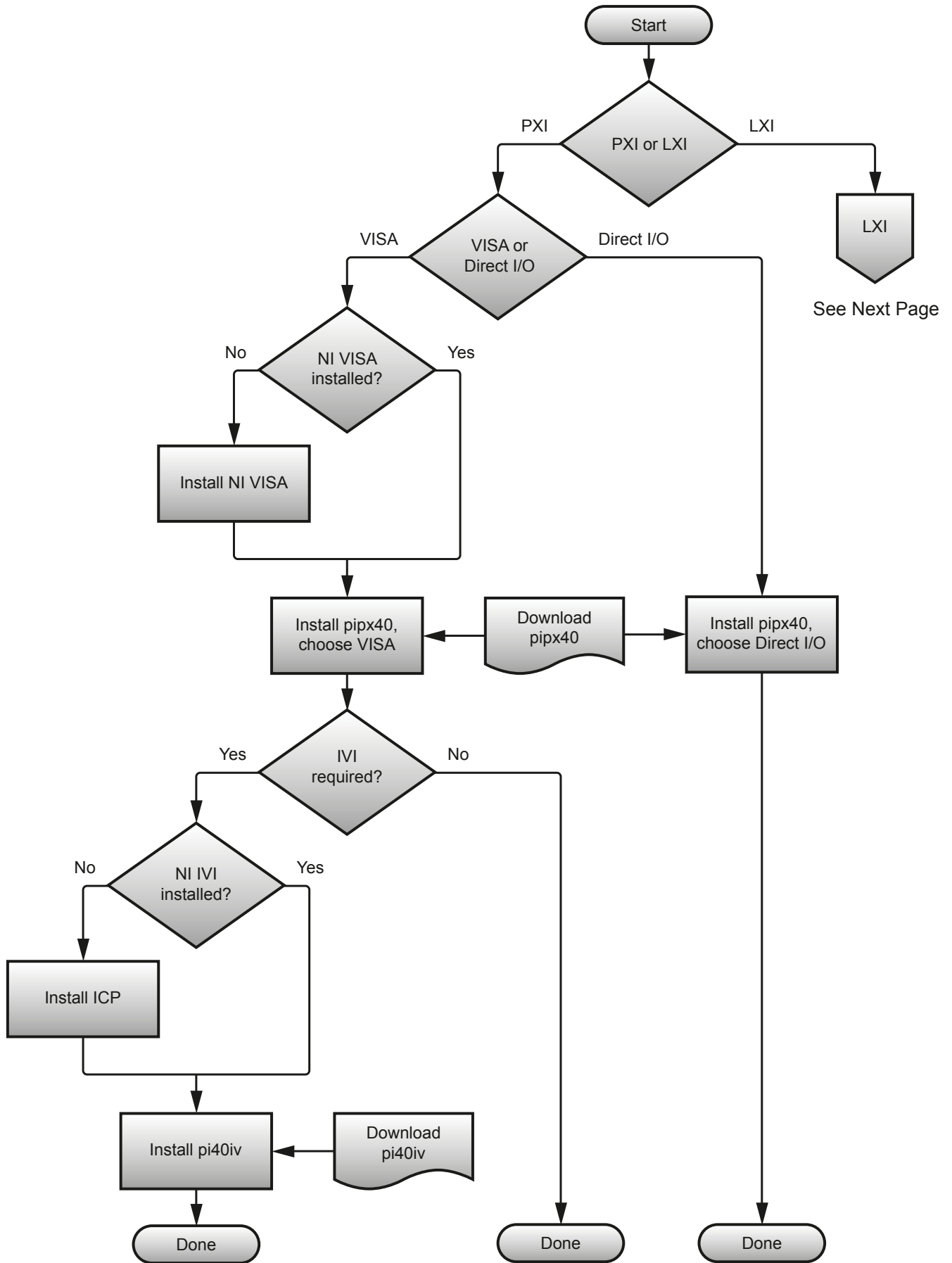
1.1.3 Operating System

Microsoft Windows supports all driver types, however other operating systems may limit the choice. There is limited availability of a VISA subsystem on most other operating systems that prevents the use of VXIPnP/VISA or IVI drivers on these systems.

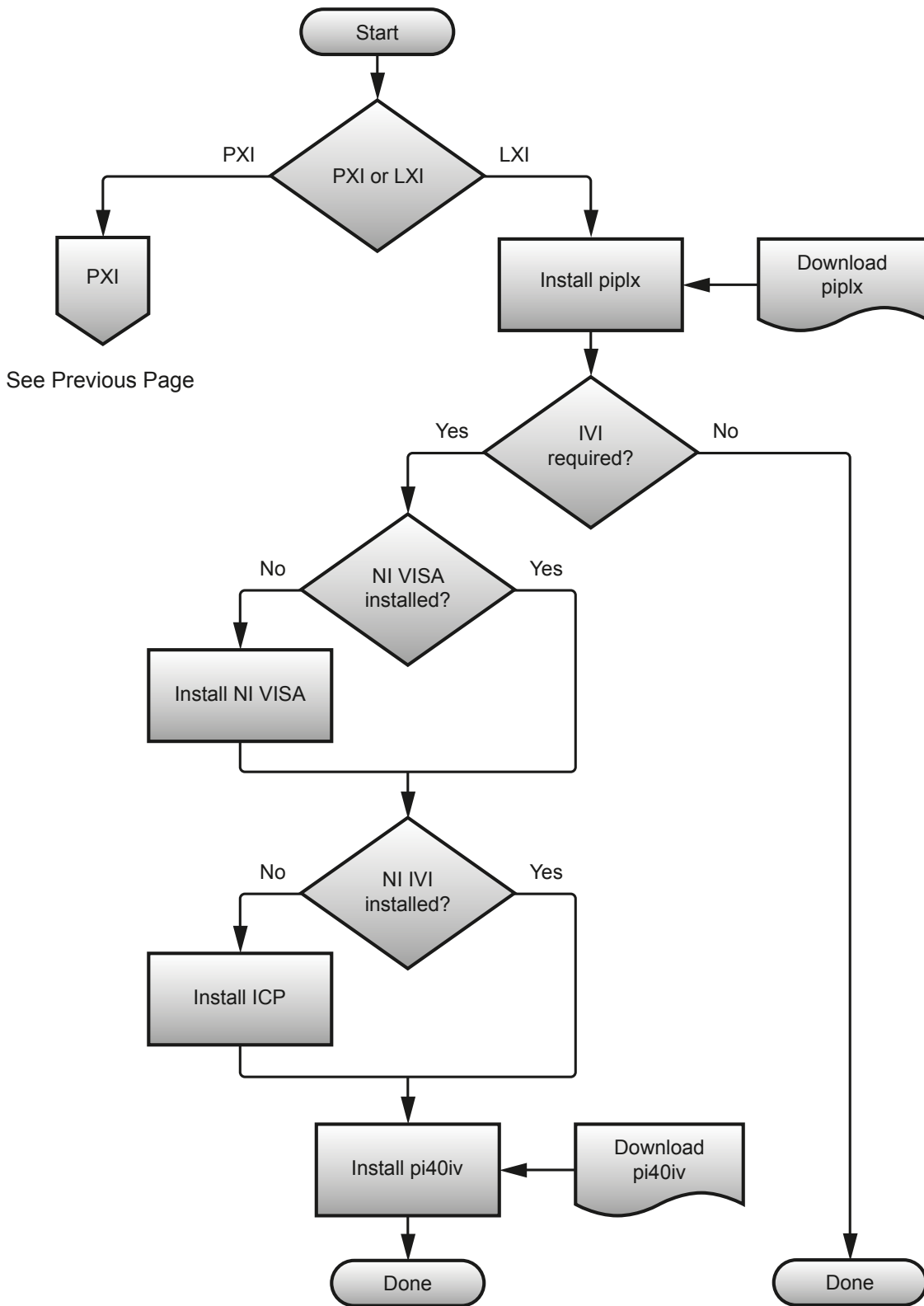
1.1.4 Preference

Sometimes company policy will dictate the use of a particular driver type, or familiarity with a driver type may lead a programmer to select a particular driver.

The diagrams on the following pages may help in deciding which drivers are required.



Driver Selection Flowchart - PXI



Driver Selection Flowchart - LXI

1.2 INSTALLATION

1.2.1 PCI/PXI Products

Decide whether to use Direct I/O or VXIPnP/VISA or IVI.

The Direct I/O driver and the VXIPnP/VISA driver are contained in the same installer package, the user is offered a choice during the installation procedure.

Direct I/O:

Locate and install the Pickering PXI driver package.

The installation package can be found on the DVD delivered with the hardware or the latest version from the Pickering download site at:

http://downloads.pickeringtest.info/downloads/drivers/PXI_Drivers/

During installation choose the Direct I/O driver option.

VXIPnP/VISA:

Ensure a suitable VISA system is installed.

Locate and install the Pickering PXI driver package as above but select the VXIPnP/VISA option during the installation process (please note that in this case the Direct I/O driver is also installed and may also be used, but the system sets the cards to be visible to VISA).

1.2.2 LXI Products

Direct I/O:

A separate driver, ClientBridge, is provided for LXI products and it may be obtained from the delivered DVD or from the Pickering Interfaces website:

<http://www.downloads.pickeringtest.info/downloads/drivers/Sys60/>

1.2.3 Both PCI/PXI and LXI

IVI:

If using the IVI driver with PCI/PXI products first install the VXIPnP driver as detailed above.

If using the IVI driver with LXI first install the ClientBridge driver as detailed above.

In either case ensure both the VISA system and IVI Compliance Package (ICP) are installed.

Locate, obtain and install the Pickering IVI driver either from the delivered DVD or the latest version from:

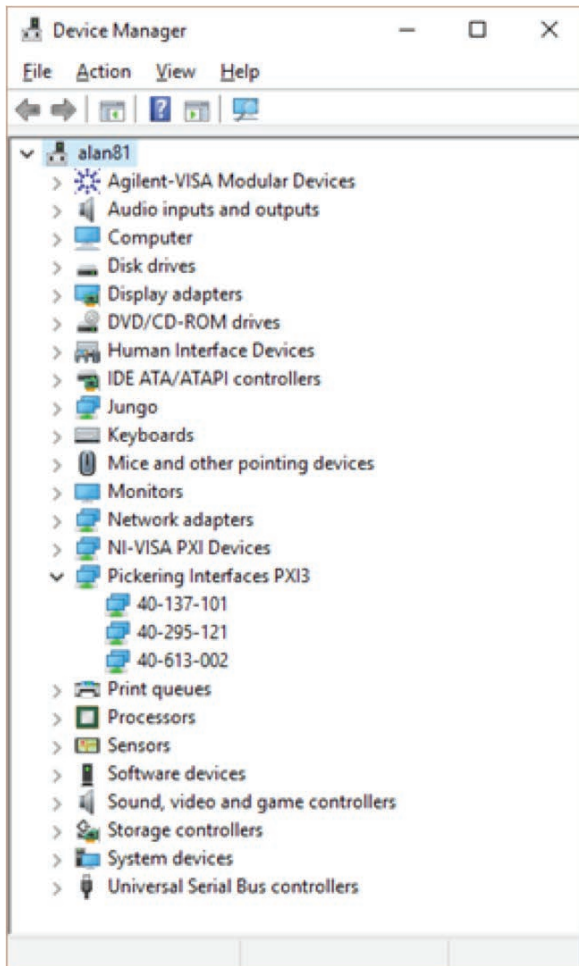
<http://downloads.pickeringtest.info/downloads/drivers/IVI/>

In all cases wrappers for interface to common programming environments are installed together with the driver.

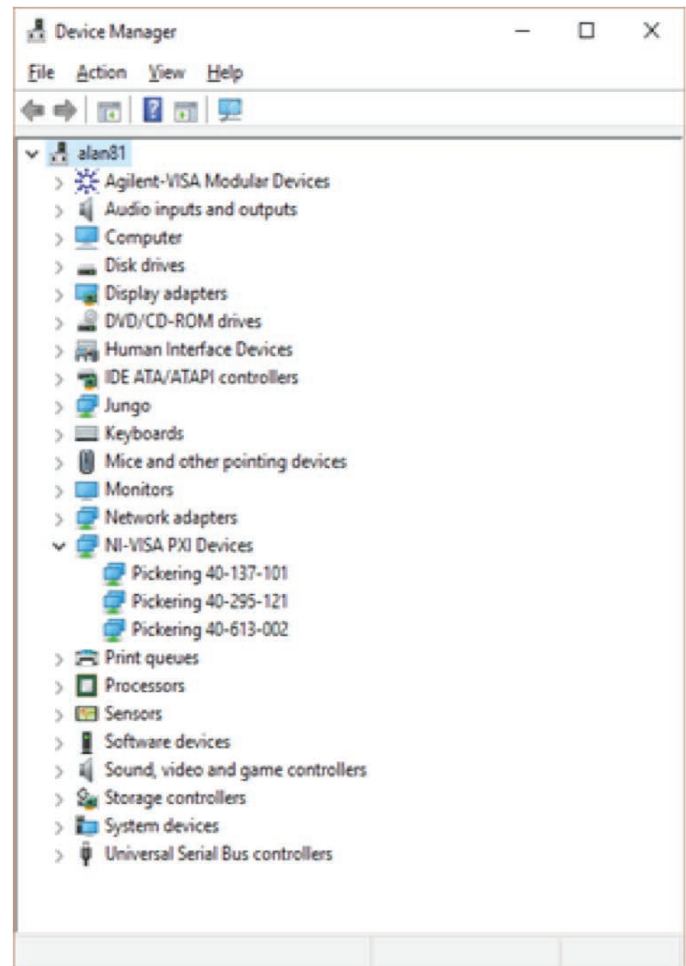
1.3 TESTING AND DIAGNOSTICS

1.3.1 Check the PCI/PXI Drivers are Correctly Installed

The first step in checking the installation is to verify that the fitted cards are visible and correctly located in Windows Device Manager:



**Cards Installed
as Direct I/O**

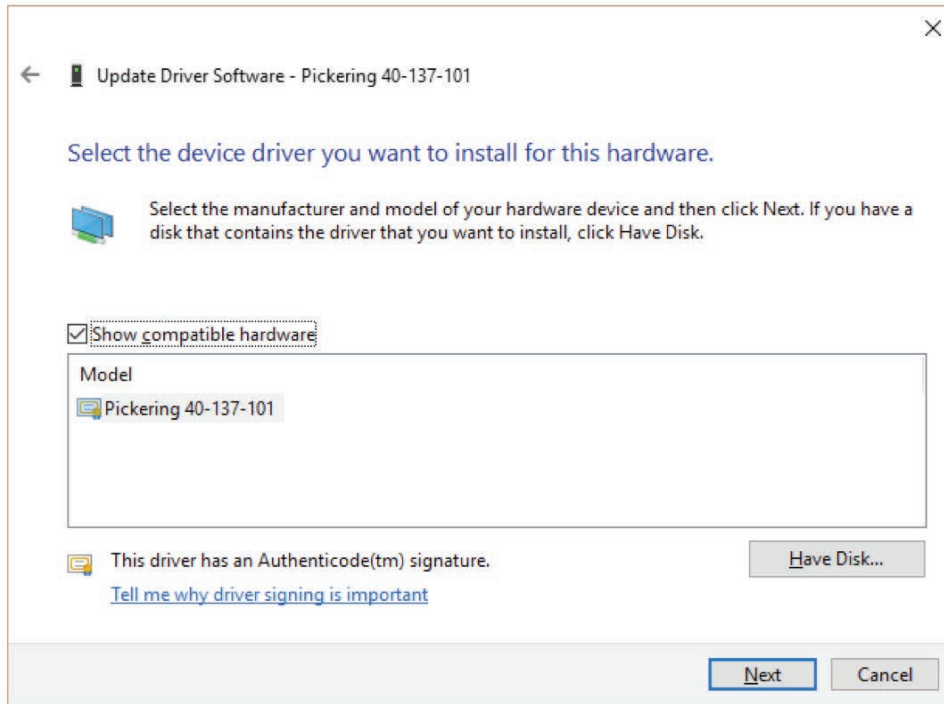


**Cards Installed
as VXIPnP/VISA**

If the cards are not correctly displayed, try re-booting the system to see if the cards are then correctly identified.

If still not correctly displayed after a re-boot, try uninstalling then re-installing the driver.

If the cards continue to be incorrectly displayed, right click on the entry which relates to the card and select 'Update Driver'. Select the options required to allow manual selection of the driver until a screen is reached that lists the possible drivers for the card. An entry starting with the with 'Pickering' relates to the VXIPnP driver, or with just the card model number to the Direct I/O driver. Select the appropriate driver and complete the action.



Driver Update Screen

If all the above fails, locate and execute the Pickering diagnostic utility, usually to be found from the Windows 'Start' menu at 'All Apps' – 'Pickering Interfaces Ltd' - 'Diagnostic Utility'. This utility generates a textual report which contains system information that can assist diagnosis of the problems. The diagnostic report is described in more detail in another section of this manual.

If it is not clear what is causing the problem, mail the report to the Pickering support team at: support@pickeringtest.com with any additional information that may be relevant.

1.3.2 Checking Cards are Functional

The General Soft Front Panel (GSFP) is a graphical tool and is described in Section 5. It is a quick means to check if the cards are being found and are working.

Should the cards not be found by the GSFP, then some investigation is required.

For PCI/PXI cards the diagnostic report will open the cards with the driver or drivers that are installed and verify their status.

SECTION 2 - USING A PCI/PXI CARD OR LXI PRODUCT

NOTE: This section confines itself to the 2 basic drivers for PCI/PXI products, PILPXI and PIPX40. Later sections deal with other drivers.

2.1 INITIALIZING A CARD

The first operation required in any application is to obtain a session handle for the card to be controlled.

PILPXI:

The PILPXI driver provides three means of opening cards:

```

/* =====
 * Function:   Locate and open all installed Pickering cards.
 * Returns:   the number of Pickering cards located and opened.
 */
DWORD _stdcall PIL_OpenCards(void) MYATTR;

/* =====
 * Function:   open a card, specified by logical bus and slot location
 * Arguments:  Bus = logical bus location of card to open
 *             Slot = logical slot location of card to open
 *             CardNum = pointer to variable to receive the Pickering card
 *                   reference of target (unity-based)
 * Returns:   0 = success; non-zero = error code
 */
DWORD _stdcall PIL_OpenSpecifiedCard(DWORD Bus, DWORD Slot, DWORD *CardNum) MYATTR;

/* =====
 * Function: Initialize function using Alisases

 * Arguments:  Alias           = Alias defined in a resource management file
 *             Storage         = Path to the stored resource management file
 *             Access          = Unused
 *             CardNum         = Pickering card reference of target (unity-based)
 * Returns:   0 = success; non-zero = error code
 */
DWORD _stdcall PIL_Init(CHAR *Alias, CHAR *Storage, DWORD Access, DWORD* CardNum)
MYATTR;

```

The first opens all available cards and returns the number of cards opened, cards are then accessed using a 'CardNum' value between 1 and the number of cards opened.

The second opens a single card at a specified location leaving all other cards unopened.

The third uses an alias for the card location, see next section for more detail.

Once a control session is no longer required it should be terminated using the appropriate method:

PIL_CloseCards

PIL_CloseSpecifiedCard

PIPX40:

```
//*****
// Initialize
// Establishes communications with the instrument.
//   rsrcName (in)   Instrument description VXI/LA
//   id_query  (in)   VI_TRUE = Perform in-system verification
//                   VI_FALSE = Do not perform in-system verification
//   reset_instr (in) VI_TRUE = Perform reset operation
//                   VI_FALSE = Do not perform reset operation
//   vi (out)        Instrument handle
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_init (ViRsrc rsrcName,
                              ViBoolean id_query,
                              ViBoolean reset_instr,
                              ViPSession vi);
```

Currently the two ViBoolean parameters are ignored and have no effect.

Once a control session is no longer required it should be terminated using the appropriate method:

pipx40_close

pipx40 is a VXIPnP compliant driver using the VISA subsystem for hardware control. The VISA system provides some useful features among which is a means of aliasing a card address.

Important note:

Currently all methods to open a card will reset the card, setting all switches to their default, non-energized, states.

This is an inevitable consequence of the design of the software/hardware system.

When a session is closed, the switches are left in their current states.

2.2 CARD ADDRESS ALIASING

An issue that many users have encountered is that when a system controller is upgraded, or a system replicated, or additional PXI chassis are added to a system, the PCI/PXI addresses of existing cards may change.

If an application has been written with embedded card addresses such a change will cause the application to fail or to open incorrect cards.

The address of a card is defined in the early stages of the system boot-up sequence when the PCI bus is enumerated and resource allocations are made. It is not possible to modify these addresses later. Any significant change to the system can cause the addresses to be differently allocated which would render any card addresses embedded in an application to be incorrect.

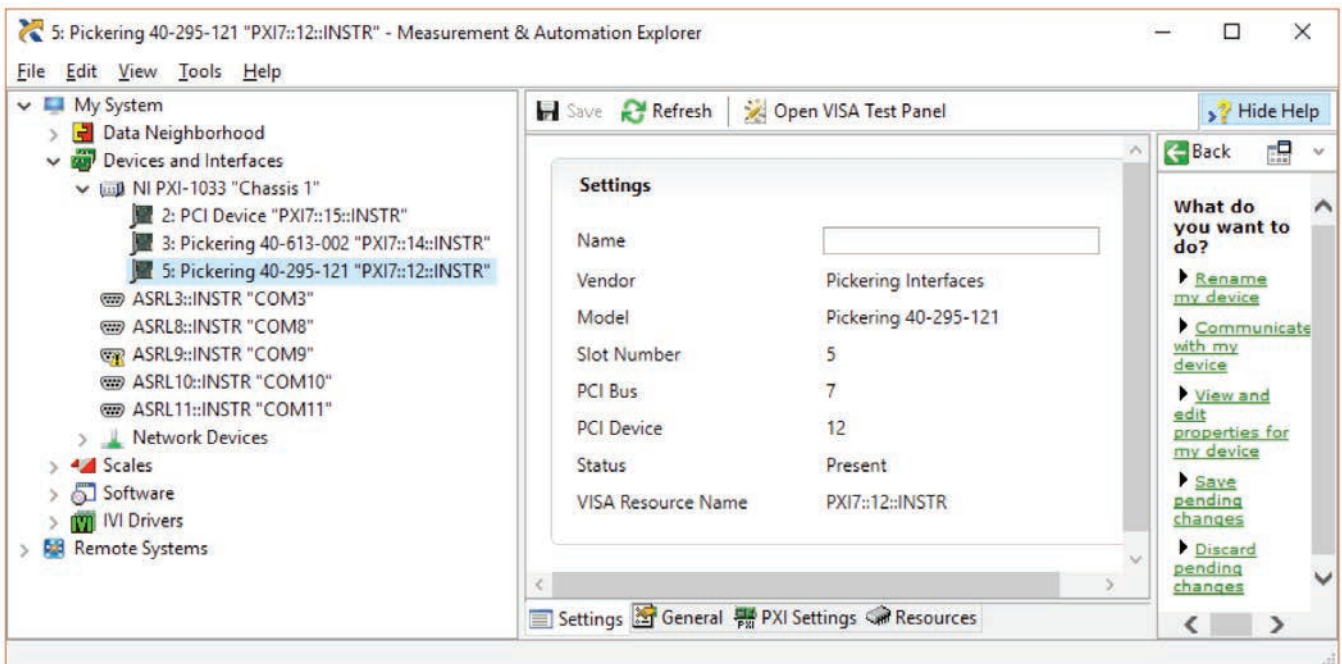
One possible solution would be for the application to interrogate the PCI/PXI bus system, check each card found and establish the current locations of cards by model and/or serial number.

A better solution to this problem is to avoid hard-coding physical addresses into applications and to instead use an alias that is stored in a system data file.

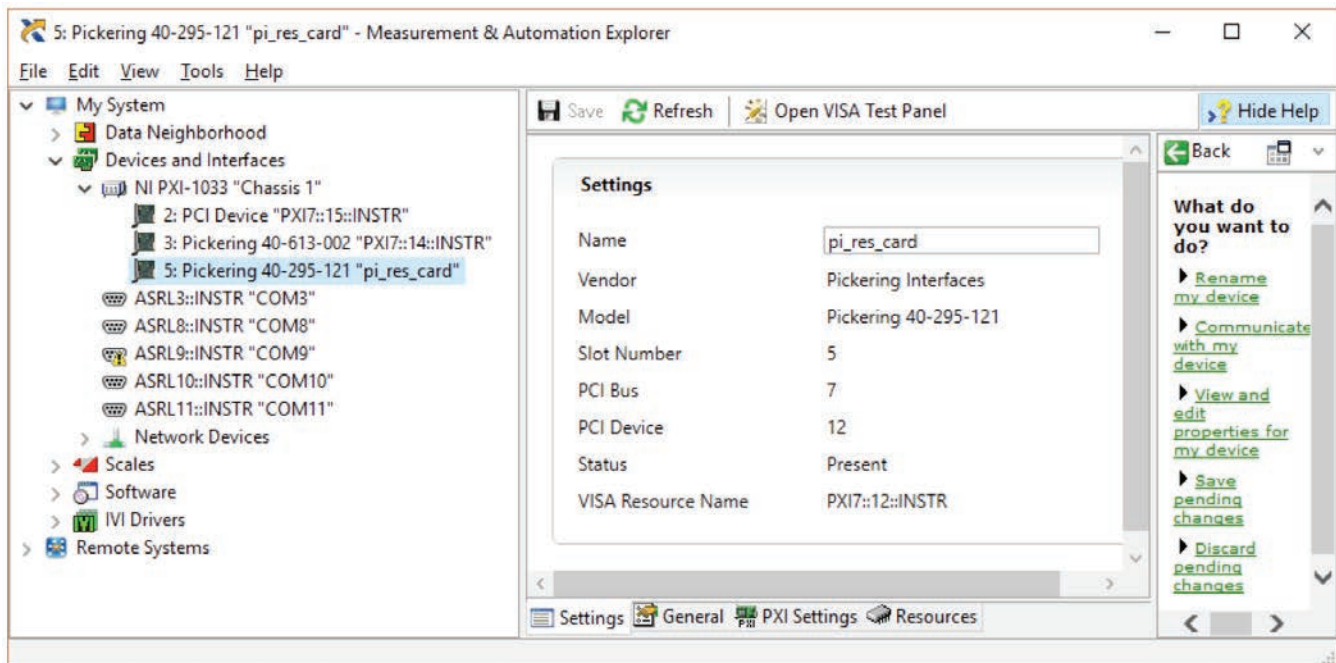
2.2.1 PIPX40 Aliases

The VISA system provides a means to define an alias for a card. This is stored in the VISA system configuration file but is commonly viewed and edited using a software tool such as NI MAX.

Here the PCI/PXI devices are listed:



When one is selected in the device tree, the window to the right offers the possibility of adding a 'Name' to the card:



Now the application may be written to establish a control session using the alias instead of a hard-coded VISA resource string:

```
int main()
{
    ViSession vi;
    ViStatus err;
    ViChar id[100];
    ViUInt16 bus, device;

    err = pipx40_init("pi_res_card", VI_FALSE, VI_FALSE, &vi);

    err = pipx40_getCardId(vi, id);
    viGetAttribute(vi, VI_ATTR_PXI_BUS_NUM, &bus);
    viGetAttribute(vi, VI_ATTR_PXI_DEV_NUM, &device);

    pipx40_close(vi);

    printf("Card id: %s At location %d:%d\n", (char*)id, bus, device);

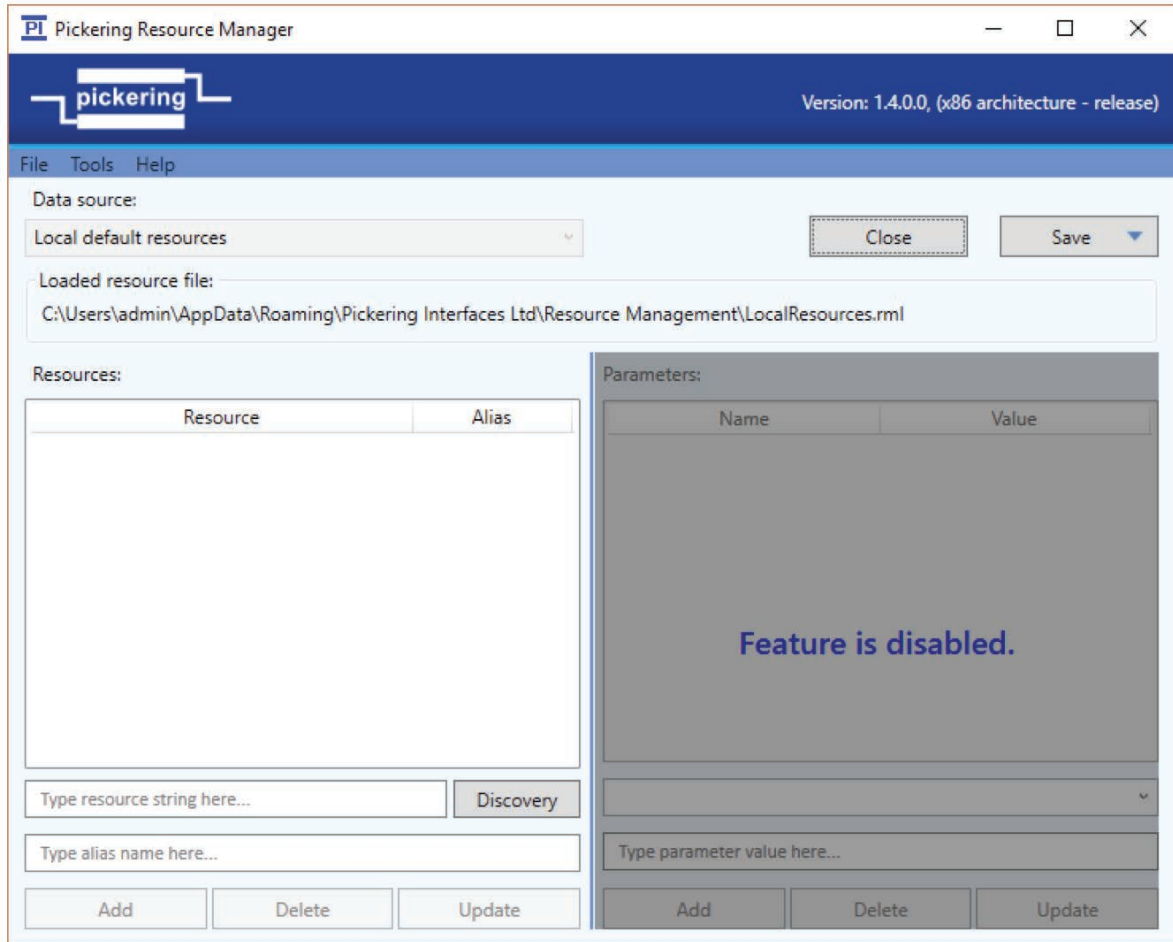
    return 0;
}
```


2.2.2 PILPXI Aliases

A similar alias feature has been recently added to Pickering non-VISA driver PILPXI.

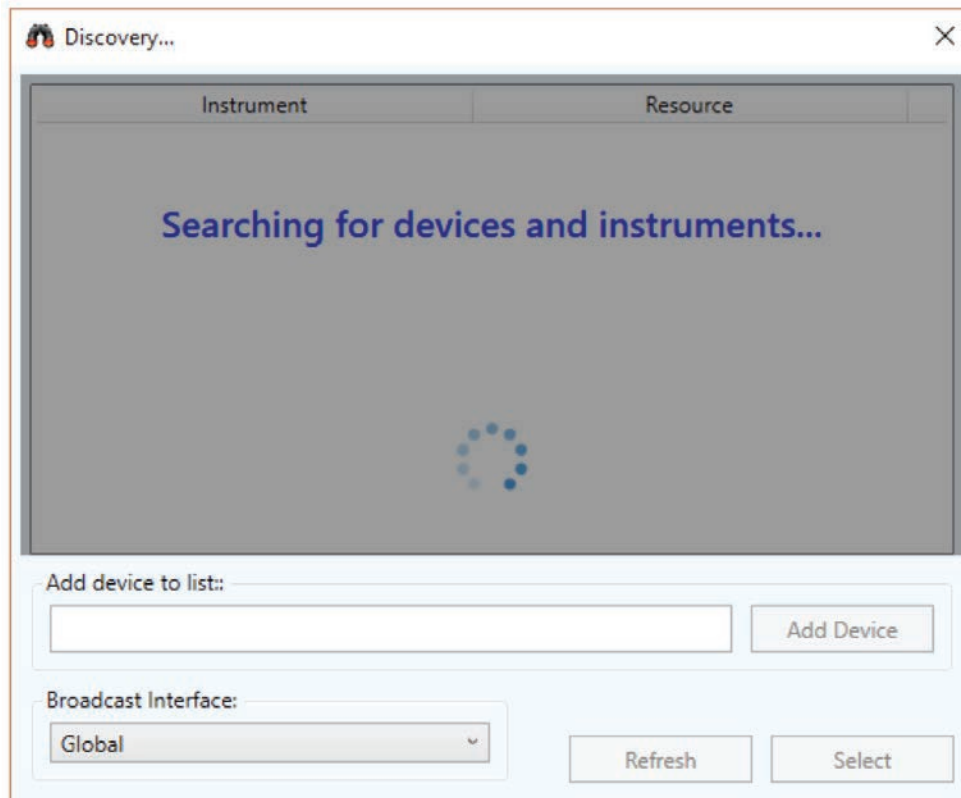
To view and edit aliases execute the Pickering Resource Manager application.

Select the Data Source 'Local default resources':

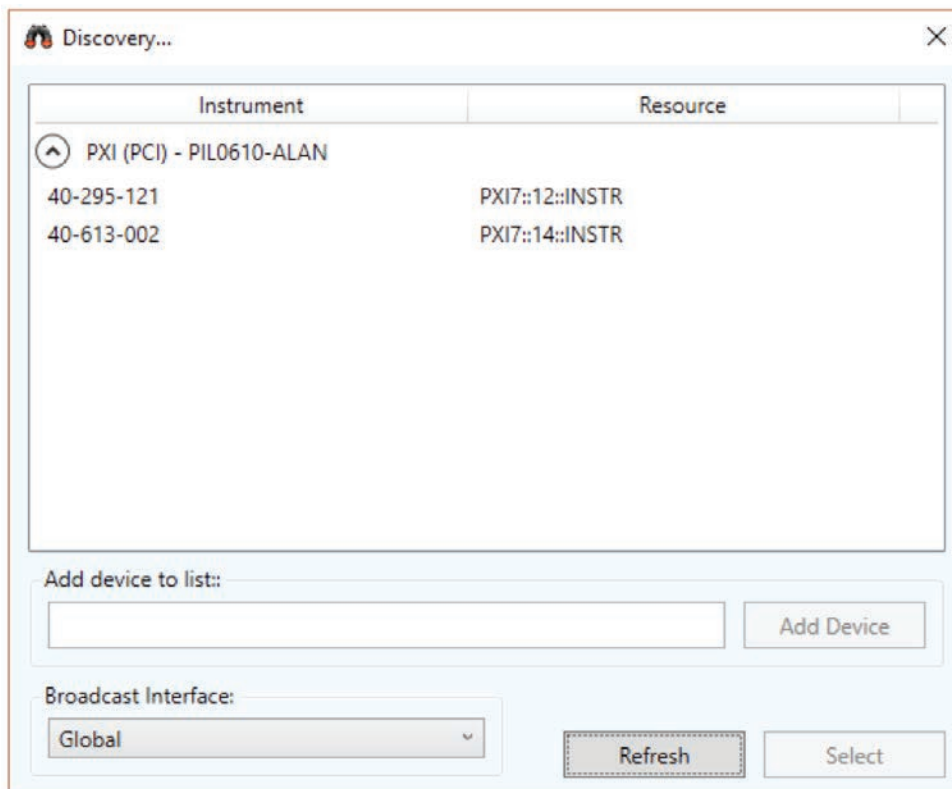


In this case there are no defined aliases.

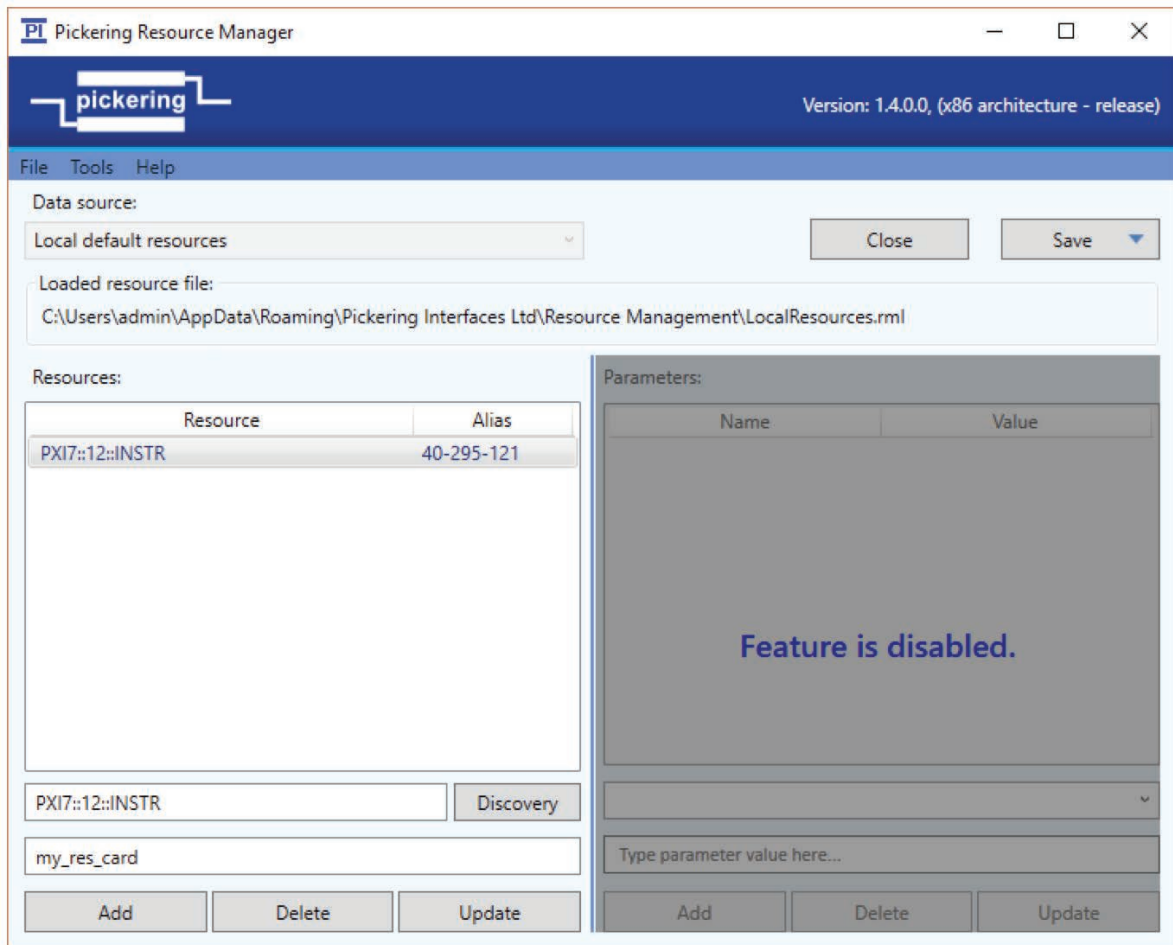
Click on 'Discovery' and then 'Refresh' to locate any Pickering devices available on either PCI/PXI or LXI systems:



When discovery is complete the window will be populated with available devices:



Select the device to be aliased and click on 'Select', the device will then appear in the main application window.



A default alias - the card model number - is entered automatically. This may not be convenient, especially if there are multiple cards of the same type, so the user may edit the alias as they wish.

Select the lower edit box and type in a new alias, then click on 'Update'.

Finally click on 'Save' to update the system data file with the new alias.

Now an application may be coded to access the card by its alias:

```
int main()
{
    DWORD err, card, bus, device;
    CHAR id[100];

    err = PIL_Init("my_res_card", NULL, NULL, &card);

    err = PIL_CardId(card, id);
    err = PIL_CardLoc(card, &bus, &device);
    PIL_CloseSpecifiedCard(card);

    printf("Card id: %s At location %d:%d\n", (char*)id, bus, device);

    return 0;
}
```

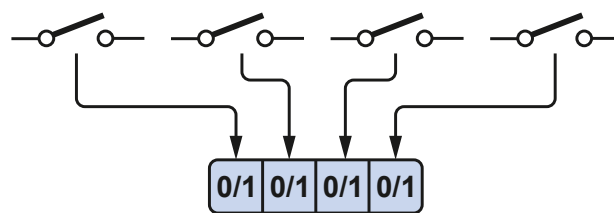
Should the device address change for any reason, then the alias may be edited in the Pickering Resource Manager so that the application would need no change.

2.3 CARD ARCHITECTURE

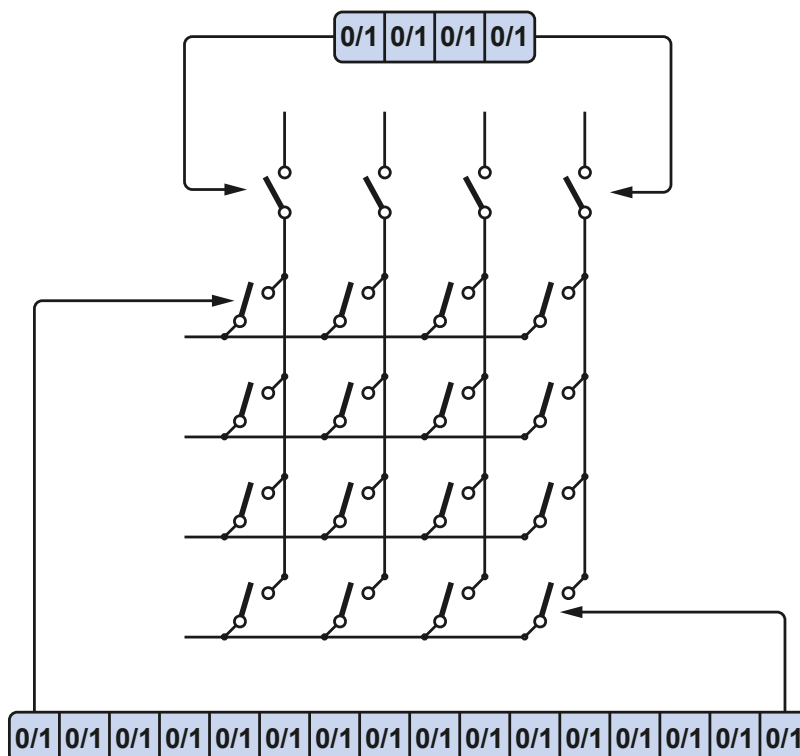
Before attempting to control switches on a Pickering card it is important to understand its general architecture.

A Pickering card will present switches in groups, which we call SubUnits, each card will contain one or more SubUnits with each SubUnit containing a collection of one or more switches, or other binary object types. Each SubUnit contains a set of objects of similar type or purpose with each object being represented by a single binary bit where the state of that bit represents the state of the hardware object, normally logic 1 for ON and logic 0 for OFF. These bits are contained in 32-bit unsigned integer containers, a DWORD type in PILPXI and a ViUInt32 type in PIPX40.

For example a simple card containing 4 switches will present these switches as a single SubUnit containing 4 binary bits where the state of each bit represents the on/off state of each switch. A switch is controlled by altering the state of the bit representing that switch:



In more complex cases, the card will present multiple SubUnits. For example a 4x4 matrix with isolation switches on the X axis will present 2 SubUnits: one containing 16 bits to represent the matrix, and another SubUnit containing 4 bits to represent the isolation switches:



Refer to the manual for the card being programmed to ascertain the structure of the card as presented via the driver.

2.4 SUBUNIT TYPES

A number of different SubUnit types are defined in the base drivers and each type has specific methods of operation supported by driver functions:

| | |
|--------------|---|
| TYPE_SW | = 1, /* Uncommitted switches */ |
| TYPE_MUX | = 2, /* Relay multiplexer (single-channel only) */ |
| TYPE_MUXM | = 3, /* Relay multiplexer (multi-channel capable) */ |
| TYPE_MAT | = 4, /* Standard matrix */ |
| TYPE_MATR | = 5, /* RF matrix */ |
| TYPE_DIG | = 6, /* Digital outputs (or inputs)*/ |
| TYPE_RES | = 7, /* Programmable Resistor */ |
| TYPE_ATTEN | = 8, /* Programmable Attenuator */ |
| TYPE_PSUDC | = 9, /* Power supply - DC */ |
| TYPE_BATT | = 10, /* Battery simulator */ |
| TYPE_VSOURCE | = 11, /* Programmable voltage source */ |
| TYPE_MATP | = 12, /* Matrix with restricted operating modes */ |
| TYPE_MUXMS | = 13, /* Relay multiplexer (MUXM hardware emulated as MUX) */ |
| TYPE_FI | = 14, /* Sub-unit with restricted operation */ |

The means of controlling these types is discussed below relating to the 2 core drivers, PILPXI and PIPX40. Use of the IVI and LXI drivers are discussed in later sections.

The quantity and types of SubUnits on a card may be queried using functions provided in the drivers.

```

//*****
// Get sub-unit counts
// Obtains counts of the number of input and output sub-units on the card.
//   vi (in)      Instrument handle
//   inSubs (out) The number of input sub-units
//   outSubs (out) The number of output sub-units
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_getSubCounts (ViSession vi,
                                       ViPUInt32 inSubs,
                                       ViPUInt32 outSubs);

/* =====
* Function:    get Pickering card input and output sub-unit counts
* Arguments:  CardNum = Pickering card reference of target (unity-based)
*             InSubs = pointer to variable to receive input sub-unit count
*             OutSubs = pointer to variable to receive output sub-unit count
* Returns:    0 = success; non-zero = error code
*/
DWORD _stdcall PIL_EnumerateSubs (DWORD CardNum, DWORD *InSubs, DWORD *OutSubs) MYATTR;

```

These functions allow the user to query the number of input and output SubUnits present on a card.

The type and size of a SubUnit may be queried using one of the provided functions:

```

/* =====
* Function:    get Pickering card sub-unit information (in string format)
* Arguments:  CardNum = Pickering card reference of target (unity-based)
*             SubNum  = sub-unit of target to access (unity-based)
*             Out     = TRUE for output sub-unit, FALSE for input sub-unit
*             Str     = pointer to character string to receive result
* Returns:    0 = success; non-zero = error code
*/
DWORD _stdcall PII_SubType (DWORD CardNum,
                           DWORD SubNum,
                           BOOL Out,
                           CHAR *Str) MYATTR;

/* =====
* Function:    get Pickering card sub-unit information (numeric format)
* Arguments:  CardNum = Pickering card reference of target (unity-based)
*             SubNum  = sub-unit of target to access (unity-based)
*             Out     = TRUE for output sub-unit, FALSE for input sub-unit
*             TypeNum = pointer to variable to receive type code result
*             Rows    = pointer to variable to receive row dimension result
*             Cols    = pointer to variable to receive column dimension result
* Returns:    0 = success; non-zero = error code
*/
DWORD _stdcall PII_SubInfo (DWORD CardNum,
                           DWORD SubNum,
                           BOOL Out,
                           DWORD *TypeNum,
                           DWORD *Rows,
                           DWORD *Cols) MYATTR;

//*****
// Get sub-unit type
// Obtains sub-unit type description, in string format.
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   out (in)     VI_TRUE to obtain output sub-unit information
//               VI_FALSE to obtain input sub-unit information
//   subType (out) Identification string
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_getSubType (ViSession vi,
                                    ViUInt32 subUnit,
                                    ViBoolean out,
                                    ViString subType);

//*****
// Get sub-unit information
// Obtains sub-unit type information, in numeric format.
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   out (in)     VI_TRUE to obtain output sub-unit information
//               VI_FALSE to obtain input sub-unit information
//   subType (out) Sub-unit functionality
//   rows (out)   Sub-unit row dimension
//   columns (out) Sub-unit column dimension
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_getSubInfo (ViSession vi,
                                    ViUInt32 subUnit,
                                    ViBoolean out,
                                    ViPUInt32 subType,
                                    ViPUInt32 rows,
                                    ViPUInt32 columns);

```

The parameters of these functions define the card session handle, the SubUnit number, a boolean to define whether an output SubUnit is being queried, followed by either 3 pointers to integers which will be filled by the numerical SubUnit type and its dimensions or by a string to contain a textual version of the information.

2.4.1 Simple Switch Types

TYPE_SW

This is a simple collection of switches each of which may be independently controlled. Each switch is represented by a binary digit which may be set TRUE or FALSE to represent the energized or de-energized.

The principal functions to control such a SubUnit are:

```
/* =====
 * Function:   operate a single output bit
 * Arguments: CardNum = Pickering card reference of target (unity-based)
 *            OutSub  = sub-unit of target to access (unity-based)
 *            BitNum  = output bit number (unity-based)
 *            Action  = TRUE (for ON) or FALSE (for OFF)
 * Returns:   0 = success; non-zero = error code
 */
DWORD _stdcall PIL_OpBit(DWORD CardNum,
                        DWORD OutSub,
                        DWORD BitNum,
                        BOOL Action) MYATTR;

//*****
// Set the state of a single output
// Sets the state of an individual output channel.
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   channel (in) Channel number
//   state (in)   State to set (VI_OFF = channel OFF, VI_ON = channel ON)
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_setChannelState (ViSession vi,
                                         ViUInt32 subUnit,
                                         ViUInt32 channel,
                                         ViBoolean state);
```

The parameters of these functions define the card session handle, the SubUnit number, the index of the switch in the collection and the state to set that switch to. Since changing a relay state requires that due time is allowed for a mechanical relay to settle, by default each call delays by a time defined in the card firmware.

The state of any given switch may be viewed using the functions:

```

/* =====
 * Function:   get the state of an individual output bit
 * Arguments: CardNum = Pickering card reference of target (unity-based)
 *            OutSub  = sub-unit of target to view (unity-based)
 *            BitNum  = output bit number (unity-based)
 *            State   = pointer to BOOL variable to accept result
 * Returns:    0 = success; non-zero = error code
 */
DWORD _stdcall PIL_ViewBit(DWORD CardNum,
                           DWORD OutSub,
                           DWORD BitNum,
                           BOOL *State) MYATTR;

//*****
// Get output channel state
// Obtains the state of an individual output channel.
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   channel (in) Channel number
//   state (out)  The state of the specified output channel
//                (VI_OFF = OFF or logic '0', VI_ON = ON or logic '1')
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_getChannelState (ViSession vi,
                                          ViUInt32 subUnit,
                                          ViUInt32 channel,
                                          ViPBoolean state);

```

Since setting multiple switches individually would incur multiple settling delays, both drivers additionally provide means to control the entire SubUnit in a single operation with just a single settling delay.


```

//*****
// Set all of a sub-unit's outputs.
// Sets the states of all of a sub-unit's outputs.
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   pattern (in) Array containing a bit-pattern representing
//                 the desired output states
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_setChannelPattern (ViSession vi,
                                           ViUInt32 subUnit,
                                           ViAUInt32 pattern);

//*****
// Get all of a sub-unit's outputs.
// Gets the states of all of a sub-unit's outputs.
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   pattern (out) Array to accept a bit-pattern representing
//                 the sub-unit's output states
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_getChannelPattern (ViSession vi,
                                           ViUInt32 subUnit,
                                           ViAUInt32 pattern);

/* =====
* Function:   write data to an output sub-unit
* Arguments: CardNum = Pickering card reference of target (unity-based)
*            OutSub  = sub-unit of target to write (unity-based)
*            Data    = pointer to array of DWORD values to be written
* Returns:    0 = success; non-zero = error code
* The data array must be dimensioned to hold the number of bits representing
* the specified sub-unit. The appropriate number of least significant bits in
* the data array are written.
*/
DWORD _stdcall PII_WriteSub(DWORD CardNum,
                           DWORD OutSub,
                           DWORD *Data) MYATTR;

/* =====
* Function:   get the state of an output sub-unit
* Arguments: CardNum = Pickering card reference of target (unity-based)
*            OutSub  = sub-unit of target to view (unity-based)
*            Data    = pointer to DWORD array to accept result
* Returns:    0 = success; non-zero = error code
* The data array must be dimensioned to hold the number of bits representing
* the specified sub-unit. The result occupies the appropriate number of least
* significant bits in the data array.
*/
DWORD _stdcall PII_ViewSub(DWORD CardNum,
                          DWORD OutSub,
                          DWORD *Data) MYATTR;

```

In these cases the final parameter is an array of 32-bit unsigned integers sufficiently large that the bit count can contain the entire SubUnit. For example a SubUnit with 64 switches would require an array of 2 integers to contain the data.

To calculate the size of the array required to contain the data the user may use the 'SubInfo' function to obtain the number of rows and columns of the SubUnit and then calculate the number of 32-bit elements required to contain (rows*columns) bits:

```
size = (rows * columns - 1) / 32 + 1;
```

Note: Recent releases of PILPXI include a function to query the size:

```
DWORD _stdcall PIL_GetSize(DWORD CardNum,  
                          DWORD SubUnit,  
                          DWORD *bits,  
                          DWORD *dwords) MYATTR;
```

This function provides a calculation both of the number of bits in the SubUnit and the number of 'DWORD's required to contain them.

2.4.2 Multiplexer Types

TYPE_MUX

This is a collection of switches operating as a multiplexer, that is, one side of each switch is connected to a common connection and only one switch in the SubUnit may be energized at a time.

Energizing a switch in a multiplexer will automatically de-energize all other switches in that SubUnit.

Operation of this SubUnit type is through the use of:

DWORD PIL_OpBit

or

ViStatus pipx40_setChannelState

However, in this case the multiple switch function is NOT permitted and use of these functions will return an error code.

TYPE_MUXM

This is a rather strange type which is a hybrid between TYPE_SW and TYPE_MUX. It is a multiplexer in the sense that one side of all the switches are commoned together, however multiple connections are permitted.

In this case the multiple switch functions are permitted,

TYPE_MUXMS

This recently added type represents a MUXM SubUnit which is being forced to operate in classic MUX mode.

To force a MUXM to behave as a MUX, use the 'SetAttribute' function to set the type attribute of the subunit:

```
ATTR_TYPE = 0x400,      /* Gets/Sets DWORD attribute value of Type of the Sub-unit
                          (values: TYPE_MUXM, TYPE_MUXMS) */
```

2.4.3 Matrix Types

TYPE_MAT

This represents a basic matrix being a rectangular array of switches having a number of rows and columns.

The basic data representation treats the matrix as a collection of switches of size rows*columns, the data is packed contiguously into a 32-bit unsigned integer container, one bit per switch.

This SubUnit type may be handled like TYPE_SW, however the user would need to calculate the index of an individual switch from its row and column position.

To simplify matrix operation additional functions are provided:

```

/* =====
 * Function:      set the state of a matrix crosspoint
 * Arguments:    CardNum = Pickering card reference of target (unity-based)
 *              OutSub  = sub-unit of target to view (unity-based)
 *              Row     = crosspoint row (Y) location (unity-based)
 *              Column  = crosspoint column (X) location (unity-based)
 *              Action  = TRUE (for ON) or FALSE (for OFF)
 * Returns:      0 = success; non-zero = error code
 */
DWORD _stdcall PIL_OpCrosspoint(DWORD CardNum,
                                DWORD OutSub,
                                DWORD Row,
                                DWORD Column,
                                BOOL Action) MYATTR;

//*****
// Set crosspoint state (added 26/07/02)
// Sets the state of an individual matrix crosspoint.
//   vi (in)          Instrument handle
//   subUnit (in)    Sub-unit number
//   row (in)        Crosspoint row (Y) location
//   column (in)     Crosspoint column (X) location
//   state (in)      State to set (VI_OFF = crosspoint OFF, VI_ON = crosspoint ON)
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_setCrosspointState (ViSession vi,
                                              ViUInt32 subUnit,
                                              ViUInt32 row,
                                              ViUInt32 column,
                                              ViBoolean state);

```

These have corresponding 'get' functions to query the existing state of a switch.

2.4.4 Resistor SubUnits

Historically, Pickering first provided simple resistor cards which can be considered as a series chain of switches each associated with a resistor. Closing a relay would short its resistor, opening the relay puts its resistor in the series path. Operations on this type of resistor card are performed using the same functions as would be used on the 'TYPE_SW' SubUnit. The most widely used card of this type is the 40-295 range.

More recently Pickering introduced a range of 'precision' resistor cards. These cards provide much enhanced resistor elements, many include fine setting capability down to milliohm levels, all include calibration data and additional driver functions to facilitate setting of resistance by Ohmic value. The internal construction of these cards render operation with the simple bit-wise functions difficult, if not impossible. The 40-297 is one such card.

Operation of Simple Resistor SubUnits

These cards contain series of resistors in 1:2:4:8... sequence, each bit represents a resistor 2x the nominal value of the previous bit. So, to calculate the bit pattern required to achieve a particular overall resistance, the user should know the value of the lowest resistor and calculate the multiplier required. In many cases the lowest resistor is 1Ω, making the calculation trivial, the pattern required is simply the integer value of the required resistance.

For example, if the card has a lowest value of 0.5Ω and the user requires 100Ω, then the bit pattern required is $100/0.5 = 200$. The value of 200 should therefore be sent in a 'WriteSub' or 'SetChannelPattern' call.

The software provides no means to ascertain the lowest value of the resistor.

Operation of Precision Resistor SubUnits

Operation of these resistor types is straightforward:

```
//*****
// Set resistance as close as possible to the given value
//   vi (in)           Instrument handle
//   subUnit (in)      Sub-unit number
//   mode (in)         A value indicating how the given resistance value is to be
//   applied
//   resistance (in)   the resistance value to set
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_resSetResistance (ViSession vi,
                                           ViUInt32 subUnit,
                                           ViUInt32 mode,
                                           ViReal64 resistance);

/* =====
* Function:   set resistance as close as possible to the given value
* Arguments: CardNum = Pickering card reference of target (unity-based)
*            OutSub  = sub-unit of target to access (unity-based)
*            Mode    = a value indicating how the given resistance value is to be
*            applied
*            Resistance = the resistance value
* Returns:    0 = success; non-zero = error code
*/
DWORD _stdcall PII_ResSetResistance (DWORD CardNum,
                                     DWORD OutSub,
                                     DWORD Mode,
                                     double Resistance) MYATTR;
```

Corresponding 'get' functions are provided to retrieve the currently set value. It should be noted that these 'get' functions return the actual value set and not the value requested.

For example, a 40-297 card may have a resolution of 1Ω, if the user requests 100.8Ω the card will set to the nearest possible value, which would be 101Ω, use of the 'get' function will return the calculated pin-to-pin resistance of 101Ω and not the 100.8 requested value.

This feature allows the user a means to adjust an algorithm to compensate for the resolution of the resistor.

For example, if using the resistor to simulate a Platinum Resistance Thermometer and requires to set a resistance equivalent to 200°C, the actual resistance achieved could be used to calculate the equivalent temperature represented and so adjust their algorithm for the discrepancy.

At this point it is worth considering the utility library provided by Pickering Interfaces:

http://downloads.pickeringtest.info/downloads/drivers/Utility_Software/pi_prt.zip

This library contains conversion functions between temperature and resistance for a wide variety of standard PRT types.

Set Resistance Modes

The 'SetResistance' function has a parameter called 'Mode', this allows some control over the transition between resistance settings:

```
RES_MODE_SET           = 0, // Legacy/Default mode
                        // break before make with settling delay.
RES_MODE_MBB           = 1, // Make before break with settling delay.
RES_MODE_APPLY_PATTERN_IMMEDIATE = 2, // Apply new pattern immediately and
                        // wait till settling delay.
RES_MODE_NO_SETTLING_DELAY = 4, // Disable settling delay.
RES_MODE_DONT_SET      = 999, // Do the calculations but don't set the card.
```

RES_MODE_SET The default mode is to perform the transition from current value to target value in two stages, first any switches that are to be released are actioned followed by a settling delay, then the new pattern is applied fully actioning any relays that are require to be closed, followed by an-other settling delay. The purpose of this behaviour is the ensure that transient values of the pin-to-pin resistance of the chain always go higher than either current or target resistance which avoids any possibility of exceeding current ratings of any of the components in the chain.

Alternate modes have been provided which should be used with some caution:

RES_MODE_MBB This reversed the transition to perform the 'make' operations before the 'break' op-erations. This has the effect of forcing the transient values during the transition to be always lower than either current or target resistances. However, there could be a transient value approaching 0Ω and so could result in excessive current being drawn for a few milliseconds.

RES_MODE_APPLY_PATTERN_IMMEDIATE Does not split the change into two stages but applies the new pattern in one operation and then waits for settling. There is no control over the transition, but there is no settling delay during the transition so performing the change of resistance in short time, a frac-tion of a millisecond.

RES_MODE_NO_SETTLING_DELAY As for the immediate mode, but no settling delay. The user should allow sufficient time after the call to allow relays to be fully stable.

RES_MODE_DONT_SET Used mainly in-house for checking the driver algorithms, this performs all the internal calculations to arrive at the closest possible target resistance but does not actuate the relays. Could be useful to a user to calculate what resistance would be achieved.

The 'ResInfo' function allows the user to query a SubUnit to determine the resistance characteristics such as minimum and maximum values:

```
/* =====
 * Function:    obtain resistor characteristics
 * Arguments:  CardNum = Pickering card reference of target (unity-based)
 *             OutSub  = sub-unit of target to access (unity-based)
 *             MinRes  = pointer to variable to receive the minimum resistance setting
 *             MaxRes  = pointer to variable to receive the maximum resistance setting
 *             RefRes  = pointer to variable to receive the reference resistance value
 *             PrecPC  = pointer to variable to receive percentage precision (+/- per-cent)
 *             PrecDelta = pointer to variable to receive delta precision (+/- ohms)
 *             Int1    = pointer to variable to receive (currently unused) value
 *             IntDelta = pointer to variable to receive internal precision (+/- ohms)
 *             Capabilities = pointer to variable to receive capabilities flags
 * Returns:    0 = success; non-zero = error code
 */
DWORD _stdcall PIL_ResInfo(DWORD CardNum,
                          DWORD OutSub,
                          double *MinRes,
                          double *MaxRes,
                          double *RefRes,
                          double *PrecPC,
                          double *PrecDelta,
                          double *Int1,
                          double *IntDelta,
                          DWORD *Capabilities) MYATTR;

//*****
// Obtain detailed resistor information
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   minRes (out) pointer to variable to receive the minimum resistance setting
//   maxRes (out) pointer to variable to receive the maximum resistance setting
//   refRes (out) pointer to variable to receive the reference resistance value
//   precPC (out) pointer to variable to receive percentage precision (+/- per-cent)
//   precDelta (out) pointer to variable to receive delta precision (+/- ohms)
//   int1 (out)   pointer to (currently unused) variable
//   intDelta (out) pointer to variable to receive internal precision (+/- ohms)
//   capabilities (out) pointer to variable to receive capabilities flags
//   Return = ViStatus
//*****
```

2.4.5 Other SubUnit Types

TYPE_DIG

This is a digital I/O type, it is in all respects equivalent to the TYPE_SW.

Note that Input SubUnits are always of TYPE_DIG.

TYPE_ATTEN

Represents an RF attenuator, control is by setting of dB attenuation:

```

//*****
// Set attenuation value
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   atten (in)   The attenuation value to set
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_attenSetAttenuation(ViSession vi,
                                             ViUInt32 subUnit,
                                             ViReal32 atten);

/* =====
 * Function:    set attenuation value
 * Arguments:   CardNum = Pickering card reference of target (unity-based)
 *              SubNum  = sub-unit of target to access (unity-based)
 *              Atten   = the attenuation value to set
 * Returns:    0 = success; non-zero = error code
 */
DWORD _stdcall PII_AttenSetAttenuation(DWORD CardNum,
                                       DWORD SubNum,
                                       float Atten) MYATTR;

```

Corresponding 'get' functions are provided and an 'info' query function.

TYPE_PSUDC

Represents a DC power supply SubUnit set by a voltage value:

```

/* =====
 * Function:    set power supply output voltage
 * Arguments:   CardNum = Pickering card reference of target (unity-based)
 *              SubNum  = sub-unit of target to access (unity-based)
 *              Voltage = the voltage to set
 * Returns:    0 = success; non-zero = error code
 * Note:       The output voltage will be set to the nearest available DAC
 *              step value.
 */
DWORD _stdcall PII_PsuSetVoltage(DWORD CardNum,
                                 DWORD SubNum,
                                 double Voltage) MYATTR;

```

Corresponding 'get' functions are provided and an 'info' query function.

TYPE_BATT

```

/* =====
* Function:    set Battery Simulator (BATT type) channel output voltage
* Arguments:   CardNum = Pickering card reference of target (unity-based)
*             SubNum  = sub-unit of target to access (unity-based)
*             Voltage = the voltage to set (in Volts)
* Returns:    0 = success; non-zero = error code
*/
DWORD _stdcall PIL_BattSetVoltage(DWORD CardNum,
                                  DWORD SubNum,
                                  double Voltage) MYATTR;

/* =====
* Function:    set Battery Simulator (BATT type) channel sink current
* Arguments:   CardNum = Pickering card reference of target (unity-based)
*             SubNum  = sub-unit of target to access (unity-based)
*             Current = the current to set (in Amps)
* Returns:    0 = success; non-zero = error code
*/
DWORD _stdcall PIL_BattSetCurrent(DWORD CardNum,
                                  DWORD SubNum,
                                  double Current) MYATTR;

//*****
// Set battery simulator output voltage
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   voltage (in) Voltage to set
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_battSetVoltage (ViSession vi,
                                         ViUInt32 subUnit,
                                         ViReal64 voltage);

//*****
// Set battery simulator output sink current
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   current (in) Current to set
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_battSetCurrent (ViSession vi,
                                         ViUInt32 subUnit,
                                         ViReal64 current);

```

Corresponding 'get' functions are provided and 'info' query functions.

There are additional functions to control the interlock mechanism.

TYPE_VSOURCE

```

//*****
// Set voltage source output voltage
//   vi (in)      Instrument handle
//   subUnit (in) Sub-unit number
//   voltage (in) Voltage to set
//   Return = ViStatus
//*****
ViStatus _VI_FUNC pipx40_vsourceSetVoltage (ViSession vi,
                                           ViUInt32 subUnit,
                                           ViReal64 voltage);

```

Corresponding 'get' functions are provided and 'info' query functions.

TYPE_FI

A SubUnit type representing a Pickering Fault Insertion module. This sub-unit type prevents user to connect two TYPE_FI Fault buses together. The user can use SetAttribute function along with ATTR_MODE and RESTRICTED or UNRESTRICTED parameter to set or unset the functionality.

Restriction is set by default.

```

ATTR_MODE = 0x401, /* Gets/Sets DWORD attribute value of Mode of the Card */

/* Fault Insertion restrictions, to be used with Set Attribute function */

enum
{
    UNRESTRICTED = 0,
    RESTRICTED = 1
}

```

2.5 USING AN LXI PRODUCT

2.5.1 General

The LXI standard was introduced a few years ago to formalize the growing population of ethernet controlled instrumentation products.

The standard may be found here:

<http://www.lxistandard.org/>

Where additional information may be found.

Pickering Interfaces is one of the founding members of the LXI Consortium, has taken and continues to take an active role in the consortium.

2.5.2 Pickering LXI Products

The majority of Pickering LXI products are essentially PXI systems controlled by an internal operating system and exporting a programming interface over an ethernet link.

The driver used internally is the Pickering Direct I/O driver pilpxi, this is then interfaced to the server side of a client-server bridge, a client-side library set is provided for user interaction with the embedded PXI switching.

The bridge, which we call ClientBridge, mimics the normal PILPXI API interface but with added parameters to control the additional element of the IP address of the remote LXI.

A complete control session on an LXI based switch card has following format:

1. Establish a control session to the remote LXI.
2. Establish a control session on a PXI card within the LXI chassis.
3. Perform switching operations.
4. Close the PXI card session.
5. Close the LXI session.

Note: It is important to close sessions when they are no longer needed to avoid resource leaks from orphaned sessions.

The control of Pickering cards in an LXI chassis is closely associated with the PCI/PXI case.

The differences are:

- There is no VXIPnP/VISA driver, control is by a derivative of the PILPXI driver.
- The ClientBridge driver mimics the PILPXI function call set with only minor differences:
 - An additional parameter is used to define the network location of the LXI chassis containing the target card.
 - A few functions have been improved to define the size of passed arrays.
 - A set of new functions are provided to control discovery and access to the LXI chassis.

The user is referred to the preceding documentation regarding SubUnit types and control where all references to PILPXI functions may be substituted by PIPLX functions.

For example:

```
DWORD PIL_OpBit(DWORD CardNum, DWORD OutSub, DWORD BitNum, BOOL Action);
```

Becomes:

```
DWORD PIPLX_OpBit(SESSION Sid, DWORD CardNum, DWORD OutSub, DWORD BitNum, BOOL Action);
```

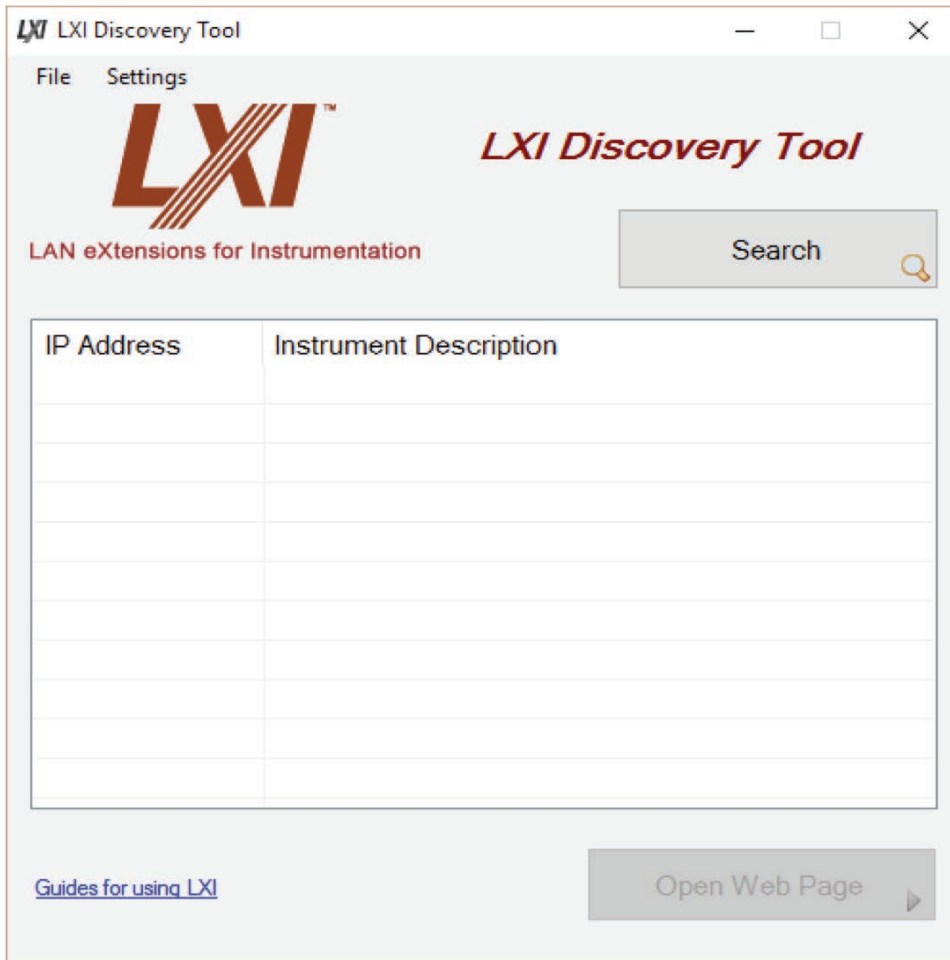
Notice the addition of a new parameter of type 'SESSION' which refers to a control session obtained on a remote LXI chassis.

2.5.3 LXI Discovery

Due to the nature of ethernet connections it can be difficult to locate the address of a remote LXI product. Some recent Pickering LXI products include a display to report the IP address, but older products do not.

The LXI standard defines mechanisms for discovery which have been adopted by the principle tool providers. For example, National Instrument includes LXI discovery in its Measurement and Automation Explorer (MAX).

Also available from the LXI Consortium website is a freely available LXI Discovery Tool:



Pickering Interfaces also provide some alternate discovery tools which are not documented here.

2.5.4 ClientBridge Communications Module – PICMLX

This module provides the mechanisms for discovery and session control to an LXI chassis. Once that session has been obtained, the session variable is used to address PIPLX calls to control switch cards, or other modules used for other product types.

The key function in this library is the 'Connect' call:

```
/**
 * <summary>
 *   Connect to an LXI device and establish communication.
 * </summary>
 * <param name="Board">Reserved for future use.</param>
 * <param name="Address">
 *   IP address or name of LXI. IP Address, name or "localhost" and "127.0.0.1"
 *   will be always trying to connect to ServerBridge. If you set it up to NULL or
 *   "PXI",
 *   than ClientBridge use an Pilpxi.dll functions => connection to PXI.
 * </param>
 * <param name="Port">Connection port of LXI.</param>
 * <param name="Timeout">Connection timeout in milliseconds.</param>
 * <param name="SID">Pointer to variable to receive valid handle of session.</param>
 * <returns>Zero for success, or non-zero error code.</returns>
 */
DWORD PICMLX_API PICMLX_Connect(DWORD Board,
                                const LPCHAR Address,
                                DWORD Port,
                                DWORD Timeout,
                                LPSESSION SID);
```

This function takes the IP address of a remote LXI chassis attached to a specified Ethernet board on the controlling computer and obtains a session value for use in card specific calls.

Note:

Local PCI/PXI devices may also be accessed using this library by using the Address string "PXI" instead of an IP address. This permits applications to be created that can access a card in either an LXI chassis or local PCI/PXI simply by changing the Address.

THIS PAGE INTENTIONALLY BLANK

SECTION 3 - IVI DRIVER PI40IV

3.1 THE IVI SWITCH CLASS

IVISwch is the IVI class driver for switching products. The specification for this driver class may be found at:

<http://ivifoundation.org/specifications/>

This driver is intended to encapsulate the functionality of switching products and systems and to provide the features of any IVI driver, that is interchangeability and ease of use.

The main benefits of this standardization are the potential benefits of interchangeability and the availability of software tools to assist developers of switching systems. Other articles in this Wiki also refer to use of the IVI driver IVI Driver

This article covers only the use of IVI Swch class defined functions, specific functions will be covered elsewhere.

3.2 THE IVI SWITCH PARADIGM

The IVI approach to switching differs from the approach taken by the majority of drivers. The IVI Swch driver considers a switching system as a 'black box' viewed from the perspective of its connection terminals and not from the detail of the switching elements used within. Switching routes are established by commanding the driver to Connect or Disconnect terminal pins which it refers to as Channels. Exactly how that route is established is entirely dealt with by the internals of the driver relieving the user of the need to understand the detail of the individual switches and interconnections within the 'box'.

The driver internally generates a set of channel names which are used in Connect and Disconnect function calls. Each manufacturer decides the channels names to apply and so this may differ from product to product, to deal with this variability the IVI system provides a means to 'alias' these channel names; this is one key aspect of interchangeability.

In the examples below we use the normal channel naming conventions used by Pickering Interfaces, other manufacturers may use different labelling, but will generally be of similar form,

The channel names applied to different switch forms in the Pickering pi40iv driver are shown in the following pages.

3.2.1 Simple Switch



The terminals or channels are prefixed 'com' and 'ch', the IVI Swch commands to control routing through this switch are:

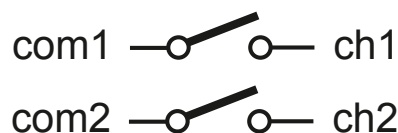
```
pi40iv_Connect(session_handle, "com", "ch");
```

```
pi40iv_Disconnect(session_handle, "com", "ch");
```

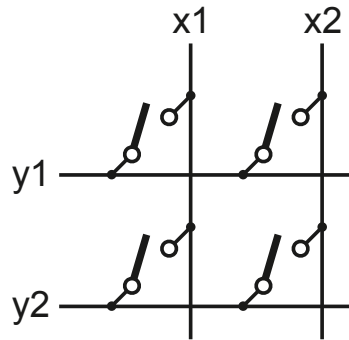
Where 'session_handle' has been previously obtained by opening a control session on the device containing the switch element.

In this case the Connect function will operate the switch so making a path between 'com' and 'ch', Disconnect will open the switch.

Where there are multiple switches within the switch product, each switch takes a numeric suffix to enumerate the series of switches.



3.2.2 Matrix

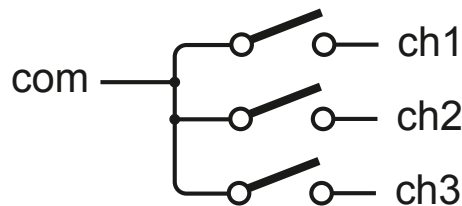


For a matrix element the columns are prefixed with 'x' and the rows with 'y', a numeric then iterates over the extents of the rows and columns.

```
pi40iv_Connect(session_handle, "x1", "y1");
```

```
pi40iv_Disconnect(session_handle, "x1", "y1");
```

3.2.3 Multiplexer



For a multiplex element the common connection is labelled 'com' and the multiplex channels 'ch' plus a numeric.

```
pi40iv_Connect(session_handle, "com", "ch1");
```

```
pi40iv_Disconnect(session_handle, "com", "ch1");
```

Some Pickering multiplex cards allow multiple selections within a multiplex, however the default IVI Switch behaviour is to permit only a single connection.

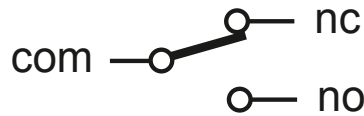
To allow multiple connections that default behaviour must be over-ridden, to do so modify the Option String to append the text 'DisMuxCheck;'. This will only work correctly on multiplex cards designed to allow multiple paths.

Important notes

When using IVI switching it is important to remember that the IVI Switch paradigm considers paths between terminal channels and not the internal switching. The driver keeps record of which paths are connected and will normally refuse to allow a path to be made if it comprises a channel already used in another route. The implication of this is that the user must Disconnect an existing route before using Connect to make a path using a common channel. This is particularly evident on a multiplex, if the user issues Connect("com", "ch1") then Connect("com", "ch2") the second call will be rejected since the channel 'com' is already in use. In order to make this alternate route the user must first Disconnect("com", "ch1").

3.2.4 Change-over or Form-C switch

The change-over switch needs some special attention.



The IVI driver considers routes through the switch object, not switch positions, this leads to the apparently odd situation that the changeover switch has in effect 3 possible states: no route, com-nc, com-no. In reality there can be no physical implementation corresponding to both paths being disconnected, however the concept is needed in order to support routing algorithms. If the user has explicitly made a connection “com”, “nc” then another route will be unable to use this switching element to construct another route. If the user issues disconnect “com”, “nc”, then both routes are available to routing algorithms.

However, there is no definition of what state the physical switch should take in the ‘all disconnected’ condition and different manufacturers may and do adopt different strategies. Pickering drivers will revert the switch to its default state, that is there will be a physical connection between “com” and “nc” but the IVI driver will record this as disconnected.

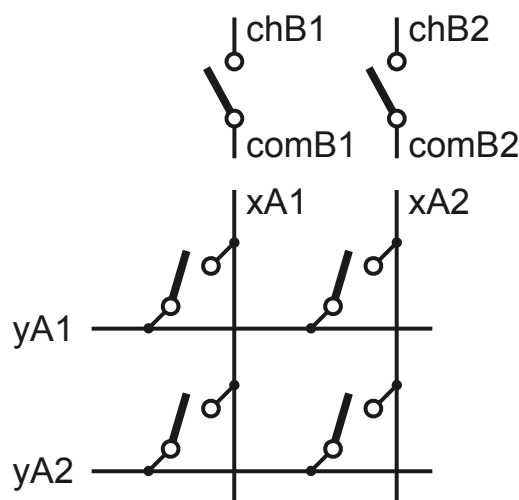
In order to deal with possible undesirable consequences of this ambiguity it is important that users are aware that this connection will exist and that they use the full and correct method that is that the previous connection is disconnected prior to making an alternate connect. In the case of Pickering cards it may be tempting to rely on the fact that Disconnect(“com”, “no”) will result in the default route “com”, “nc” being made, however if interchangeability is being used and the switch may be substituted by an alternate model in the future, then that alternate may behave differently and not fall back to the “com”, “nc” connection.

3.2.5 Multi-function switch products

Pickering provides a rich variety of switching products and many contain banks of different types of switch element. In the documentation for Pickering products we refer to these banks as ‘subunits’.

In such cases Pickering adds a letter to identify the different subunits.

For example, a matrix product with independent isolation switches on the X axis would contain 2 banks of switches, so two subunits. The first subunit will be a set of matrix switches, the second subunit will contain the isolation switches.



The letters A and B indicate the different switch banks in use.

Note that in this case the driver has no knowledge of connections between xA1 and comB1, and between xA2 and comB2. In any routing software hard wires would need to be added to complete the route using whatever technique that routing software provides.

3.3 USING THE IVI-C DRIVER

Obtaining a control session

A driver session may be opened on a target device using one of two possible pi40iv function calls:

pi40iv_InitWithOptions

pi40iv_init

```
ViStatus pi40iv_init      (ViRsrc resourceName,
                          ViBoolean IDQuery,
                          ViBoolean resetDevice,
                          ViSession *vi);

ViStatus pi40iv_InitWithOptions (ViRsrc resourceName,
                                  ViBoolean IDQuery,
                                  ViBoolean resetDevice,
                                  ViConstString optionString,
                                  ViSession *newVi);
```

The first parameter is the address of the target switch card or module and must comply with the normal conventions for addressing, for example:

“PXI5::15::INSTR”

or

“TCPIP::192.168.1.100::b5d15::INSTR”

or

Logical Name defined in the IVI Configuration Store

Note that Pickering used the 3rd section of the address string to define the location of a card in an LXI chassis, the syntax is a letter ‘b’ followed by the bus number then a letter ‘d’ followed by the device number. So, the above example refers to the card at PXI address 5::15 at the IP address 192.168.1.100.

The values of the 2nd and 3rd parameters are optional.

The OptionString parameter defines the model number of the card, it must be a model number listed in the pi40iv.ini file located in the installation folder, usually “C:\Program Files (x86)\IVI Foundation\IVI\Drivers\pi40iv”. A brief extract of that file:

```
[MODULE_INDEX]
1 = 40-110-021
2 = 40-110-121
3 = 40-115-021
```

If the card to be controlled is not listed in the pi40iv.ini file first try updating the file, the latest release can be found at:

<http://www.pickeringtest.info/downloads/drivers/IVI/>

If the card is still not listed, then contact Pickering support at support@pickeringtest.com.

The final parameter is a reference to a variable to hold the session token.

The better way to open a card is to define the address and options in the IVI Configuration Store and to then use pi40iv_init.

In this case the resourceName is the Logical Name of the device from the IVI Configuration store.

This is better because the address and model number are no longer hard-coded, so in the event of an address change or a change in the card model being used, the code does not need to be changed, all the changes can occur within the Configuration Store.

An even more elegant approach that takes advantage of the ‘Interchangeable’ feature of IVI uses the class driver function ivi_init, this is covered later.

3.4 OPERATING A SWITCH

The IVI-C Switch driver operates switches by defining the end-points of the path to be connected or disconnected. These are called channel names.

```
pi40iv_Connect(session_variable, "CHANNEL1", "CHANNEL2");
```

The driver defines a list of channel names for the opened card that will follow the following rules.

Simple Switch:

For a simple switch the channel names are "com#" and "ch#" where # is a numeric value. So, for example, a switch card containing 4 simple switches will use:

```
com1-ch1  
com2-ch2  
com3-ch3  
com4-ch4
```

Typical commands will be:

```
pi40iv_Connect(session_variable, "com1", "ch1");  
pi40iv_Disconnect(session_variable, "com1", "ch1");
```

Multiplexer:

A multiplexer is in effect a set of simple switches with a common connection on one side.

These will be labelled "com" and "ch#". So, for example, a 4 way multiplexer will use:

```
com-ch1,ch2,ch3,ch4
```

Typical commands will be:

```
pi40iv_Connect(session_variable, "com", "ch1");  
pi40iv_Disconnect(session_variable, "com", "ch1");
```

Note: The default behaviour is to treat a multiplexer as 'pure', that is only one connection may be made. Some Pickering cards permit multiple connections, to allow this mode of operation the initialize function 'optionString' should include an additional element "DisMuxCheck;"

Matrix:

A matrix has a set of X connections and a set of Y connections, a path is formed by connecting an X to a Y. So, a 4x4 matrix will have end-points:

```
x1, x2, x3, x4, y1, y2, y3, y4
```

Typical commands will be:

```
pi40iv_Connect(session_variable, "x1", "y1");  
pi40iv_Disconnect(session_variable, "x1", "y1");
```

Multiple banks of switches:

Many Pickering cards contain multiple banks, for example a dual 4x1 multiplex. Where multiple banks exist the driver appends a letter to the channel name prefix.

For example a dual 4x1 multiplex will have channel names:

```
comA, chA1, chA2, chA3, chA4  
comB, chB1, chB2, chB3, chB4
```

The letter corresponds to the sub-unit number on the card, A=1, B=2 etc.

3.5 INTERACTION WITH NI MAX

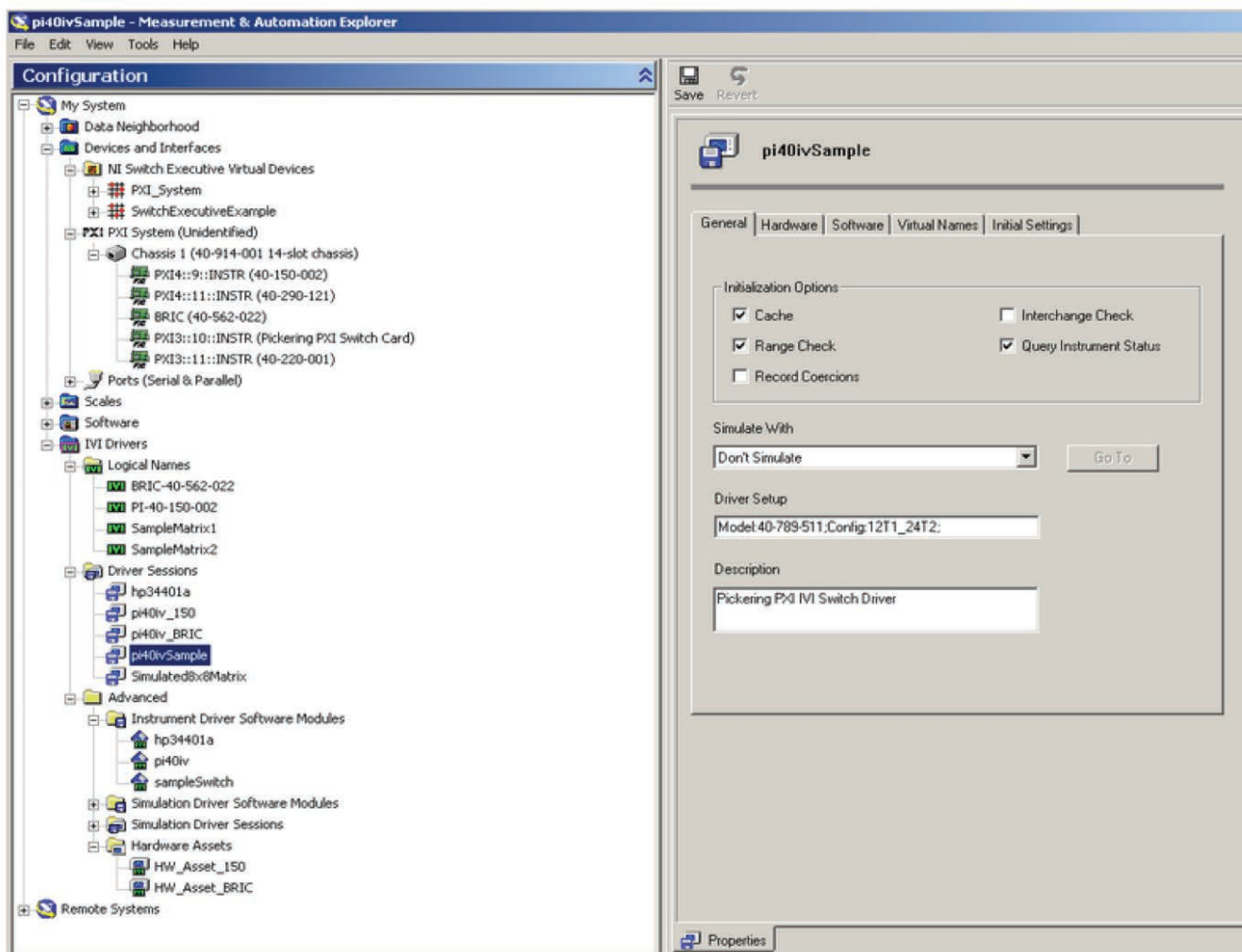
To set up a card to work with the IVI system requires that the IVI Configuration Store has correct entries defining the card, its address and manufacture specific data. While it is possible to edit the store directly, it is not advisable; the most common means to set up a card is to use NI Measurement and Automation Explorer (NI MAX).

The Pickering Interfaces IVI installation pi40iv.msi places an example driver session (pi40ivSample) into the MAX configuration (see picture below).

There is also an entry in the Instrument Driver Software Modules folder (pi40iv). All created devices for PI switch modules will relate to these entries.

Each instrument used must have a valid resource descriptor in the Hardware Assets folder or the “Simulate with” entry field is set to “Specific Driver” otherwise an error will occur when creating a Switch Executive Virtual Device.

Note: The images below might not reflect the latest MAX version.



One way to setup MAX for Pickering Interfaces switch modules is to use 'Create a New Driver Session' and copy the format of the provided pi40ivSample. This will be described on the following pages.

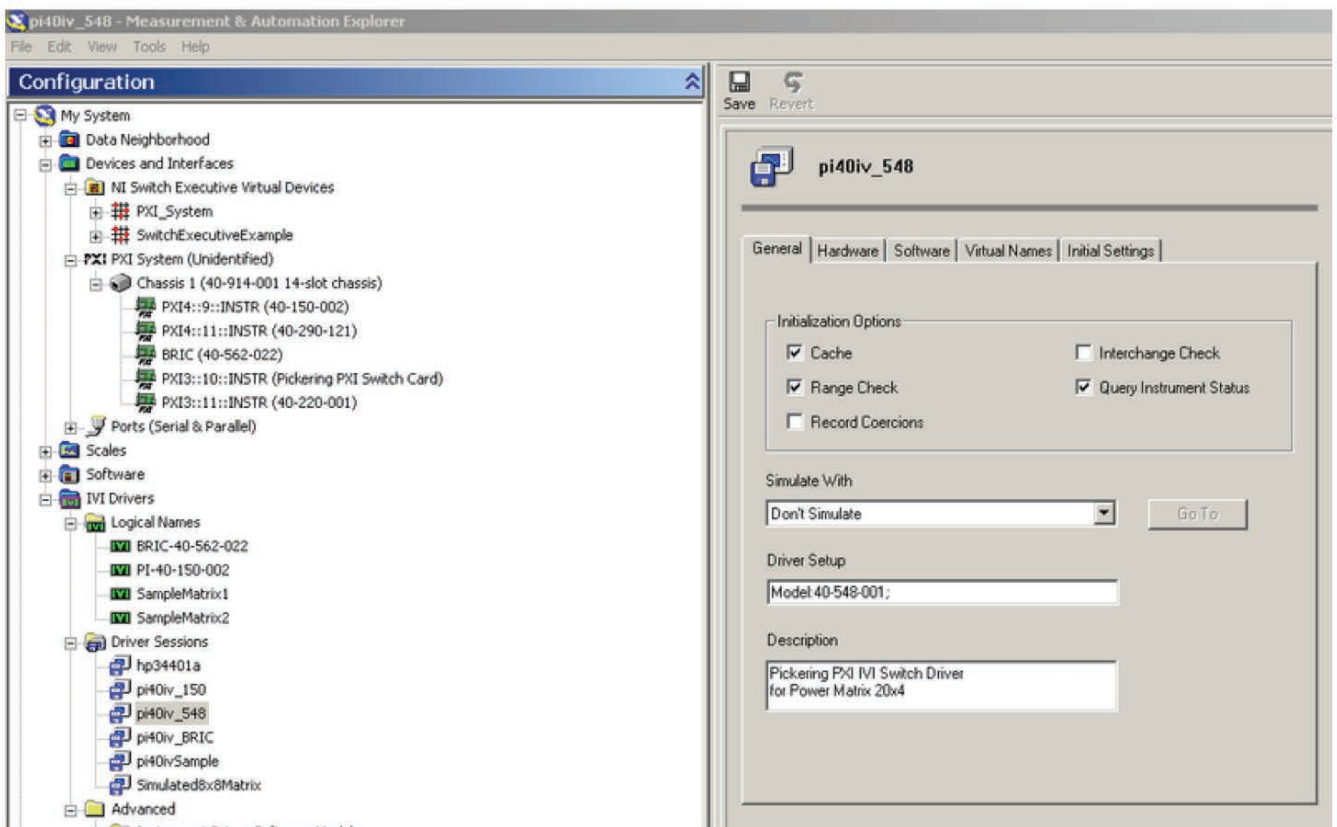
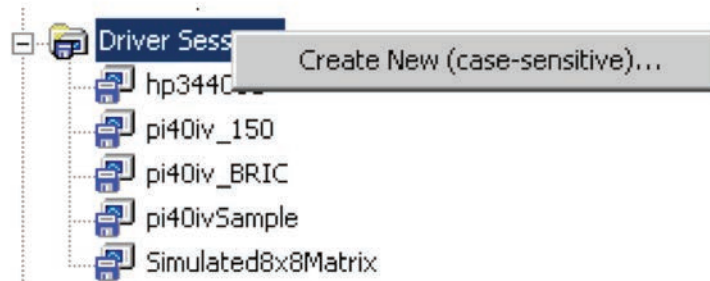
A simple way to add Pickering devices to the store is to use the Pickering IVI Wizard described in Section 5.

3.6 EXAMPLE: ADDING PI MODULE 40-548-001 USING MAX

3.6.1 Create IVI Driver Session

Create a New Driver Session by right clicking the Driver Session folder

In the driver's general tab the most important entry is the model description: **Model:40-548-001**

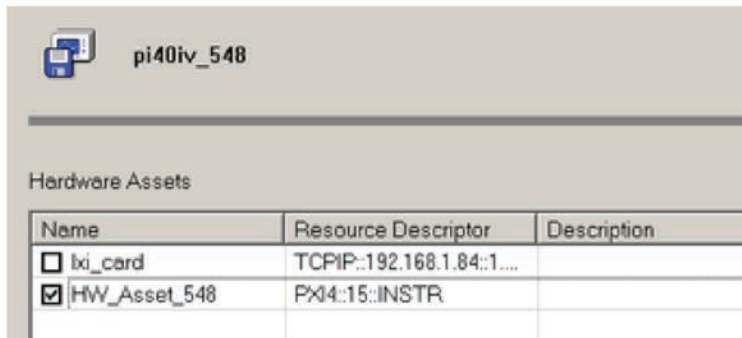


The model number entered must match exactly with a model in the Pickering card definition file pi40iv.ini which may be located in the IVI folder system, usually at C:/Program Files/IVI Foundation/IVI/Drivers/pi40iv.

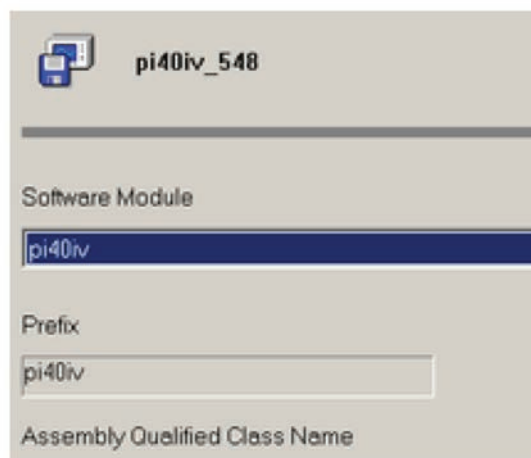
To simulate the card, select 'Specific Driver' in the 'Simulate With' selection combo box.

In the driver's hardware tab a hardware asset with the exact resource description for each module has to be selected.

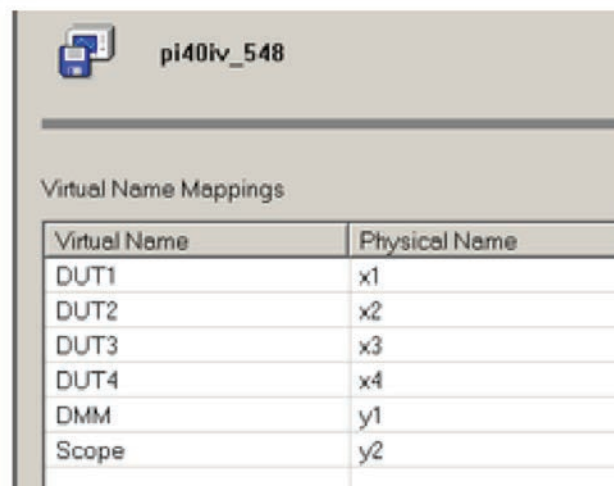
If required create a new asset and change the descriptor:
 (Don't forget to check the tickbox to select the asset !)



In the driver's software tab select the pi40iv as the necessary software module:



If you wish to use Virtual Names, these can be defined on the Virtual Names tab:



Please note however that NI Switch Executive does NOT pick up these Virtual Names and so aliases would have to be applied within Switch Executive.

3.6.2 Create IVI Logical Name

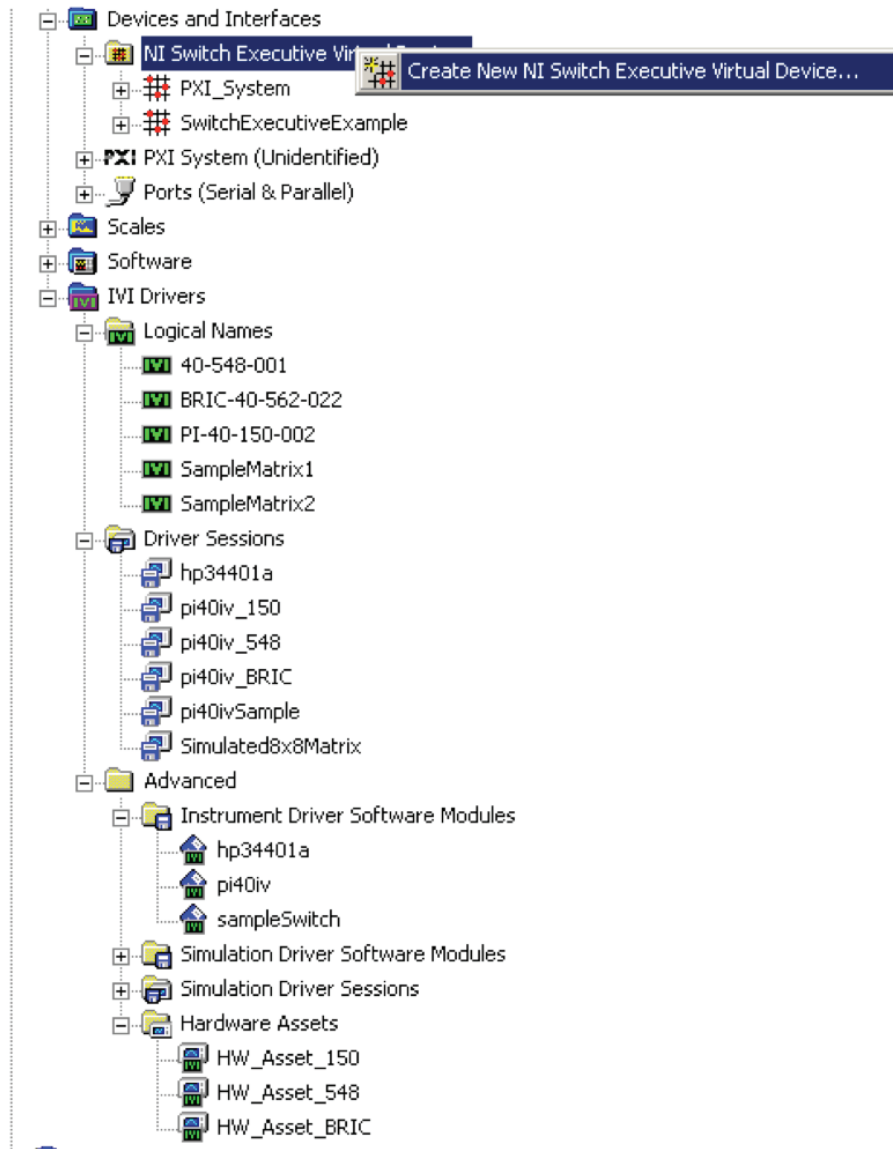
Create New Logical Names by right clicking the Logical Names folder.

A driver session for the logical name is selected in the consecutive window.

3.7 USING THE IVI DRIVER IN NI SWITCH EXECUTIVE

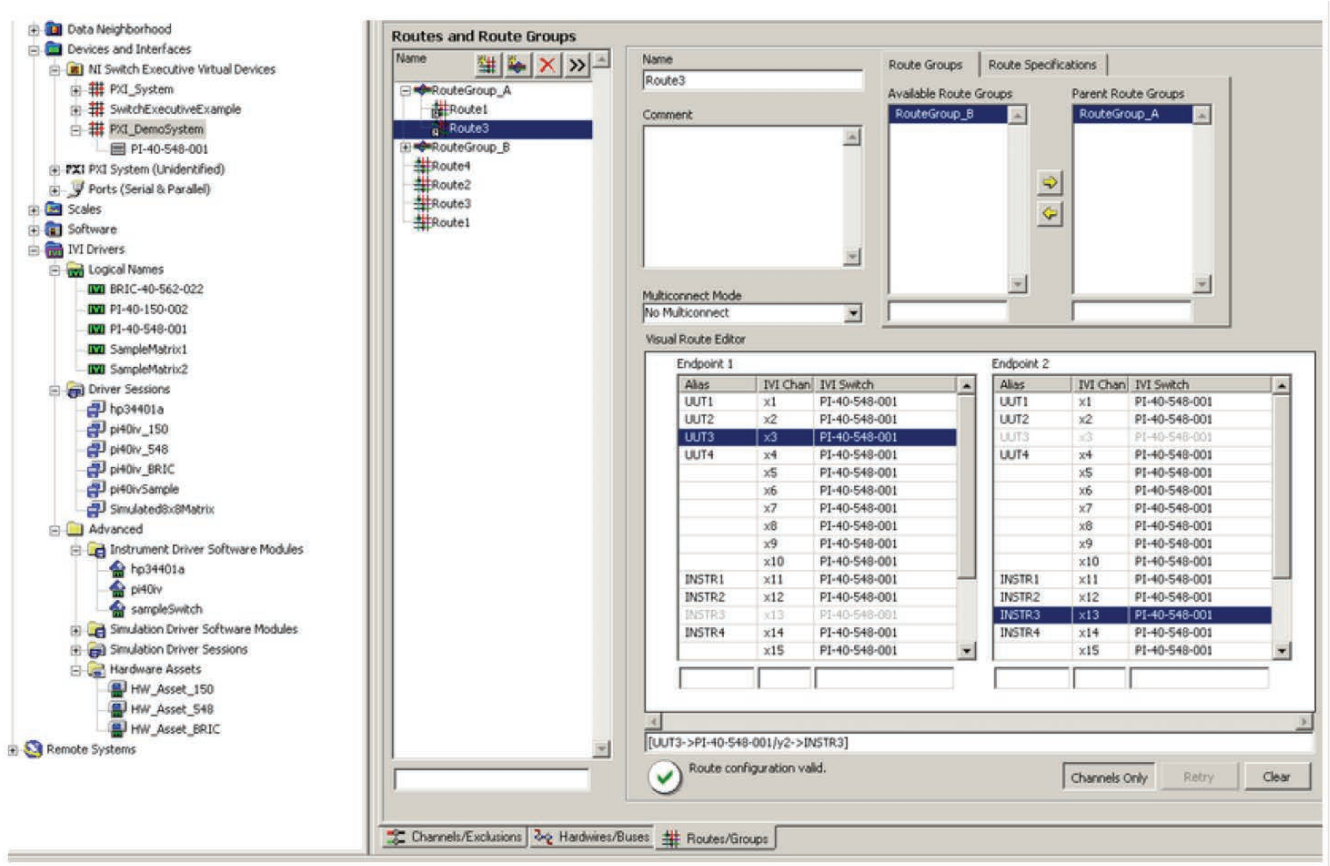
Create New Switch Executive Virtual Device by right clicking the 'Switch Executive Virtual Device' folder. Add all necessary devices from the list.

All system modules which have to be interconnected must be listed in one device:



3.7.1 Edit the Virtual Device, Adding Aliases

Edit the IVI channel alias names, check the channels which will be reserved for routing (usually the y-channels). In the example no exclusion are defined:



3.7.2 Edit the Virtual Device, Adding Routes

The screenshot displays the Pickering software interface for editing a virtual device. The left pane shows a tree view of the device hierarchy, including 'PXI DemoSystem' and 'Logical Names'. The main area is divided into two panels: 'Channels' and 'Exclusions'.

Channels Panel:

| Alias | IVI Chan | IVI Switch |
|--------|----------|---------------|
| UUT1 | x1 | PI-40-548-001 |
| UUT2 | x2 | PI-40-548-001 |
| UUT3 | x3 | PI-40-548-001 |
| UUT4 | x4 | PI-40-548-001 |
| | x5 | PI-40-548-001 |
| | x6 | PI-40-548-001 |
| | x7 | PI-40-548-001 |
| | x8 | PI-40-548-001 |
| | x9 | PI-40-548-001 |
| | x10 | PI-40-548-001 |
| INSTR1 | x11 | PI-40-548-001 |
| INSTR2 | x12 | PI-40-548-001 |
| INSTR3 | x13 | PI-40-548-001 |
| INSTR4 | x14 | PI-40-548-001 |
| | x15 | PI-40-548-001 |
| | x16 | PI-40-548-001 |

Exclusions Panel:

| Name | Type |
|----------------|--------|
| SourceChannels | Mutual |

Available Channels Table:

| Alias | IVI Chan | IVI Switch |
|-------|----------|---------------|
| UUT1 | x1 | PI-40-548-001 |
| UUT2 | x2 | PI-40-548-001 |
| UUT3 | x3 | PI-40-548-001 |
| UUT4 | x4 | PI-40-548-001 |
| | x5 | PI-40-548-001 |
| | x6 | PI-40-548-001 |

Edit Routes and Route Groups as your connections require.

Note:

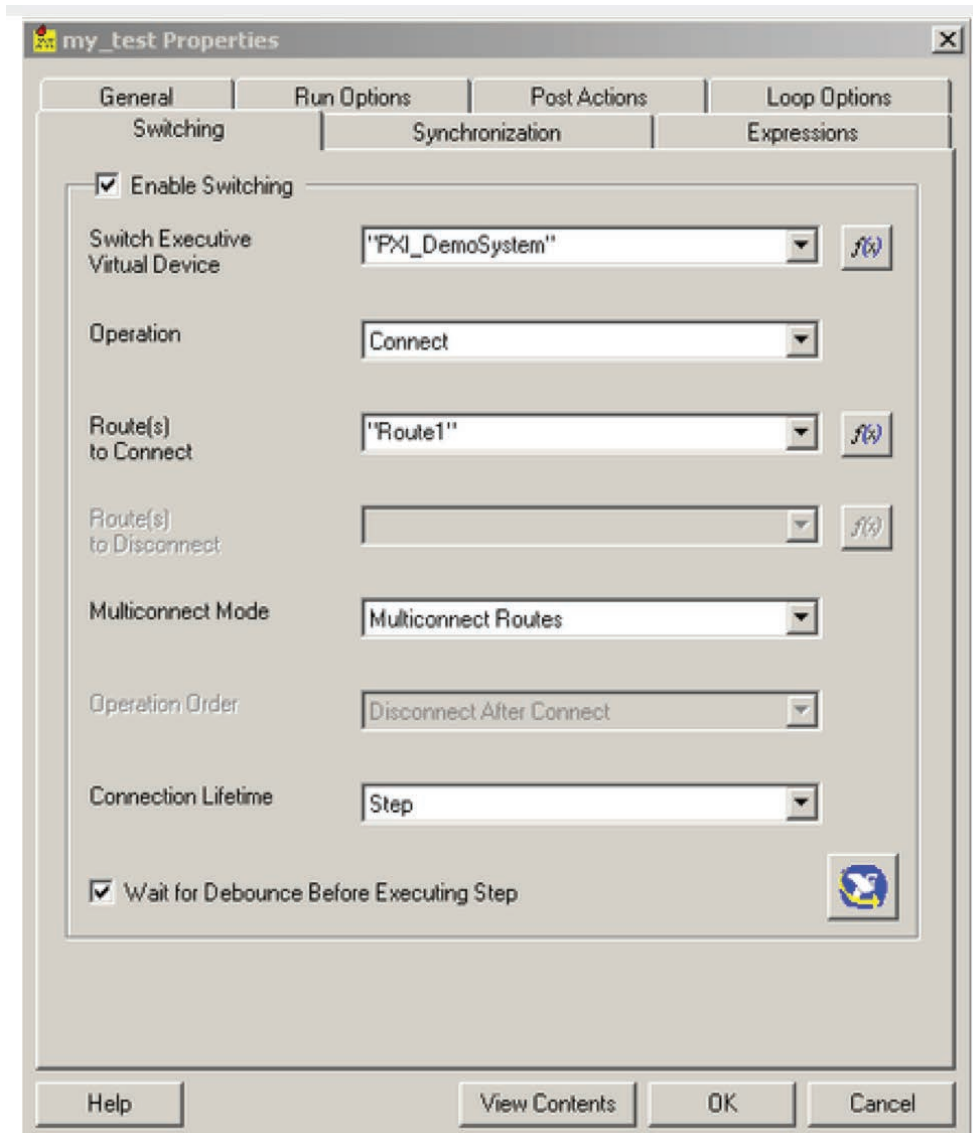
Pickering Interfaces offer an alternative to Switch Executive: Switch Path Manager. Please see:

<http://www.pickeringtest.com/product/switch-path-manager-software>

This product integrates the entire range of Pickering switch products.

3.8 USING THE IVI DRIVER IN NI TestStand

Launch TestStand. Create a simple test sequence with one test step. Right click the test step and select 'Properties'. In the Switching tab, the required switching can now be selected:



SECTION 4 - PROGRAMMING ENVIRONMENTS

4.1 PICKERING SOFTWARE ENVIRONMENTS

4.1.1 Switch Path Manager (SPM)

The Switch Path Manager signal routing software speeds up switch system development by managing switch module configurations and automated signal connections using endpoint aliases.

The software supports all Pickering's PXI, LXI and PCI switching systems.

Within small switching system configurations, or when utilizing just single switch modules, the user typically applies device drivers with the provided API to control the relays. Simple CLOSE and OPEN commands with additional parameters like module number and channel number control the required relays. As switching systems get more complex, this seemingly simple task can get quite complicated.

The user must always take care in order to avoid shorts or malfunctions even when performing simple switching tasks. If there are many relays involved, the risk of error increases significantly. Add to this the potential complexity of test code for the overall test program, and developing and debugging of the switching portion can be lengthy and prone to error.

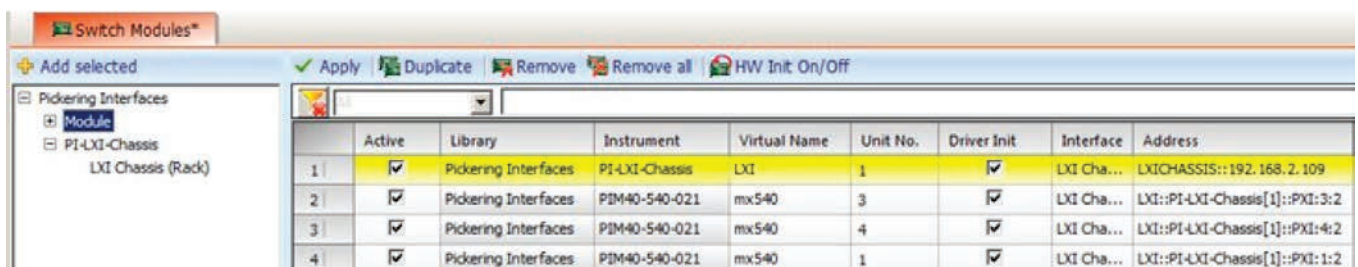
Switch Path Manager virtually describes any switching architecture and processes all stored project data for switching and routing at runtime. The user selects the modules from the library, and defines the physical interconnections, as well as the endpoints. Endpoints are the boundary of the system where measurement and stimuli equipment and all the UUT access points are connected. By calling Point-to-Point or Point-to-Multipoint functions the routing is processed and the required relays are controlled to establish a signal path between these endpoints. The router will never interfere with existing routes and will find an alternate bypass or will terminate with an error message if not successful. In addition, the router is intelligent enough to minimize the risk of unintended short circuits.

APIs available for C, C++, .NET, LabWindowsCVI LabVIEW and Python.

Switching in 4 Steps

Just 4 steps to perform switching:

Step 1 - Add your switching modules to your SPM project and set its resource addresses:

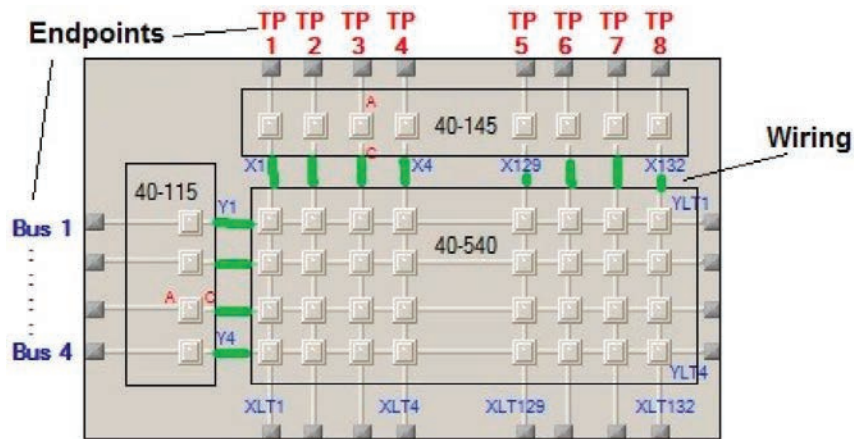


| | Active | Library | Instrument | Virtual Name | Unit No. | Driver Init | Interface | Address |
|---|-------------------------------------|----------------------|----------------|--------------|----------|-------------------------------------|------------|---------------------------------|
| 1 | <input checked="" type="checkbox"/> | Pickering Interfaces | PI-LXI-Chassis | LXI | 1 | <input checked="" type="checkbox"/> | LXI Cha... | LXI CHASSIS::192.168.2.109 |
| 2 | <input checked="" type="checkbox"/> | Pickering Interfaces | PIM40-540-021 | mx540 | 3 | <input checked="" type="checkbox"/> | LXI Cha... | LXI::PI-LXI-Chassis[1]::PXI:3:2 |
| 3 | <input checked="" type="checkbox"/> | Pickering Interfaces | PIM40-540-021 | mx540 | 4 | <input checked="" type="checkbox"/> | LXI Cha... | LXI::PI-LXI-Chassis[1]::PXI:4:2 |
| 4 | <input checked="" type="checkbox"/> | Pickering Interfaces | PIM40-540-021 | mx540 | 1 | <input checked="" type="checkbox"/> | LXI Cha... | LXI::PI-LXI-Chassis[1]::PXI:1:2 |

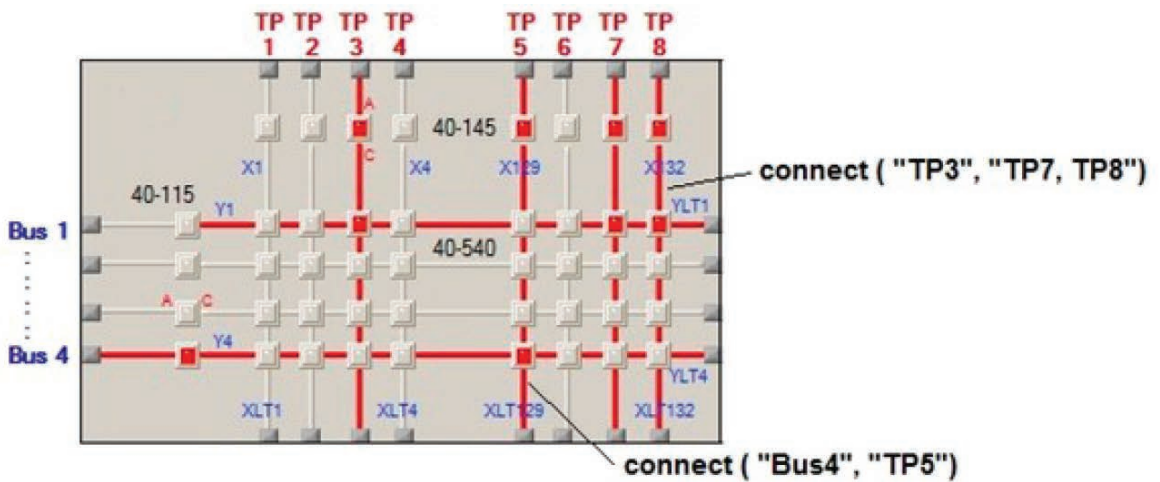
Step 2 - Virtually wire the modules interconnection: describe the physical wiring.

In case there is only one module used in the configuration omit Step 2 and continue with Step 3.

Step 3 - Define all required endpoints:



Step 4 - Connect and disconnect endpoints:



More details can be found in the SPM Manual or at:

<http://www.pickeringtest.com/de-de/product/switch-path-manager-software>

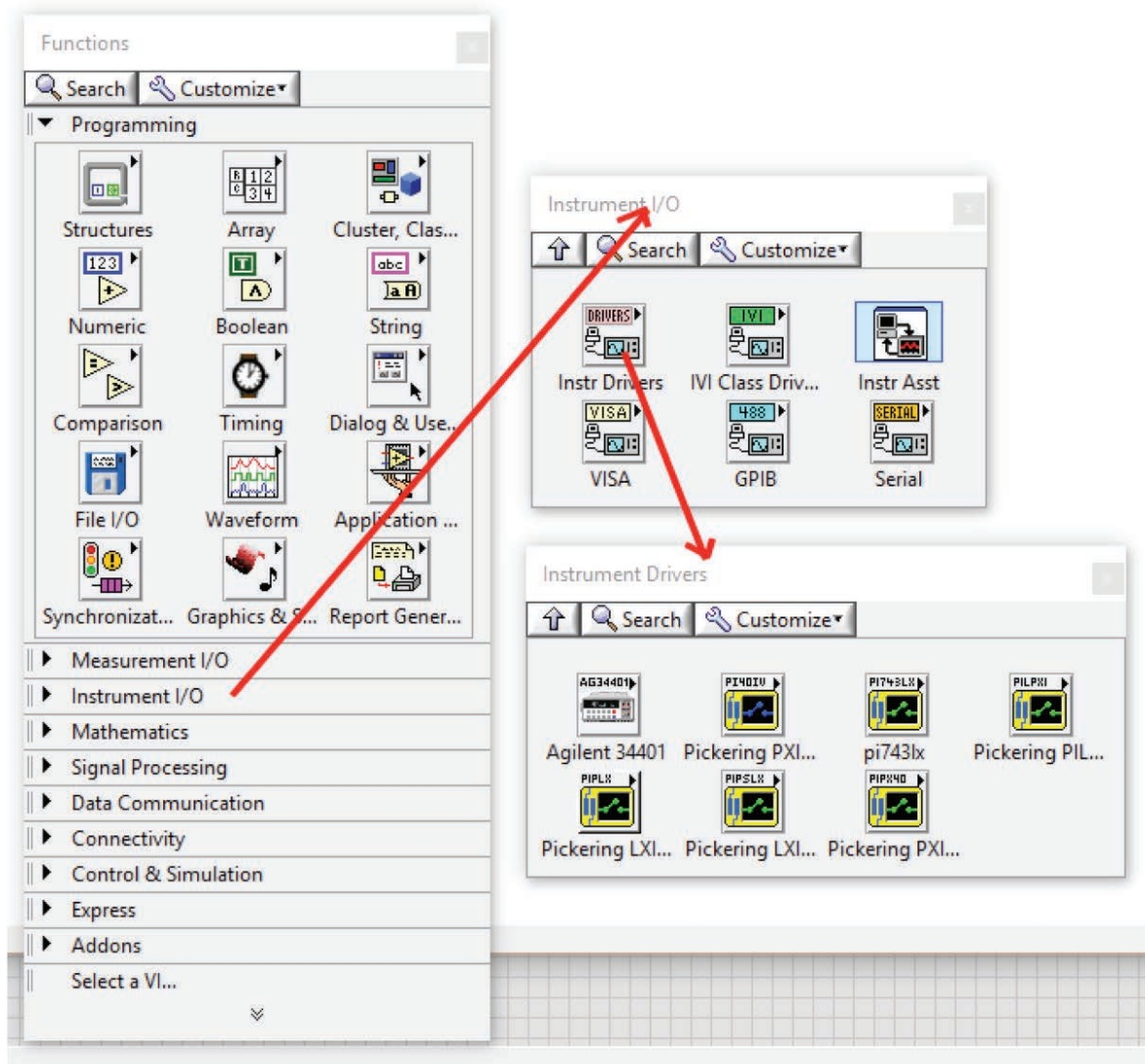
4.2 NATIONAL INSTRUMENTS SOFTWARE ENVIRONMENTS

4.2.1 LabVIEW

All Pickering drivers include a LabVIEW library to permit full operation of the Pickering product from the LabVIEW environment. These wrappers are normally installed to the current LabVIEW folder system during installation of the Pickering driver.

Since the installer always attempts to install the library to the LabVIEW version indicated in the Windows repository as being the latest version, there are times when the libraries need to be added to an earlier LabVIEW version that may be present. To achieve this, copy the LabVIEW folder from the Pickering installation folder to the instr.lib folder of the LabVIEW installation then start or re-start LabVIEW. Copying from another LabVIEW instr.lib folder may not work.

The control palettes can be found in the normal manner and are located thus:



The process of controlling a Pickering product normally consists of the following steps:

1. Open a control session on the product
2. Use functions in the library to operate the product
3. Close the control session

In the case of LXI products there is an additional step to establish a communication session on the remote LXI device prior to opening the control session on the hardware. This communication session should also be terminated when no longer required.

The LabVIEW libraries are wrappers of the underlying low level driver, full details of the functions can be found in the corresponding low level driver documentation.

Various examples of LabVIEW vi's may be found on the Pickering download site at:

http://downloads.pickeringtest.info/downloads/example_software/LabVIEW/

Included are examples for the various Pickering drivers and a selection of Pickering cards. Search in the folder related to the driver in use and in the 'product specific' folder.

Most of the examples were generated using LabVIEW 2016 and so cannot be used in earlier versions of LabVIEW. Contact support@pickeringtest.com if no suitable example can be found, we may have other examples available or may be able to export an example to an earlier LabVIEW version.

4.2.2 LabWindows/CVI

Most Pickering drivers include a CVI wrapper to permit full operation of the Pickering product from the LabWindows/CVI environment.

The LabWindows/CVI libraries are wrappers of the underlying low level driver and full details of the functions can be found in the corresponding driver documentation.

Example software is available on the Pickering download site at:

http://www.downloads.pickeringtest.info/downloads/example_software/LabWindowsCVI/

4.2.3 LabVIEW RT and LabWindows/CVI RT

The Pickering VXIPnP/VISA driver, pipx40, is suitable for use with LabVIEW RT and LabWindows/CVI RT systems.

Install pipx40 normally onto the development system, this should place a copy of the pipx40 LabVIEW library into the instr.lib folder of the LabVIEW installation.

Copy the required files to the remote LabVIEW RT system /ni-rt/system folder.

These are:

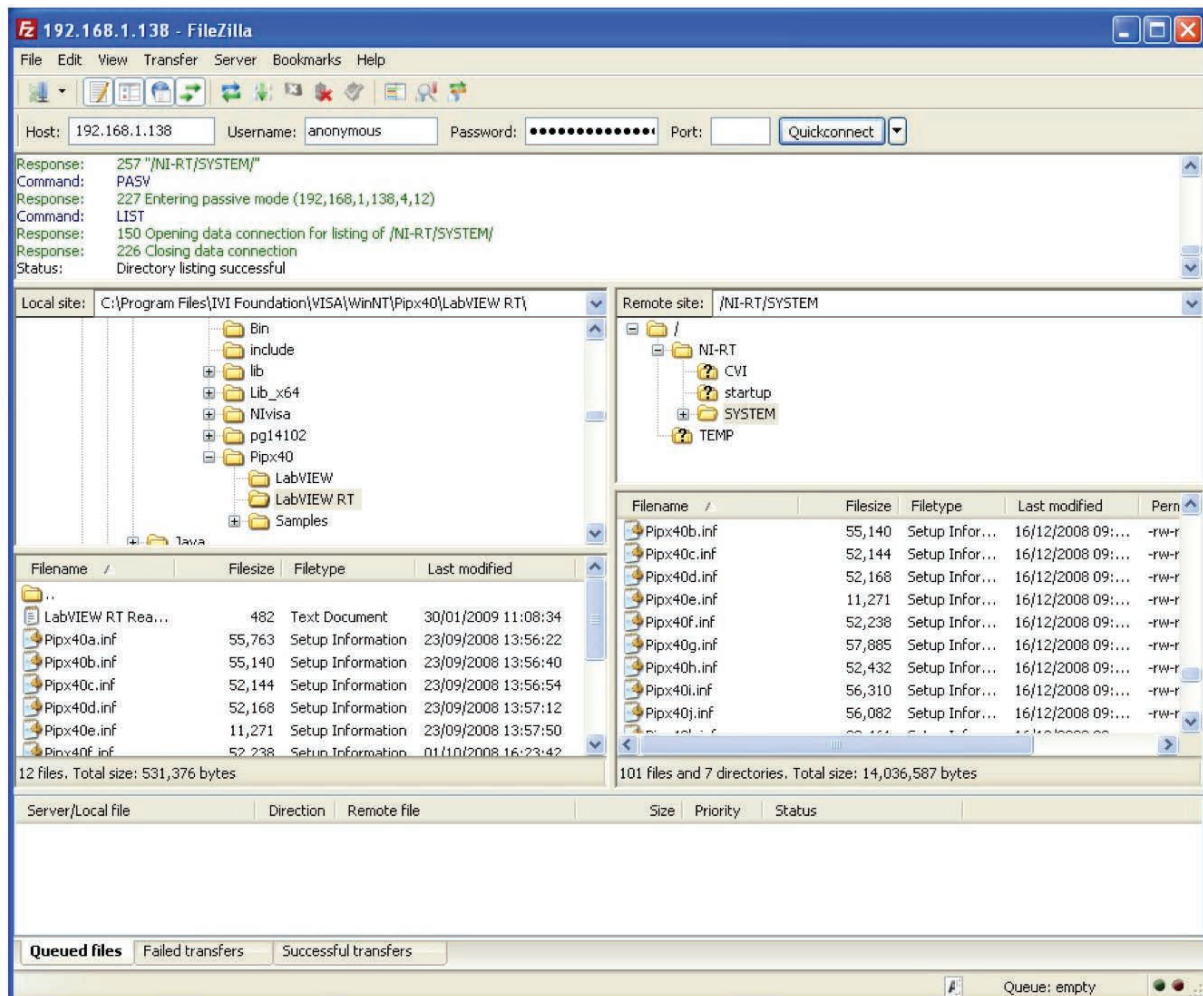
Pickering card INF files, which may be found in:

C:\Program Files (x86)\IVI Foundation\VISA\WinNT\Pipx40\LabVIEW RT

The pipx40_32.dll, which may be found in:

C:\Program Files (x86)\IVI Foundation\VISA\WinNT\Bin

For example, using Filezilla:



4.2.4 VeriStand

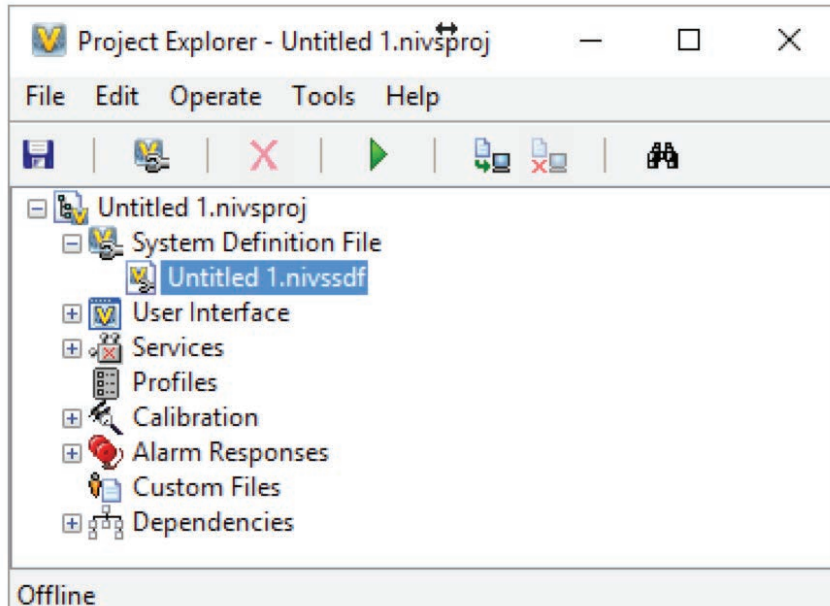
Pickering VeriStand packages may be found here:

<http://downloads.pickeringtest.info/downloads/drivers/Veristand/>

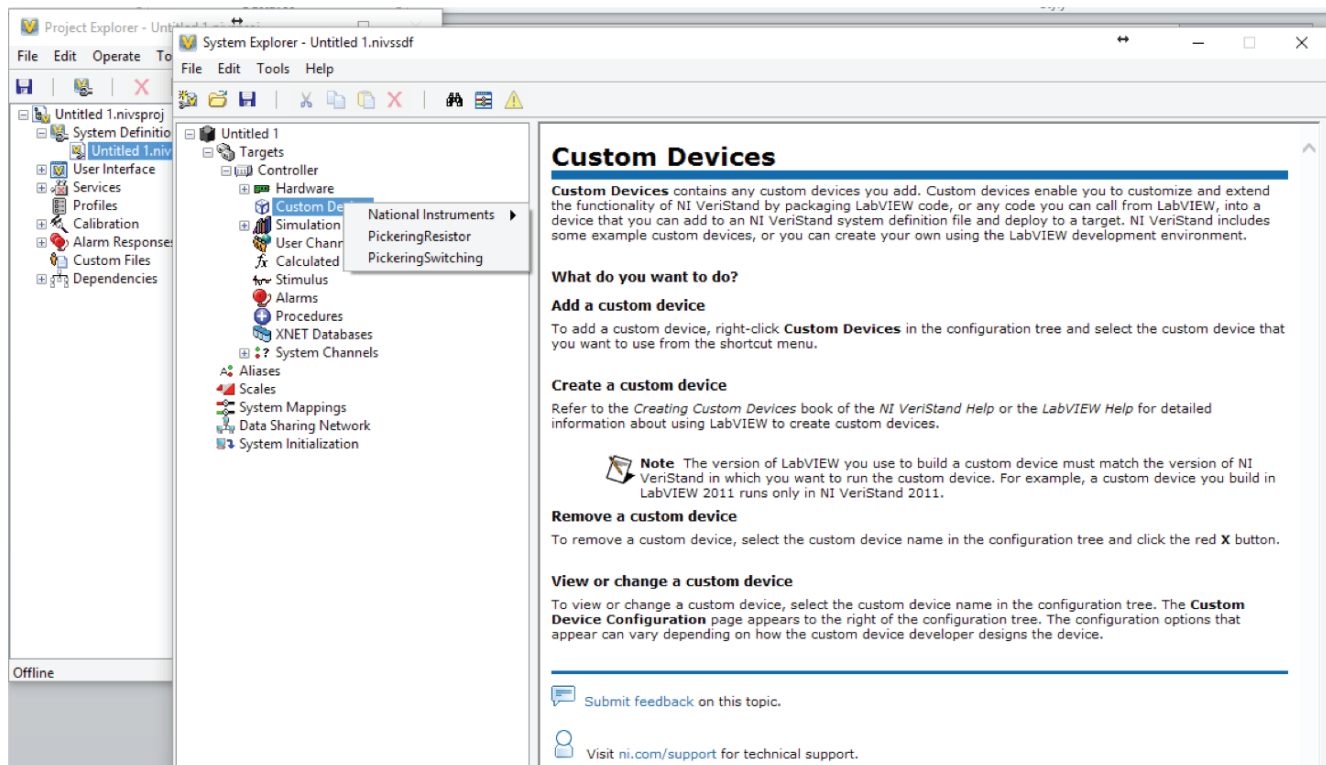
Unpack the zipped file and copy to the VeriStand Custom Device folder,

...Documents\National Instruments\NI VeriStand\Custom Devices

Open VeriStand and start a new project, then open the **System Definition File**:

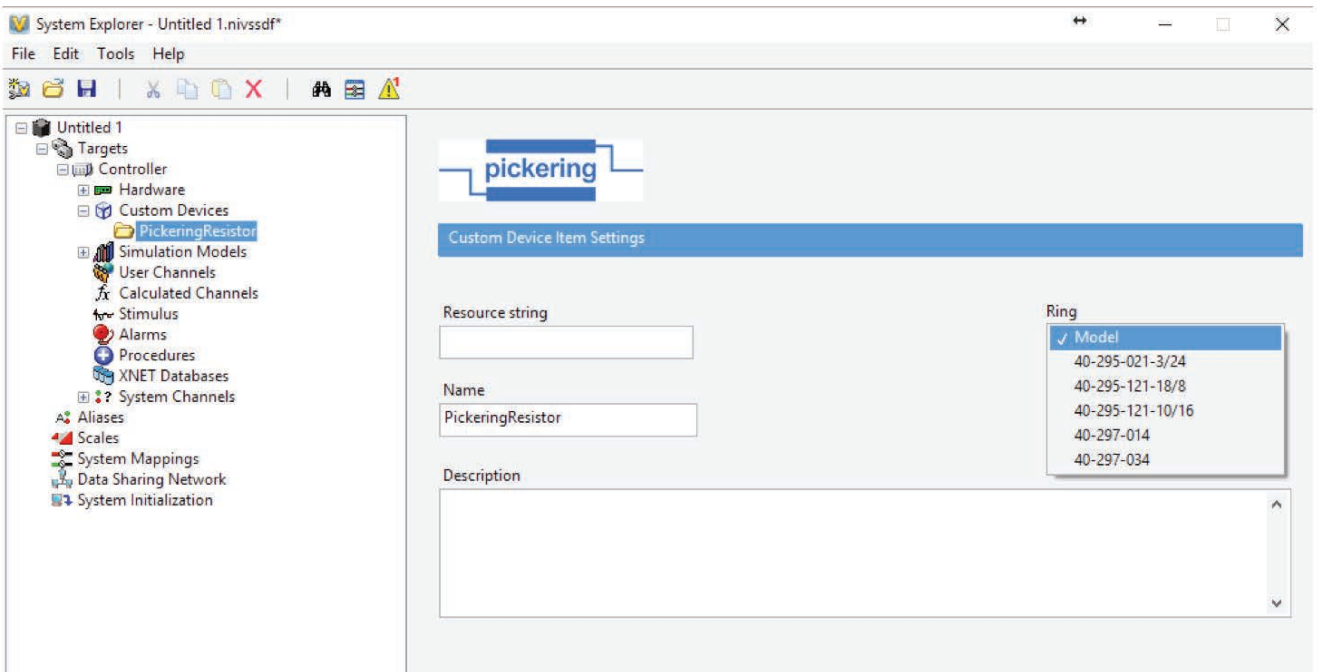


Go to Targets -> Controller and right-click Custom Devices:

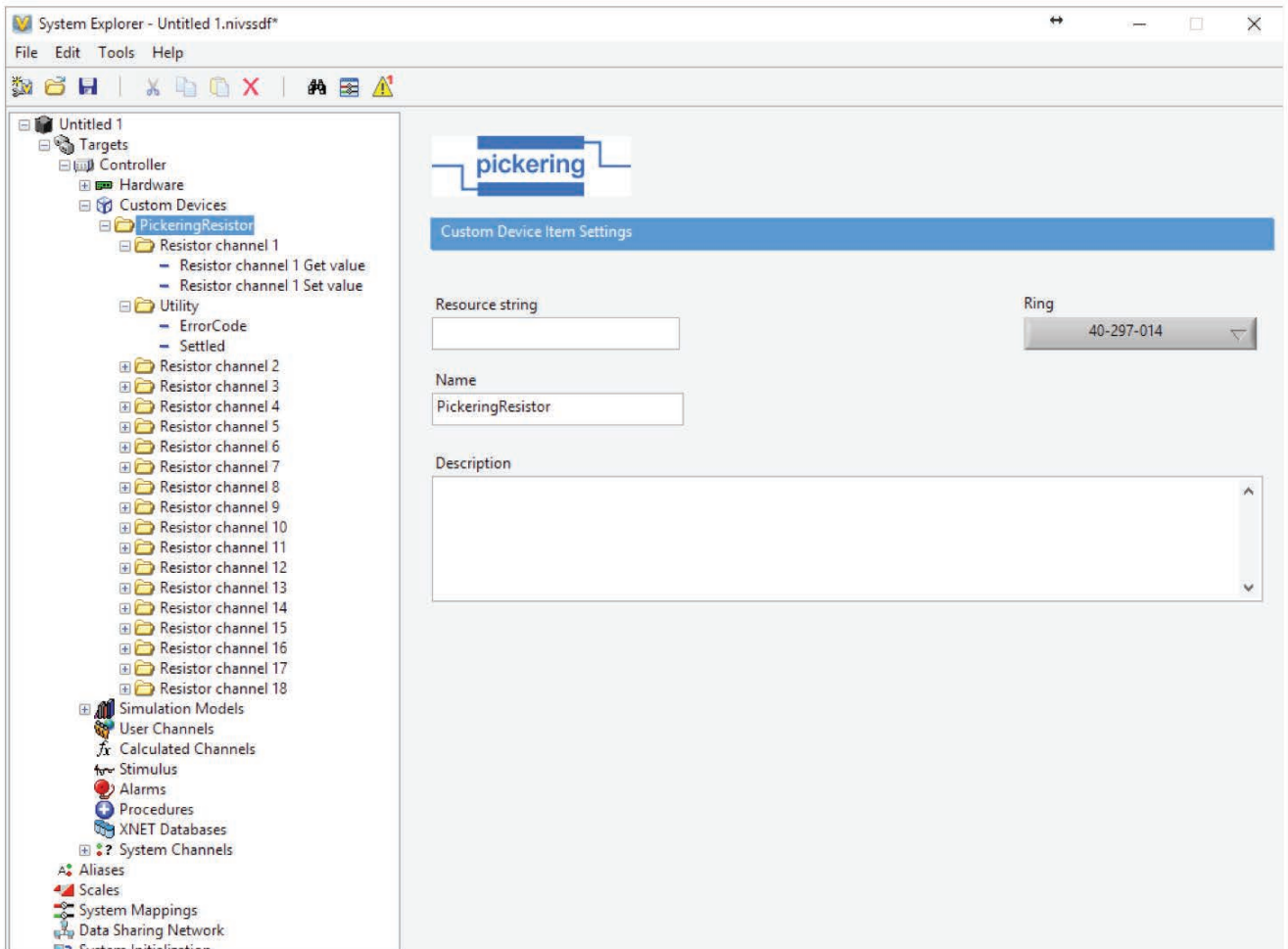


Select the custom device you want to add.

In the configuration page, select the card model and resource string:



Channels will be generated that may be used in the usual way:



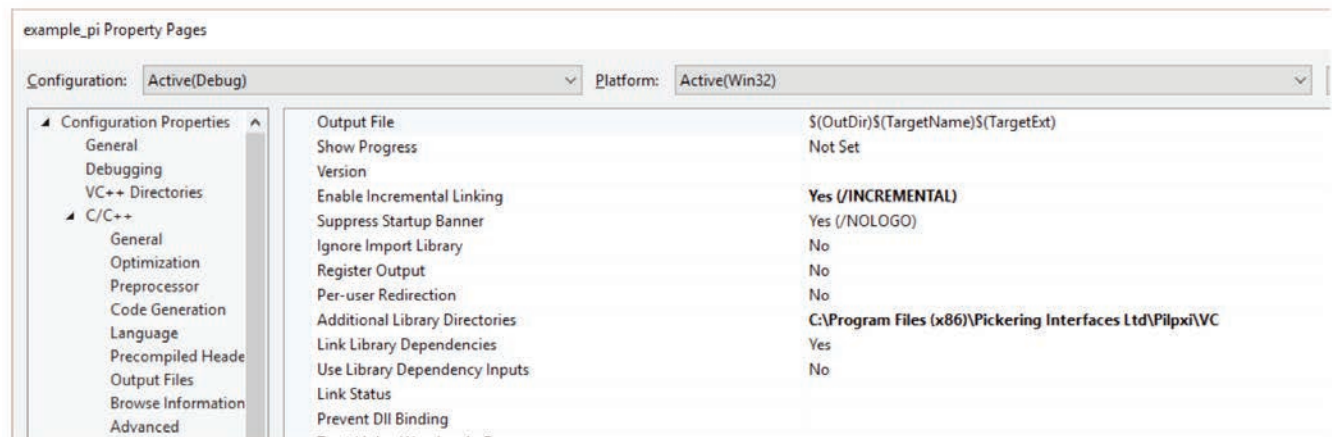
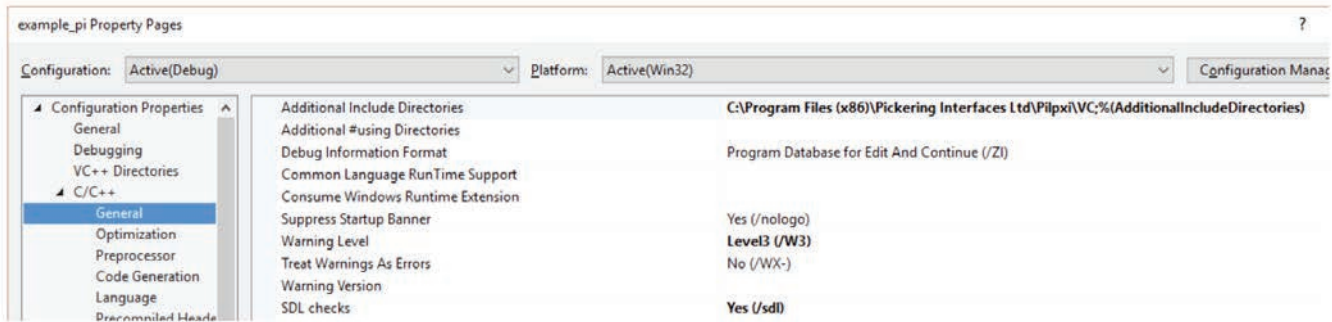
4.3 MICROSOFT VISUAL STUDIO

4.3.1 C/C++ AND PILPXI

To compose an application using the PILPXI Direct I/O driver one must include the Pilpxi.h file to obtain function declarations for the API calls and add pilpxi.lib to the linker.

First add the path to the installation folder containing the required files to the project Properties pages. The path is normally:

```
C:\Program Files (x86)\Pickering Interfaces Ltd\Pilpxi\VC
```



Then include <Windows.h> to obtain definitions of the data types used and <Pilpxi.h> to get the API function declarations.

The application must be linked to pilpxi.lib.

Refer to the section on PILPXI and the Appendixes for more detail of the API calls available.

Example programs are provided in the installation folder and on the Pickering website.

4.3.2 C/C++ and PIPX40

To compose an application using the PIPX40 VXIPnP driver one must include the Pipx40.h file to obtain function declarations for the API calls and add pipx40.lib to the linker.

In this case the data type definitions are obtained by including <Visa.h>.

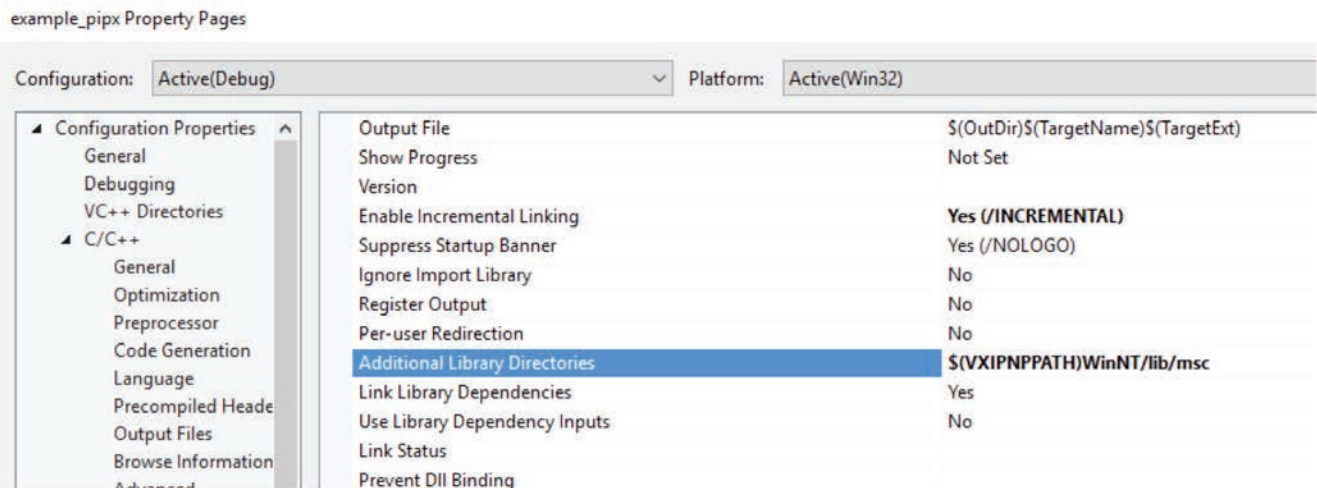
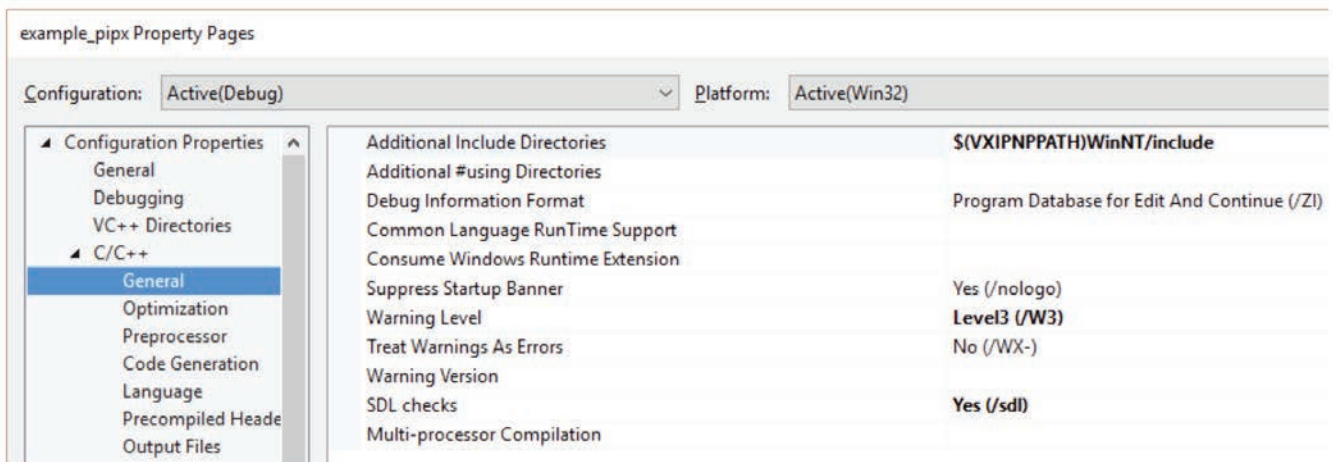
The paths to these files is easiest added with the help of an environment variable added during installation of the VISA software system, VXIPNPPATH, extended to create the paths to the include and lib files.

`$(VXIPNPPATH)WinNT/include`

and

`$(VXIPNPPATH)WinNT/lib/msc`

In the case of Pipx40, discovery of cards is left to the VISA system and so is beyond the scope of this document. The reader is referred to the documentation on viFindRsrc.



4.3.3 C/C++ and PI40IV

To compose an application using the PI40IV IVI driver one must include the pi40iv.h file to obtain function declarations for the API calls and add pi40iv.lib to the linker.

In this case the data type definitions and API function declarations are obtained by including <pi40iv.h>, which in turn includes the necessary header files.

The paths to these files is easiest added with the help of an environment variable added during installation of the IVI software system, IVIROOTDIR32, extended to create the paths to the include and lib files.

4.4 PICKERING .NET DRIVER

Pickering provide a native .NET interface and a set of .NET interop assembly 'wrappers' for individual drivers.

4.4.1 Native .Net Interface

The .NET driver offers a native .NET interface suitable for use in any .NET programming environment, such as C#, C++ or Visual Basic.

Alone it is suitable for Pickering LXI product control, but with the additional installation of the 'normal' PXI driver package it is also suitable for control of PCI and PXI products.

Installation Procedure:

The driver is located on the distribution disk in the following location:

Drivers_LX\dotnet_setup.zip

or may be downloaded from the Pickering download website at:

www.pickeringtest.info/downloads

The normal installation folder for the driver is:

C:\Program Files\Pickering Interfaces\ClientBridge.NET SDK 1.1

This folder contains the drivers, plus documentation and example programs. The driver is rich in functionality and the user is advised to examine these items before attempting to program any cards.

Additional Installation Procedure - PXI/PCI Products:

To control PXI/PCI products directly connected to the host PC PCI bus, the Direct I/O driver must also be installed. This may be installed either before or after the .NET driver.

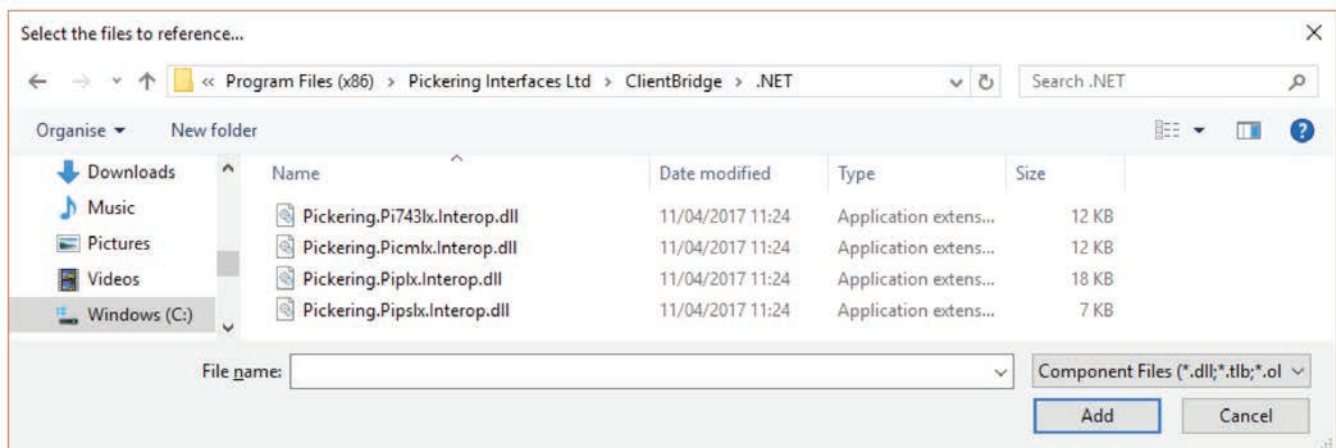
Adding the .NET driver to a Visual Studio Project:

The .NET driver package must be added to Visual Studio before it may be accessed from a program, proceed as follows.

- a) Create a Visual Studio project in the normal way
- b) Add a reference to the Pickering .NET drivers

From the Project menu item, select Add Reference...

This will bring up the Add Reference dialog box. Browse to the folder containing the assemblies and add to the project.



To control a switch card two of these assemblies are required: Picmlx and Piplx.

4.4.2 .NET Wrappers

After installation of the driver packages the relevant .NET interop assemblies may be found at the following locations:

C:\Program Files (x86)\IVI Foundation\VISA\WinNT\Pipx40\NET

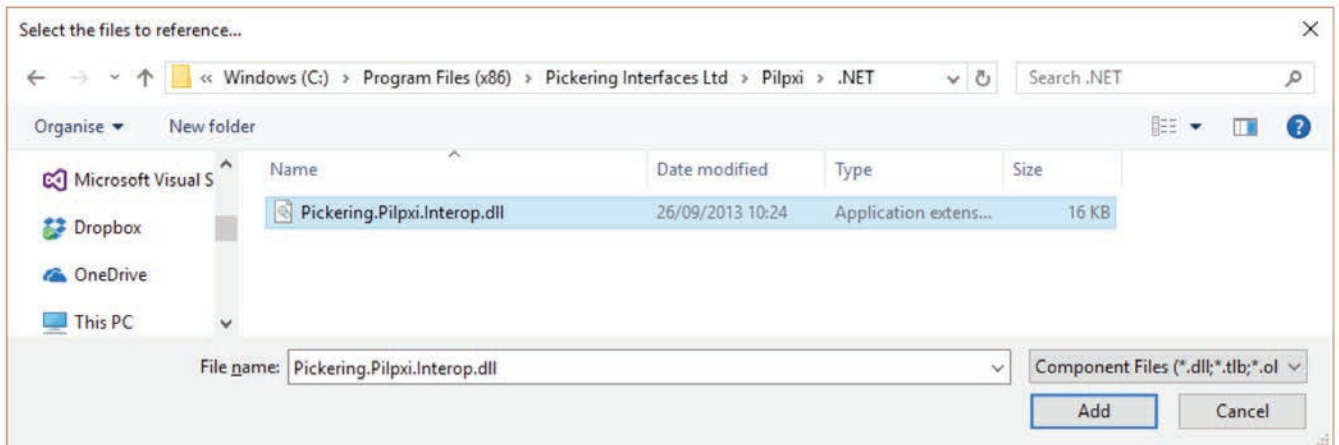
and

C:\Program Files (x86)\Pickering Interfaces Ltd\Pilpxi\NET

and

C:\Program Files (x86)\IVI Foundation\IVI\Drivers\pi40iv\NET

Add the assembly to a project using the Visual Studio/Project/Add Reference menu item, for example:



Example programs may be found at:

http://downloads.pickeringtest.info/downloads/example_software/dotNET/

4.4.3. The Communication Module

This sub-section contains “How To” information regarding the use of the Communication Module. The module is part of the .NET Client Bridge and is a common module for other modules of the Client Bridge. The following is a list of questions and answers which may help you if you want to use the Communication Module.

Q: What is the Communication Module?

A: The communication module is a part of the .NET Client Bridge. It contains classes which are used by other modules or classes to support the communication interface.

Q: What type of devices can it be used for?

A: It is designed only for LXI devices. PXI or PCI devices use a different technique.

Q: Is it CLS compliant?

A: Yes, it can be used in C#, CB.NET, Delphi, etc.

Q: What .NET platforms are supported?

A: At this time only Microsoft .NET (MS.NET) and Mono are supported (for desktop or server computers). Others such as .NET Compact Framework or COM are not supported. For more information, contact PI support: support@pickeringtest.com

Q: What services does the Communication Module offer?

A: The Communication Module offers services only for LXI devices. Not for PXI or PCI devices. Offered services are:

- Searching for LXI devices connected to a network.
- Getting information about the LXI devices.
- Providing access to an LXI device.
- Reading number of cards based on type of card.
- Resource management.

Q: How can I search for LXI devices?

A: There is an `LxiDiscovery` class in namespace `Pickering.Lxi.Communication` which is the base class if you want to search for any LXI devices. Short introduction to using the class in an example code:

```
// search for any LXI device
ReadOnlyCollection<DeviceInfo> ldis = new LxiDiscovery().BroadcastEcho(9999, 0, 4000);
// write info about every searched device
foreach (DeviceInfo ldi in ldis)
{
    WriteLxiInfo(ldi);
}
```

Q: Can I search for LXI devices from another manufacturer other than Pickering Interfaces?

A: No, this version only supports searching for LXI devices from Pickering Interfaces. According to the LXI standard manufacturers including Pickering Interfaces use VXI-11 or Zeroconf as well. Pickering Interfaces offers VXI-11 SDK as another software package or browsers with GUI for VXI-11 and Zeroconf.

Q: How can I get information about an LXI device?

A: In the previous example class `DeviceInfo` was used. This class represents information about a device (mostly for LXI devices). Example shows properties of the class:

```
Console.WriteLine("Name: \t\t\t\t {0}", ldi.Name);
Console.WriteLine("Host: \t\t\t\t {0}", ldi.Host);
Console.WriteLine("Model: \t\t\t {0}", ldi.Model);
Console.WriteLine("Firmware: \t\t\t {0}", ldi.Firmware);
Console.WriteLine("Version: \t\t\t {0}", ldi.Version);
Console.WriteLine("Lxi class: \t\t\t {0}", ldi.LxiClass);
Console.WriteLine("Serial number: \t\t {0}", ldi.SerialNumber);
Console.WriteLine("Port: \t\t\t\t {0}", ldi.RpcPort);
Console.WriteLine("Echo port: \t\t\t {0}", ldi.EchoPort);
Console.WriteLine("Clients count: \t\t {0}", ldi.ClientsCount);
Console.WriteLine("Opened cards count: \t\t {0}", ldi.OpenedCardsCount);
Console.WriteLine("Cards count: \t\t\t {0}", ldi.CardsCount);
Console.WriteLine("Description: \t\t\t {0}", ldi.Description);
```

Another way to get information is using class `LxiDeviceManagement` which is part of every `DeviceManager` or its descendants.

Q: I'm not able to find any PI LXI devices, but I have one connected. What is the problem?

A: Here are some of the more common problems:

- The LXI device is on a different sub-net – discovery only works if both client computer's default network adapter and a LXI device are on the same sub-net. Refer to the LXI Getting Started Guide shipped with your product and on the distribution disc, also make sure that the LXI device and your computer are on the same sub-net. The module supports searching for an LXI device or communication with an LXI device which is on a different sub-net as well. For instance there is another kind of `BroadcastEcho()` method which accepts the parameter `broadcastAddress`. In this case searching in a different sub-net than the computer's default sub-net means that you have to specify broadcast address of the required sub-net.
- Your firewall blocks discovery messages, two ports have to be permitted. One port which sends discovery messages (first argument in the previous example of `BroadcastEcho()` method and second port which listens to incoming responses. We use port 0. It means that operating system chooses first available. If one of the ports is blocked (or both) your discovery will be not work.
- Network bandwidth is too low and devices have not responded in the specified timeout period – you will have to increase timeout. It is the last argument of `BroadcastEcho()`.
- You have more network interfaces in the system (this applies to virtual network interfaces as well). You will have to use second overloaded method, where you can specify broadcast address for valid network interface (see above).
- Your LXI device uses another port for discovery mechanism. The current value can be read from the WWW pages of the LXI device. Default port number is 9999 for discovery and 1024 for control.

Q: How can I get information about the number of cards of a specific type such as Power Sequencer?

A: This is easy: At first you have to find the device on the network or you have to know its network address. Then you can use the `DeviceManager` class. Please connect to the device and use the method `GetCountCardType()`. Example:

```
DeviceManager dm = new DeviceManager(host);
int count = dm.GetCountCardType(CardType.StandardCard);
Console.WriteLine("Total number of PI cards is {0}", count);
```

In this example you need only know information about the network address of the LXI device (the variable `host`). At present Pickering Interfaces LXI devices only support standard cards (ie those using "pilpxi" driver in PXI) or Power Sequencer. PI doesn't support the Functional Generator card and some other types of instrument cards.

Q: How can I set exclusive access to the LXI device?

A: The class `DeviceManager` includes two methods and one informational method:

- `Lock()` - tries to get exclusive access to the LXI device. So client is able to open any card without problems.
- `UnLock()` - release exclusive access of the client.
- `IsLock()` - information if the client has exclusive access or not.

The `Lock()` method locks the device so no one is able to open any card except the owner of this global device lock. When the client finishes the connection (intentionally or by a failure) then the lock is released automatically.

Note: *In the future the signature or the set of methods for exclusive access may be changed to conform to LXI standards!*

Q: I have read about RPC port and echo port. What do these terms mean?

A: RPC port represents a TCP port for remote communication with an LXI device. Every LXI device can use another port. The default setting is 1024 for the RPC port. For PXI or PCI devices this is irrelevant. Echo port is a discovery service. It is provided only with PI LXI devices. It can be UDP (Broadcast Echo) or TCP (Direct Echo). The default value of the port is 9999. Both ports can be changed through the device's web pages.

Q: How can I see which clients are using an LXI device?

A: Class `LxiDeviceManagement` contains the method `GetConnectedClients()`. This returns instances of a `DeviceClient` class which describes connected clients by their IPv4 address and the number of occurrences. So if two clients from one computer are using an LXI device then one instance is returned where `PresentCount` of `DeviceClient` class will be set to 2.

Q: How can I see which clients are using a card?

A: Class `LxiDeviceManagement` contains the method `GetConnectedCardClients()`. This returns instances of the `DeviceClient` class which describe the client by its IPv4 address and the number of occurrences. So if two clients from one computer are using a card then one instance is returned where `PresentCount` of `DeviceClient` class will be set to 2.

Q: How can I find out how many clients are using a card or device?

A: There are two ways: First you can use the previously mentioned methods `GetConnectedCardClients()`, `GetConnectedClients()` or you can use the following:

- `GetCardSessionCount()` – returns the number of clients using the card in the device.
- `GetTotalSessionCount()` – returns the number of clients using the device.

Q: How can I change an opened card's access?

A: Sometimes the user wants to change the card access of a currently opened card. For instance the card was opened for exclusive access but the user wants make it possible for other users to access the card as well. In this case `LxiDeviceManagement` class provides the method `ChangeCardAccess()`.

Q: How can I share sessions between clients and why?

A: New Server Bridge makes it possible to open a card for exclusive access (so only the client who opened the card can use it for full access) or any client can open the card and use it for full access (multi-user access).

In some cases the scenario of multi-user and exclusive access may not be sufficient. For instance, the user wants to allow full access for two clients, other clients will only be able to read the card's status. For this reason the Server Bridge makes it possible to share sessions. So in this case a client who has exclusive access to cards can share their own session with another client (well know friend). The instructions for sharing sessions between clients are:

1. Release the session which should be shared.
2. Send (lend) the session to a client together with a security token. The token is a ticket which says that recipient can reuse the session (with all opened cards).
3. Lend (use) the session. Since the time the session is shared between two clients.
4. Work with the session.
5. Release the lent shared session (optional).

Note: *The generated token can be used only once, but when the client which wants to borrow its own session will generate new token then previously generated will be invalid even if it was not used before.*

There are three methods of `LxiDeviceManagement` class dedicated to it:

- `GetActiveSession()` - returns unique session ID of the caller. Except session ID as a number the method returns security token. The token is a ticket for any foreign client which wants use the session.
- `UseForeignSession()` - adds foreign session to the actual session. So client which adds foreign session to its own session can reuse foreign session's cards opened during all live of the session.
- `ReleaseForeignSession()` - release foreign session so that since this time the client will not be able to use cards front other client (session).

Note: *When owner of the foreign session finish work on LXI and disconnects, the foreign session will then be invalid and the session's cards will not be available for use.*

Q: How can I find out if a card is used?

A: A very easy way to find out is using the method `IsCardUsed()` from the `LxiDeviceManagement` class. This can find:

- Whether the card is used by client's own session,
- Whether the card is used some lent foreign session,
- Whether the card is used by another client (its session not lent by the caller),
- Combination of all previous possibilities.

Q: How can I find if a card is used for exclusive access?

A: This is easy too by using the `HasExclusiveAccess()` method of the `LxiDeviceManagement` class. This is able to find:

- Whether the client's own session has exclusive access to the card,
- Whether the client's lent session has exclusive access to the card,
- Whether a non client's session has exclusive access to the card,
- Combination of all previous possibilities.

4.4.4. The Piplx Module

This sub-section covers the Piplx module. This is for use with Pickering cards (Switch, Multiplexer, Multichannel multiplexer, Digital I/O, Matrix, RF matrix, Resistor, Attenuator, Power supply). The module can work with both LXI and PXI chassis.

Q: What do I need to use the Piplx module?

A: You need:

- Microsoft .NET Framework Version 2.0 or higher (can be downloaded free from: <http://www.microsoft.com/download/en/default.aspx>).
- Communication module of the Client Bridge.

Q: What do I need to use the module in my program?

A: You need to set the references to both the communication module (Pickering.Lxi.Communication.dll) and the piplex module (Pickering.Lxi.Piplx.dll).

Q: How do I connect to LXI?

A: You can use one of the constructors of the `PiplxManager` and directly specify the LXI you want connect to:

```
manager = new PiplxManager("192.168.10.117", 9999, 1024);
```

or initialize the `PiplxManager` with the default constructor and use one of the `Connect` methods:

```
manager = new PiplxManager();
manager.Connect("192.168.10.117", 9999, 1024);
```

Q: How do I connect to PXI?

A: You can use both methods above in "How do I connect to LXI" but instead of valid IP address type "null" or an empty string.

```
manager = new PiplxManager(null);
```

or

```
manager = new PiplxManager("");
```

Q: How can I use VISA resource string?

A: You can use the VISA resource string in `PiplxManager` constructors or in `Connect` methods to specify the device you want connect to. You can also specify the card you want to open during the initialization. The following example initializes the `PiplxManager`, connects to LXI and opens the card on bus 2 and device 18:

```
manager = new PiplxManager("TCPIP::192.168.10.117::2.18::INSTR");
```

Following example initializes `PiplxManager`, connects to PXI and opens the card on bus and device:

```
manager = new PiplxManager("PXI2::18::INSTR");
```

Q: How do I open a specified card?

A: `PiplxManager` during initialization loads the list of `Cards` and creates the `PiplxCARD` class for each of them. After that you can obtain the card by specifying the index and use it as follows.

```
manager.Cards[index].Open();
```

By default the `Open()` method opens the card for exclusive access (if it can be opened, otherwise an exception is generated). The user can change the access to the card by use of the input argument of the method which accepts argument of type `AccessType`.

Q: How do I get information about a specified card?**A:**

```
PiplxCardInfo info = (PiplxCardInfo)manager.Cards[index].Info;
```

The variable “info” is of the type `PiplxCardInfo`. The `PiplxCardInfo` class contains information about the card. You must explicitly cast the `Info` to `PiplxCardInfo`, because it is inherited from the parent class of the type `CardInfo`. To obtain for example card’s logical bus and device location use:

```
int bus = info.Bus;
int device = info.Device;
```

`PiplxCardInfo` also has two methods (`Status()` and `Diagnostic()`), which directly contacts the device and obtains the information at that exact time. The rest of the properties are static and are stored in the memory from the initialization. To obtain `Status` flags or `Diagnostic` string use:

```
string diagnosticString = info.Diagnostic();
CardStatuses cardStatus = info.Status();
```

Q: What is a sub-unit?

A: Standard Pickering cards are basically split into one or more sub-units. One sub-unit represents for example one Multiplexer, Matrix or Power supply. On one card there can be many different sub-units. For example the Power Supply Unit card has two sub-unit of the type Power Supply, two sub-units of the type Switch and one of the type Multiplexer.

Q: How do I get a specific sub-unit?

A: You can obtain the type of the sub-unit as follows:

```
PiplxCard card = (PiplxCard)manager.Cards[index];
OutputSubunitType subunitTypeCode =
    (OutputSubunitType)card.OutputSubunits[subunitIndex].TypeCode;
```

Each `PiplxCard` class has a list of output and input sub-units, which are loaded during initialization. You can get information such as `TypeCode` or `SubunitNumber` without opening the card.

If you want to get a sub-unit, and use the whole range of functions specific to that subunit’s type, you must explicitly cast the sub-unit. For example in the following code, we know that the sub-unit with `subunitIndex` is of the type `Matrix`:

```
MatrixSubunit subunit = (MatrixSubunit)card.OutputSubunits[subunitIndex];
```

Now you can work with the subunit and the whole set of functions and properties specific to matrix subunits. For example to obtain the number of rows and columns:

```
int cols = subunit.Columns;
int rows = subunit.Rows;
```

Q: How do I get information about a specific sub-unit?

A: You can obtain the information about a sub-unit in similar way as you obtain the information about the card. For example, we have the Power Supply sub-unit and we want to get the information about it:

```
PowerSupplySubunit subunit =
    (PowerSupplySubunit)card.OutputSubunits[subunitIndex];
PowerSupplyCapabilities capabilities =
    (PowerSupplyCapabilities)subunit.Capabilities;
double current = subunit.Current;
bool isEnabled = subunit.IsEnabled;
int precision = subunit.Precision;
double voltage = subunit.Voltage;
```

Q: How do I set a specific bit?

A: Let's say we have already obtained the card and sub-unit of the type Matrix (as in previous questions). Now we want to set a bit (specified as a `bitNumber` variable) to logical 1. But remember, that first you have to open the card! Do it as follows:

```
card.Open();
subunit.OperateBit(bitNumber, true);
```

Q: How do I prevent setting of a bit?

A: You can use one of the masking functions. In this example we'll use the `MaskBit()` function. Again we use the sub-unit of the type Matrix.

```
subunit.MaskBit(bitNumber, true);
```

Now if you try to set the bit to logic 1, the driver raises the exception; `PiplxException`, which will say "Cannot activate an output that is masked."

Q: How can I set multiple bits in a single operation?

A: You can use the `WriteSubunit()` function. There is also a similar function to mask more bits in a single operation. But you should be familiar with the bit operations. For example if we have a 64 bit sub-unit of the type Matrix, and we want to set the 1., 2., 3., 4., 5. and 41. bits:

```
int[] data = new int[2] { 0x0000001F, 0x00000100 };
subunit.WriteSubunit(data);
```

Q: How can I find the right length of the array for single operations?

A: There are three ways to solve this:

1. Use the `ViewSubunit()` method before hand and use its returned array.
2. Calculate it when you know the length of array needed to contain all the bits of the subunit.
3. `Subunit` class (parent class of switch subunit) contains the static method `GetEmptySubunitBitPattern()` which returns an empty array for the appropriate number of bits.

Q: Is it possible to work with bits with boolean values?

A: Yes it is. `Subunit` class contains helper static methods for this reason:

- `BitsToBooleanArray()` method returns an array of boolean values from a bit pattern. So you can read a sub-unit's bit pattern and transform it to array of boolean values. Working with this array is quite easy.
- `BooleabArrayToBits()` transforms a boolean array to a bit pattern. You can set the boolean values required (switches) and translate it to a bit pattern before writing the pattern to a sub-unit.

Q: Some types of operation can be done across sub-units (for instance Battery Sim card). How can I do it?

A: If the card allows it then the `PiplxCard` class contains a method for getting an extension object. Extension object makes it possible to invoke some operations over all contained subunits of dedicated type. For instance for a Battery Simulator you can use the `GetBatterySimulatorExtension()` method.

Q: I can see the module contains BIRST classes, how can I use them?

A: Yes, the module may contain BIRST classes, but the classes are only for internal using by Pickering Interfaces. There is no reason to use them because the company provides GUI tools.

4.4.5. The PI743 Module

This sub-section covers the Pi743 module. This module is for use with Pickering Power Supply card 41-743. The module can work in both LXI and PXI chassis.

Q: What do I need to use the Pi743 module?

A: You need:

- Microsoft .NET Framework Version 2.0 or higher (can be downloaded free from: <http://www.microsoft.com/download/en/default.aspx>).
- Communication module of the Client Bridge.

Q: What do I need to use the module in my program?

A: You need to set the references to both the communication module (Pickering.Lxi.Communication.dll) and the piplx module (Pickering.Lxi.Pi743.dll).

Q: How do I connect to LXI?

A: You can use one of constructors of the `PsuDeviceManager` and directly specify the LXI you want to connect to:

```
manager = new PsuDeviceManager("192.168.10.117", 9999, 1024);
```

or initialize the `PsuDeviceManager` with the default constructor and use one of the `Connect` methods:

```
manager = new PsuDeviceManager();  
manager.Connect("192.168.10.117", 9999, 1024);
```

Q: How do I connect to PXI?

A: You can use both methods above in "How do I connect to LXI" but instead of a valid IP address type "null" or empty string.

```
manager = new PsuDeviceManager(null);
```

or

```
manager = new PsuDeviceManager("");
```

Q: How can I use VISA resource string?

A: You can use the VISA resource string in `PsuDeviceManager` constructors or in the `Connect` methods to specify the device you want to connect to. You can also specify the card you want to open during the initialization. The following example initializes the manager, connects to LXI and opens the card on bus 2, device 18:

```
manager = new PsuDeviceManager("TCPIP::192.168.10.117::2.18::INSTR");
```

The following example initializes `PiplxManager`, connects to PXI and opens the card on bus 2, device 18:

```
manager = new PsuDeviceManager("PXI2::18::INSTR");
```

Q: How do I open a specified card?

A: `PsuDeviceManager` during initialization loads the list of `Cards` and creates the `PsuCard` class for each of them. After that you can obtain the card by specifying the index and use it as follows.

```
manager.Cards[index].Open();
```

By default the `Open()` method opens the card for exclusive access (if it can be opened, otherwise an exception is generated). User can change the access to the card by using the input argument of the method which accepts argument of type `AccessType`.

Q: How do I get information about a specified card?

A:

```
PsuCardInfo info = (PsuCardInfo)manager.Cards[index].Info;
```

The variable “info” is of the type `PsuCardInfo`. The `PsuCardInfo` class contains information about the card. You must explicitly cast the `Info` to `PsuCardInfo`, because it is inherited from the parent class of the type `CardInfo`. To obtain for example card’s logical bus and device location use:

```
int bus = info.Bus;  
int device = info.Device;
```

Q: How are the objects are organized?

A: PSU cards can be inserted to a PXI or a LXI chassis. Simply speaking the PXI or LXI chassis is just device manager represented by the `PsuDeviceManager` class. All contained PSU cards are available through the `Cards` property as an object of type `PsuCard` class. The card can contain one channel. A channel is represented by the `PsuChannel` class and can be obtained from the property `Channel` of `PsuCard` class. The `Channel` object contains all required methods for setting or getting voltage, current, getting status and so on. The `PsuCard` class also contains properties which can be used for calibration (`Calibration` property) purposes or triggering (`Triggering` property).

4.4.6. The Power Sequencer Module

This sub-section describes the Power Sequencer module in the form of questions and answers.

Q: Does Pickering Interfaces offer a Power Sequencer device for PXI?

A: No. Pickering Interfaces only offers the Power Sequencer as an LXI device (LXI class C). However, it is also available with a simple RS232 interface.

Q: How many types of the driver does Pickering Interfaces offer?

A: At this time only Microsoft .NET (MS.NET) and Mono are supported (for desktop or server computers). Others like .NET Compact Framework or COM are not supported. For more information contact PI support: support@pickeringtest.com

Q: How can I work with this driver?

A: It is very easy. There is a class `PowerSequencerManager` which is inherited from the `DeviceManager` class from the Communication module. So you can use any methods from the parent class.

First you have to find the right LXI device. Then connect to the device by using `PowerSequencerManager`. If everything is all right you can open the Power Sequencer card (`PowerSequencerCard` class). Now you are at the card level. Every card has input or output sub-units, the Power Sequencer has no input sub-units (`PowerSequencerSubunit` class), only output subunits – channels. At this level you can start sequence of channels, set-up time for them, etc. Example:

```
PowerSequencerManager ps = new PowerSequencerManager("192.168.10.95");
PowerSequencerCard pscard =
    (PowerSequencerCard)ps.Cards[0]; //always only one PS card

pscard.Open();

PowerSequencerSubunit pssubunit =
    pscard.OutputSubunits[0]; //PS has only one subunit
pssubunit.StartSequence(SequenceType.Stop);
    // all enabled channels are switched off

ps.Disconnect();
```

4.5 PYTHON

4.5.1 Accessing Pickering PXI Driver from Python

Pickering drivers are standard Windows DLLs and as such can be accessed from most programming languages, including Python.

To access a library, use the ctypes library, here is an example of access to the Direct I/O driver PILPXI:

At the command line:

```
from ctypes import *
pilpxi = windll.LoadLibrary("pilpxi")
ver = pilpxi.PIL_Version()
```

An implementation of the PILPXI library is provided. It contains 2 class definitions:

- ***pilpxi_base*** contains non card-specific functions to locate PXI cards
- ***pilpxi_card*** contains all the card-specific functions to access and control PXI cards

Example:

```
from pilpxi import *
base = pilpxi_base()
e, count = base.CountFreeCards()
e, bus, slot = base.FindFreeCards(count)
Cardnum = 0
while cardnum < count:
    card = pilpxi_card(bus[cardnum], slot[cardnum])
    err, cid = card.CardId()
    print "Card ", cardnum
    print "Bus", bus[cardnum], " Device", slot[cardnum]
    print "ID = ", cid
    e, ins, outs = card.EnumerateSubs()
    print "subunits: ", ins, "input, ", outs, "output"

    Sub = 1
    while sub <= outs:
        e, inf = card.SubType(sub, 1)
        print "subunit ", sub, " = ", inf
        sub = sub + 1
        cardnum = cardnum + 1
```

In this example, `pilpxi_base` is used to locate all available cards. `CountFreeCards` returns the number of cards found, then `FindFreeCards` returns two arrays containing the bus and device numbers of the cards. Next `pilpxi_card` is used to open each card and then basic card information is queried and the type of each subunit presented. For basic details of the functions available the user is referred to `pilpxi` documentation.

4.5.2 Python Library for LXI Driver

The Python Library comprises a set of functions defined in two main classes:

1. **PI_BASE:** This class contains non card-specific functions to locate PXI cards
2. **PI_CARD:** This class contains all the card specific functions to access and control Pickering PXI cards.

An Example program can be executed which connects/open card(s) and and list its ID, Number of Sub Units and Sub types.

```
from pilxi import *
base = pi_base(0,"192.168.1.200",1024,1000)
e, count = base.CountFreeCards()
e, bus, slot = base.FindFreeCards(count)
Print "-----"

cardnum = 0

while cardnum < count:
    card = pi_card(0,"192.168.1.200",1024,1000,bus[cardnum],slot[cardnum])
    err, cid = card.CardId()
    print "\nCard ",cardnum+1,
    print "Bus", bus[cardnum], " Device", slot[cardnum]
    print "ID = ", cid
    e, ins, outs = card.EnumerateSubs()
    print "subunits: ", ins, "input, ", outs, "output"
    Sub = 1
    while sub <= outs:
        e, inf = card.SubType(sub, 1)
        print "subunit ",sub, " = ", inf
        sub = sub + 1
    print "\n-----"
    cardnum = cardnum +1
```

If a connecting to PCI card or PXI chassis, use the string "PXI" instead of an IP address when entering the input arguments for the two classes.

base = pi_base(0,"PXI",1024,1000) card = pi_card(0,"PXI",1024,1000,bus[cardnum],slot[card])

In the above example, pi_base is used to locate all available cards. It CountFreeCards returns the number of cards found, then FindFreeCards returns two arrays containing the bus and device numbers of the cards.

Next, pi_card is used to open each card and then basic card information is queried and the type of each subunit presented.

4.6 LINUX

Pickering provide a port of the PILPXI driver for Linux systems; since VISA for Linux is not widely available we do not offer ports of the other drivers.

The driver is available at:

http://downloads.pickeringtest.info/downloads/drivers/Linux_Drivers/

The provided driver operates in user space and so is compatible with the majority of Linux distributions.

The distribution includes some code examples and a utility program (PILMon) which allows basic command-line access to cards.

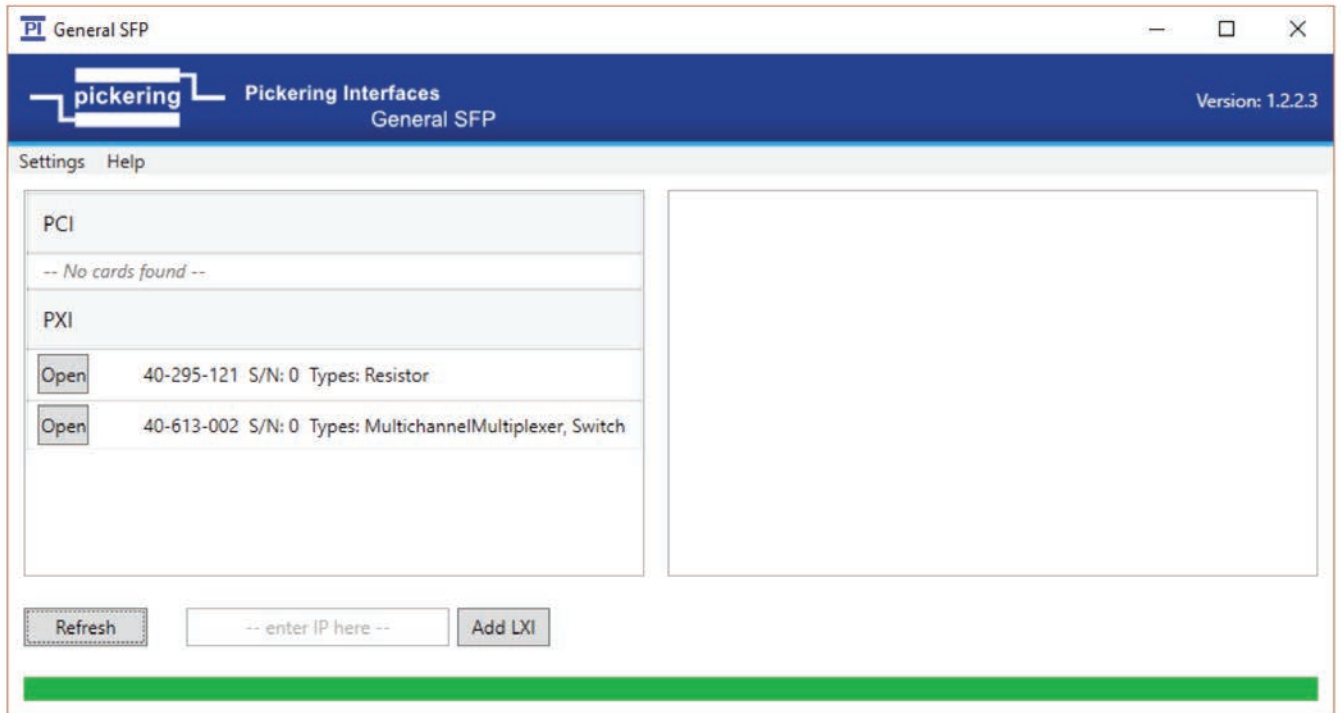
Note: The default location of libraries varies between Linux distributions. We elect to install the libraries to the generally recommended locations of `/usr/local/lib` and `/usr/local/lib64`. Should a particular distribution not include those locations in the default library path, the user should take action to correct the problem. One simple solution is to create a symbolic link in the system default folder to the libraries in `/usr/local/lib(64)`.

SECTION 5 - SUPPORT SOFTWARE

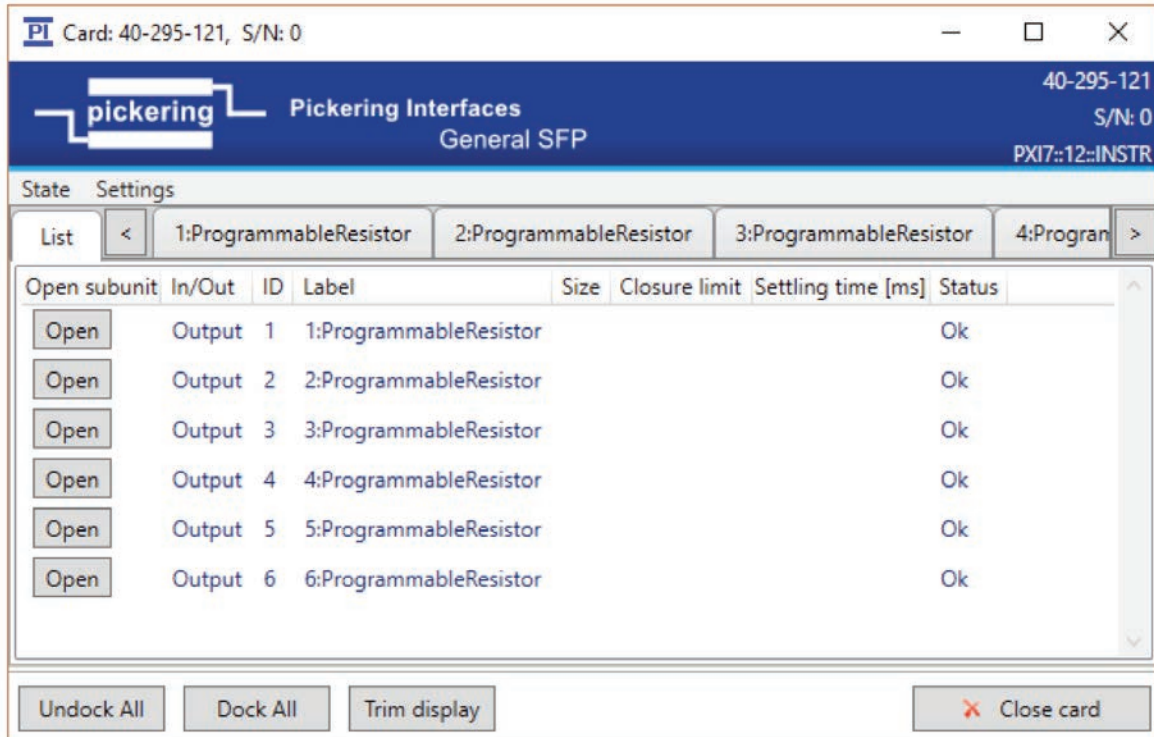
5.1 GENERAL SOFT FRONT PANEL

The General Soft Front Panel (GSFP) provides its own documentation accessible from the application panel, so detail here is kept to a minimum.

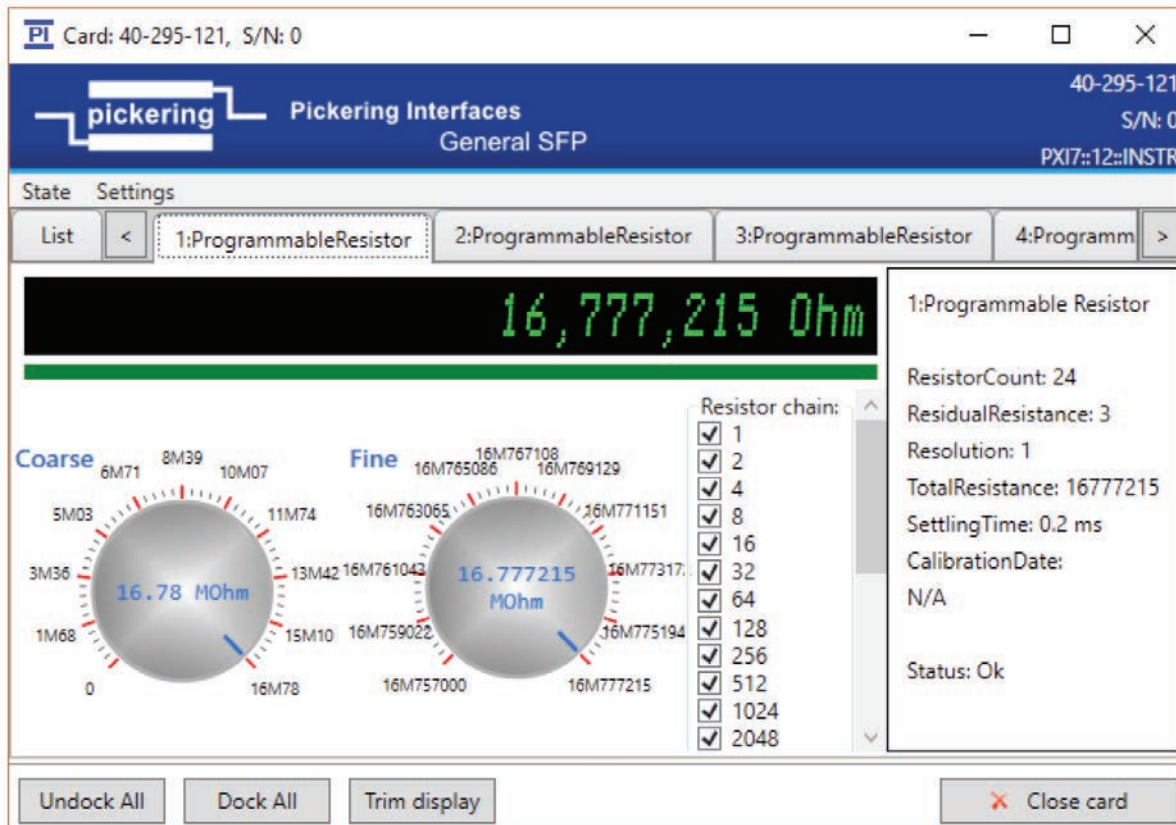
On opening the GSFP it will locate any local PCI/PXI devices present.



Clicking on the 'Open' button for a card will pop up a panel for control of that card.



In this panel clicking on 'Open' against a SubUnit entry will pop up an appropriate panel for control of that SubUnit.



5.2 IVI WIZARD

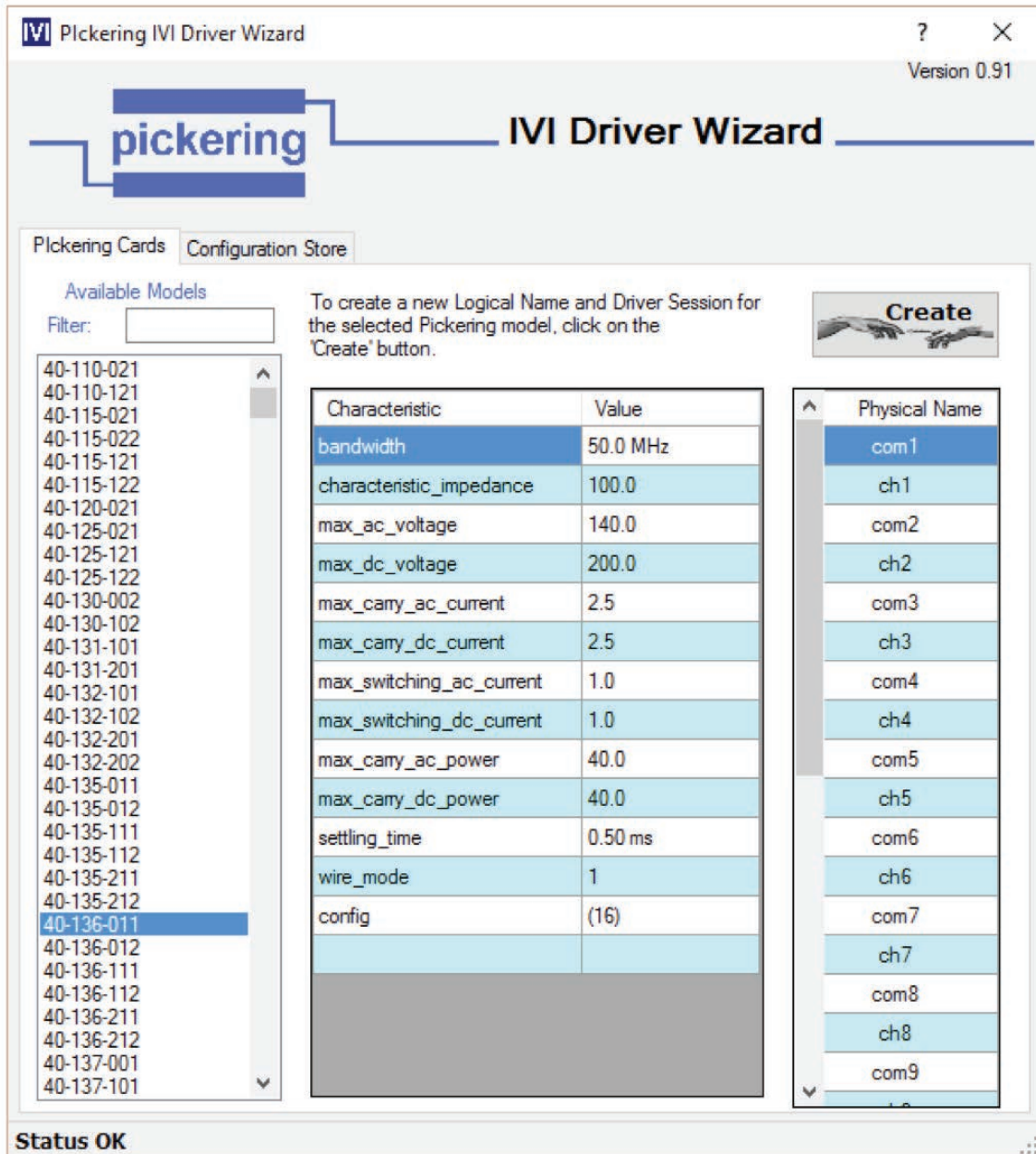
The Pickering IVI Wizard provides an easy means to set up Pickering cards for IVI use.

It can be downloaded from:

<http://downloads.pickeringtest.info/downloads/drivers/IVI/iviWizard.zip>

When executed it populates the left-hand column with a list of all cards read from the local copy of the Pickering card definition file.

A specific model can be easily found by adding the model number, or partial number, to the filter box.



When an entry in that list is selected the window panes at centre and right are populated with information about that card.

If the model you require is not present, first check for an updated definition file at:

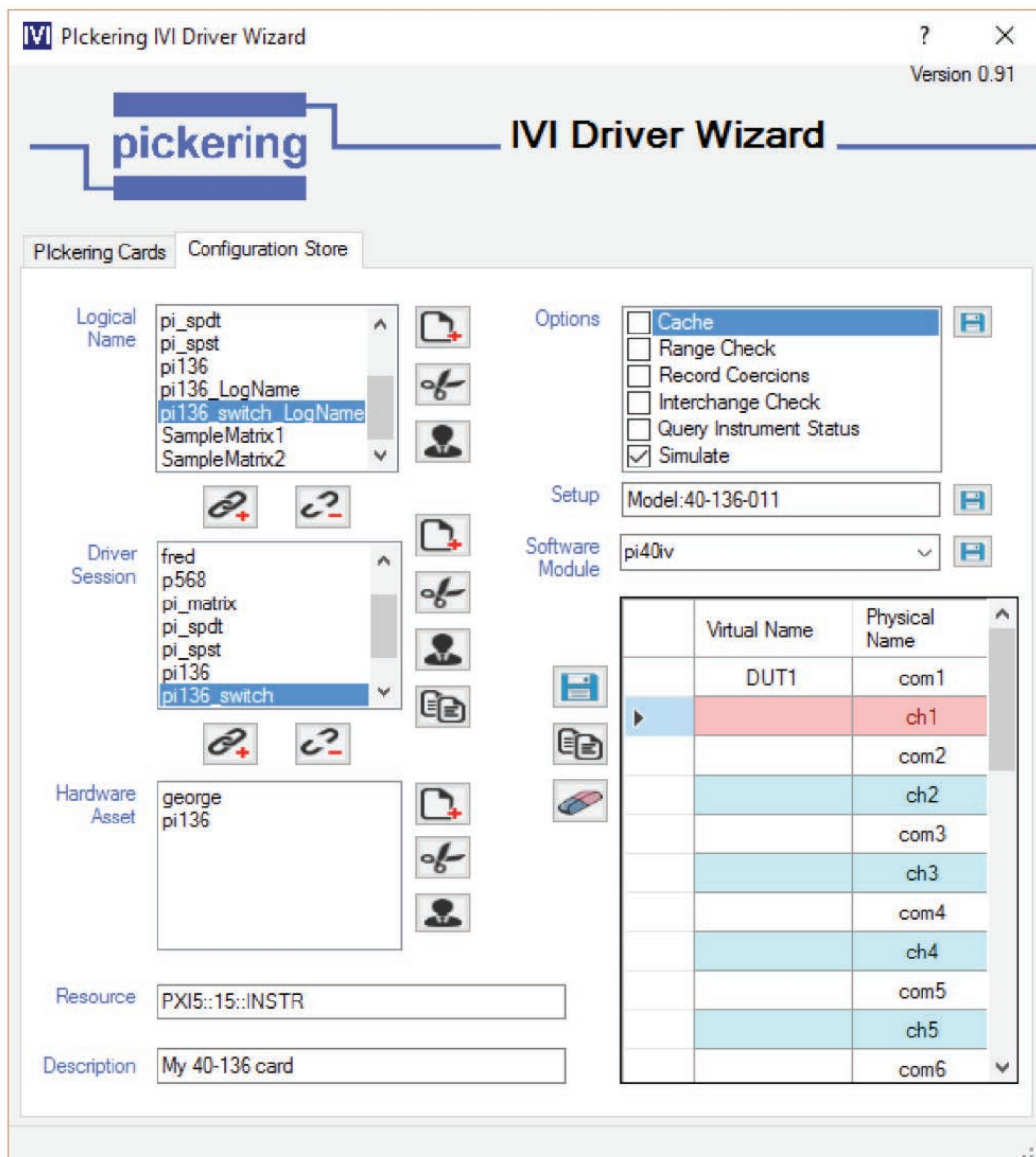
<http://downloads.pickeringtest.info/downloads/drivers/IVI/pi40iv.ini>

If still not present after updating the definition file, contact Pickering support.

To add a new entry for the selected card to the IVI Configuration Store click on the 'Create' button.

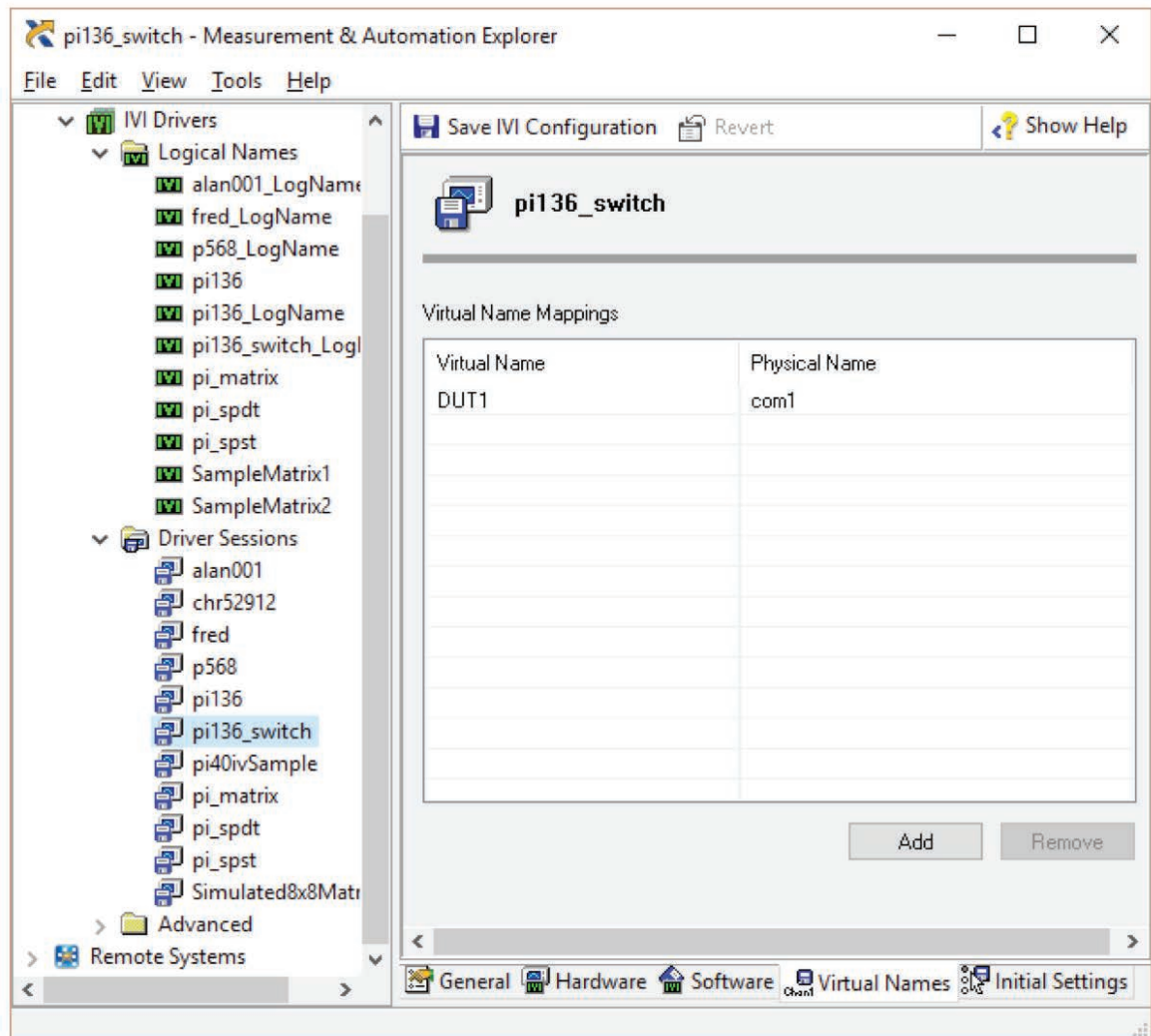
After clicking 'Create' a driver session will have been created with the name supplied and a Logical Name created linked to that driver session which may be renamed as you wish. All that remains to create a usable IVI entry is to define a hardware asset, link it to the driver session and un-check the 'Simulate' option if the card is physically present on the system.

Select the 'Configuration Store' tab and make any changes that are required.



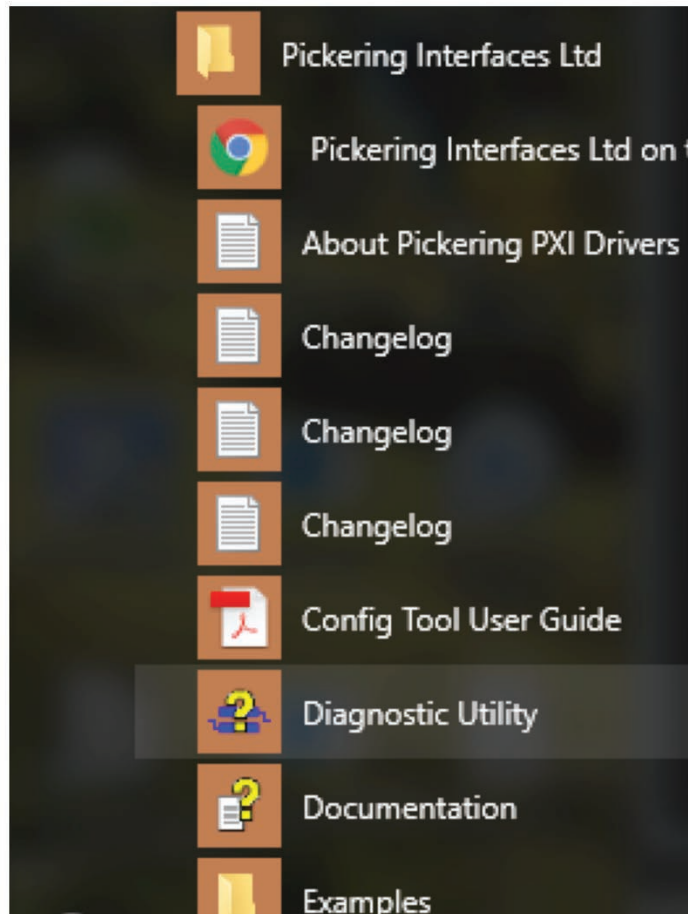
If Virtual Names are required, then enter the required names in the grid to the right. Click on the 'Save' button to commit these names to the IVI Configuration Store.

If required viewing from NI MAX will confirm that the actions have been completed.



5.3 PICKERING DIAGNOSTIC UTILITY

Once the Pickering driver is installed a Diagnostic Utility is available from the Windows Program menu.



This tool checks the hardware and software systems on the host PC and can be of help in locating problems. When it is executed it creates a text file containing the relevant information, the content of this report file is summarised below.

5.3.1 System Information

This lists the characteristics of the host PC and key file locations.

```
=====
=== System information ===
=====

*** NOTE: program is running as a 32-bit app using WOW64 in this 64-bit system

Microsoft Windows NT version 10.0 (Build 15063): Windows 10
Service Pack 0.0

BIOS date = (unobtainable)
BIOS version = "HPQOEM - 1072009"
               "A0.24"
               "American Megatrends - 5000B"

PROCESSOR_ARCHITECTURE = "x86"
NUMBER_OF_PROCESSORS = "8"
PROCESSOR_IDENTIFIER = "Intel64 Family 6 Model 94 Stepping 3, GenuineIntel"
PROCESSOR_LEVEL = "6"
PROCESSOR_REVISION = "5e03"

Processor 1 - 8:
  Vendor = "GenuineIntel"
  Identifier = "Intel64 Family 6 Model 94 Stepping 3"
  Name = "Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz"
  Clock speed = 3408MHz

SystemDrive = "C:"
SystemRoot = "C:\WINDOWS"
windir = "C:\WINDOWS"
VXIPnPPATH = "C:\Program Files (x86)\IVI Foundation\VISA\"

Scanning system INF files...
File "C:\WINDOWS\inf\oem101.inf": PICKERING - dated 25/08/2017
DriverVer=05/11/2017,1.1.1.1
CatalogFile=Pipx40q_2k.cat

Etc for all relevant INF files

Checking WinDriver system file...
File C:\WINDOWS\system32\drivers\windrvr6.sys
File version (from version resource): 11.30 built by: WinDDK

Checking WinDriver registry entries...
Registry scan completed.

Performance counter frequency = 3.328131MHz
```

The key items in this section are the check on INF file presence and the check on the location and version of the WinDriver system file used by PILPXI.

5.3.2 WMI Information

```
=====  
=== WMI information ===  
=====  
  
Connected to ROOT\CIMV2 WMI namespace  
List of PCI devices:  
PCI_BUS_0  
  PCI\VEN_8086&DEV_A121&SUBSYS_2B5E103C&REV_31\3&11583659&1&FA - Intel(R) 100  
Series/C230 Series Chipset PMC - A121 - INTEL  
  
PCI_BUS_7  
  PCI\VEN_15BC&DEV_1244&SUBSYS_00000000&REV_01\7&2C112A6E&0&78001800E0 - PCI Device  
- (null)  
  PCI\VEN_10B5&DEV_9050&SUBSYS_012A1761&REV_01\7&2C112A6E&0&70001800E0 - Pickering  
40-613-002 - Pickering Interfaces  
  PCI\VEN_10B5&DEV_9050&SUBSYS_00DF1761&REV_02\7&2C112A6E&0&60001800E0 - Pickering  
40-295-121 - Pickering Interfaces  
PCI_BUS_6  
  PCI\VEN_104C&DEV_8231&SUBSYS_72B71093&REV_03\6&9277F02&0&001800E0 - PCI Express to  
PCI/PCI-X Bridge - (Standard system devices)  
PCI_BUS_2  
  PCI\VEN_10B5&DEV_8505&SUBSYS_850510B5&REV_AA\UAA850910B5DF0E0000 - PCI Express  
Upstream Switch Port - (Standard system devices)  
PCI_BUS_3  
  PCI\VEN_10B5&DEV_8505&SUBSYS_850510B5&REV_AA\AA850910B5DF0E0008 - PCI Express  
Downstream Switch Port - (Standard system devices)  
  PCI\VEN_10B5&DEV_8505&SUBSYS_850510B5&REV_AA\AA850910B5DF0E0010 - PCI Express  
Downstream Switch Port - (Standard system devices)  
  PCI\VEN_10B5&DEV_8505&SUBSYS_850510B5&REV_AA\AA850910B5DF0E0018 - PCI Express  
Downstream Switch Port - (Standard system devices)  
  PCI\VEN_10B5&DEV_8505&SUBSYS_850510B5&REV_AA\AA850910B5DF0E0020 - PCI Express  
Downstream Switch Port - (Standard system devices)  
PCI_BUS_9  
  PCI\VEN_10EC&DEV_8168&SUBSYS_2B5E103C&REV_0C\01000000684CE00000 - Realtek PCIe GBE  
Family Controller - Realtek
```

This contains a list of all the PCI/PXI devices found.

Pickering devices will contain '1761' as Vendor or SubVendor ID.

5.3.3 WinDriver Information

```
=====
=== WinDriver information ===
=====

WinDriver version = 1130

Bus 0 Slot 0 Function 0
VendorID 8086 DeviceID 191F Rev 07 SubVendorID 103C SubsystemID 2B5E
00 191F8086 20900106 06000007 00000000 00000000 00000000 00000000 00000000
20 00000000 00000000 00000000 2B5E103C 00000000 000000E0 00000000 00000000
40 FED19001 00000000 FED10001 00000000 000001C1 00000039 8E700047 8A000001
60 E0000001 00000000 FED18001 00000000 FF000000 00000001 FF000C00 0000007F
80 00000010 00000000 0000001A 00000000 FF000001 00000001 70700001 00000002
A0 00000001 00000002 70800001 00000002 8C800001 8C000001 8A000001 8E800001
C0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
E0 01100009 62016671 960400C8 00070000 00000000 00030FC8 00000000 00000000
  Item Bus: type 5, bus number 0, slot/func 0x0
```

This provides a dump of the PCI configuration space for each device, this section is generally of little help.

5.3.4 PILPXI Information

```
=====  
=== Pilpxi information ===  
=====  
  
Pilpxi version = 424  
  
Number of cards available to be opened = 2  
Free card locations:  
  Bus 7, slot 12  
  Bus 7, slot 14  
Number of cards opened by PIL_OpenCards = 2  
  
Card 1  
  Location = bus 7, slot 12  
  ID string = 40-295-121,0,1.00  
  Input Sub-units: none  
  Output Sub-units:  
    1 = RES(24); type = 7, rows = 1, cols = 24; floating-point calibration data  
    2 = RES(24); type = 7, rows = 1, cols = 24; floating-point calibration data  
    3 = RES(24); type = 7, rows = 1, cols = 24; floating-point calibration data  
    4 = RES(24); type = 7, rows = 1, cols = 24; floating-point calibration data  
    5 = RES(24); type = 7, rows = 1, cols = 24; floating-point calibration data  
    6 = RES(24); type = 7, rows = 1, cols = 24; floating-point calibration data  
  Status = 0x00000000  
  Diagnostic info = "No errors"  
  
Card 2  
  Location = bus 7, slot 14  
  ID string = 40-613-002,0,1.02  
  Input Sub-units: none  
  Output Sub-units:  
    1 = MUXM(4); type = 3, rows = 1, cols = 4;  
    2 = MUXM(4); type = 3, rows = 1, cols = 4;  
    3 = MUXM(4); type = 3, rows = 1, cols = 4;  
  etc  
    15 = MUXM(4); type = 3, rows = 1, cols = 4;  
    16 = MUXM(4); type = 3, rows = 1, cols = 4;  
    17 = SWITCH(15); type = 1, rows = 1, cols = 15;  
  Status = 0x00000000  
  Diagnostic info = "No errors"
```

In this section each Pickering card is opened using Pilpxi and card/subunit information retrieved, also each card status and diagnostic information are retrieved.

This will indicate if any cards have problems operating with pilpx.

5.3.5 VISA Information

```
=====  
=== VISA information ===  
=====  
  
VISA specification version = 0x00500700 (5.7.0)  
VISA implementation version = 0x01100000 (17.0.0)  
Number of PXI resources found = 2  
  
Resource "PXI7::12::INSTR":  
  VI_ATTR_MANF_NAME = "Pickering Interfaces"  
  VI_ATTR_MODEL_NAME = "Pickering 40-295-121"  
    0x10B5 Vendor ID  
    0x9050 Device ID  
    0x1761 Subsystem Vendor ID  
    0x00DF Subsystem ID  
    0x1761 VI_ATTR_MANF_ID  
    0x00DF VI_ATTR_MODEL_CODE  
    ----- VI_ATTR_PXI_SUB_MANF_ID (Obsolete attribute - unsupported)  
    ----- VI_ATTR_PXI_SUB_MODEL_CODE (Obsolete attribute - unsupported)  
    0x000C VI_ATTR_PXI_DEV_NUM  
    0x0000 VI_ATTR_PXI_FUNC_NUM  
0xDEAD0001 VI_ATTR_PXI_MEM_TYPE_BAR0  
0xDEAD0000 VI_ATTR_PXI_MEM_TYPE_BAR1  
0xDEAD0001 VI_ATTR_PXI_MEM_TYPE_BAR2  
0xDEAD0000 VI_ATTR_PXI_MEM_TYPE_BAR3  
0xDEAD0000 VI_ATTR_PXI_MEM_TYPE_BAR4  
0xDEAD0000 VI_ATTR_PXI_MEM_TYPE_BAR5  
0xDF0FEF80 VI_ATTR_PXI_MEM_BASE_BAR0  
0x00000000 VI_ATTR_PXI_MEM_BASE_BAR1  
0xDF0FF800 VI_ATTR_PXI_MEM_BASE_BAR2  
0x00000000 VI_ATTR_PXI_MEM_BASE_BAR3  
0x00000000 VI_ATTR_PXI_MEM_BASE_BAR4  
0x00000000 VI_ATTR_PXI_MEM_BASE_BAR5  
0x00000080 VI_ATTR_PXI_MEM_SIZE_BAR0  
0x00000000 VI_ATTR_PXI_MEM_SIZE_BAR1  
0x00000800 VI_ATTR_PXI_MEM_SIZE_BAR2  
0x00000000 VI_ATTR_PXI_MEM_SIZE_BAR3  
0x00000000 VI_ATTR_PXI_MEM_SIZE_BAR4  
0x00000000 VI_ATTR_PXI_MEM_SIZE_BAR5
```

This section shows the version of VISA installed and access Pickering cards using basic VISA functions.

5.3.6 PIPX40 VISA Driver Information

```

=====
=== pipx40 VISA driver information ===
=====

pipx40 version = 4.24

Resource "PXI7::12::INSTR":
  Card ID = "PICKERING INTERFACES,40-295-121,0,1.00"
  Card revision = "1.00"
  Input sub-units: none
  Output sub-units:
    1 = RES(24); type = 7, rows = 1, columns = 24; floating-point calibration data
    2 = RES(24); type = 7, rows = 1, columns = 24; floating-point calibration data
    3 = RES(24); type = 7, rows = 1, columns = 24; floating-point calibration data
    4 = RES(24); type = 7, rows = 1, columns = 24; floating-point calibration data
    5 = RES(24); type = 7, rows = 1, columns = 24; floating-point calibration data
    6 = RES(24); type = 7, rows = 1, columns = 24; floating-point calibration data
  Card status = 0x00000000
  Diagnostic info = "No errors"
  Selftest result = 0, message = "OK"
  Card reset OK

```

In this section the pipx40 driver version then each Pickering card is opened using the pipx40 driver to obtain card and subunit information and status.

5.3.7 IVI and ClientBridge Information

```

=====
=== pi40iv IVI driver information ===
=====

Pi40iv32 library:
Driver: pi40iv 4.14, Core driver: unavailable, Compiler: MSVC 12.00, Components:
IVIEngine 15.00, VISA-Spec 5.40, build: client bridge call build

=====
=== PI ClientBridge information ===
=====

Picmlx library: version 186
Piplx library: version 170

```

The final sections query the versions of other Pickering drivers but makes no attempt to open cards with them.

SECTION 6 - USEFUL INFORMATION

6.1 IVI INTERCHANGEABILITY

While it is possible to embed a specific driver into your code, the full power of the IVI driver concept is only fully realized if the code is written in a way to make the code allow cards to be interchanged from one card to another without having to recompile the application.

First, make suitable definitions of the card in MAX. Define a Driver Session referring to the specific card to be controlled and make a Logical Name referring to the Driver Session, see the sections. Alternatively, use the Pickering IVI Wizard (see later section) to create these entries.

This code segments below show 3 possible ways in which the card can be initialized.

6.1.1 Method 1 – No Interchangeability

```
// This offers NO interchangeability since:
// a) The specific driver is coded
// b) The specific card model is called

    err = pi40iv_InitWithOptions(    "PXI5::15::INSTR",
        VI_FALSE,
        VI_FALSE,
        "Simulate=0,RangeCheck=1,QueryInstrStatus=1,Cache=1,DriverSetup=Mo
del:41-182-003;",
        &vi);
    if (err != VI_SUCCESS)
    {
        pi40iv_error_message(0, err, msg);
        printf("%s\n", msg);
    }
    else
    {
        pi40iv_close(vi);
    }
```

This approach offers no interchangeability; the address and the model number are embedded in the code and the Pickering specific driver is being used. Any change to the card address or model would require the code to be re-built.

6.1.2 Method 2 – Interchangeability with Other Pickering Cards

```
// In this case the card model is dealt with in the IVI configuration store
// Which means the card could be re-defined in the store
// However, the Pickering driver is still explicitly called

err = pi40iv_init("atten_ln", 0, 0, &vi);
if (err != VI_SUCCESS)
{
    pi40iv_error_message(0, err, msg);
    printf("%s\n", msg);
}
else
{
    pi40iv_close(vi);
}
```

This approach offers some interchangeability; the address and the model number are set in the IVI Configuration Store and the card is accessed by the Logical Name assigned to the card. If the card address or model were to change, a simple edit of the Configuration Store would allow a program written with this code to continue without re-compilation. However, since the Pickering specific driver is being used, only another Pickering card could be substituted.

6.1.3 Method 3 – Fully Interchangeable (almost)

```
// In this case the generic class driver is used
// The driver session in the IVI configuration store specifies the address,
// the model and the driver to use
// This is fully interchangeable by editing the configuration store

err = IviSwtch_init("atten_ln", 0, 0, &vi);
if (err != VI_SUCCESS)
{
    IviSwtch_error_message(0, err, msg);
    printf("%s\n", msg);
}
else
{
    IviSwtch_close(vi);
}
```

This approach offers near full interchangeability; all aspects of the card are set in the IVI Configuration Store, the card is accessed by the Logical Name assigned to the card. If the card address or model were to change, a simple edit of the Configuration Store would allow a program written with this code to continue without re-compilation. This includes the possibility to change to a different manufacture of card using a different driver.

However, there remains an obstacle, that of channel names. Different manufactures can and do use different naming conventions for the end-points of the switching network. For example, the Pickering driver allocates channel names 'base 1' that is ch1, ch2, ch3 etc whereas National Instruments allocate 'base 0', so ch0, ch1, ch2.

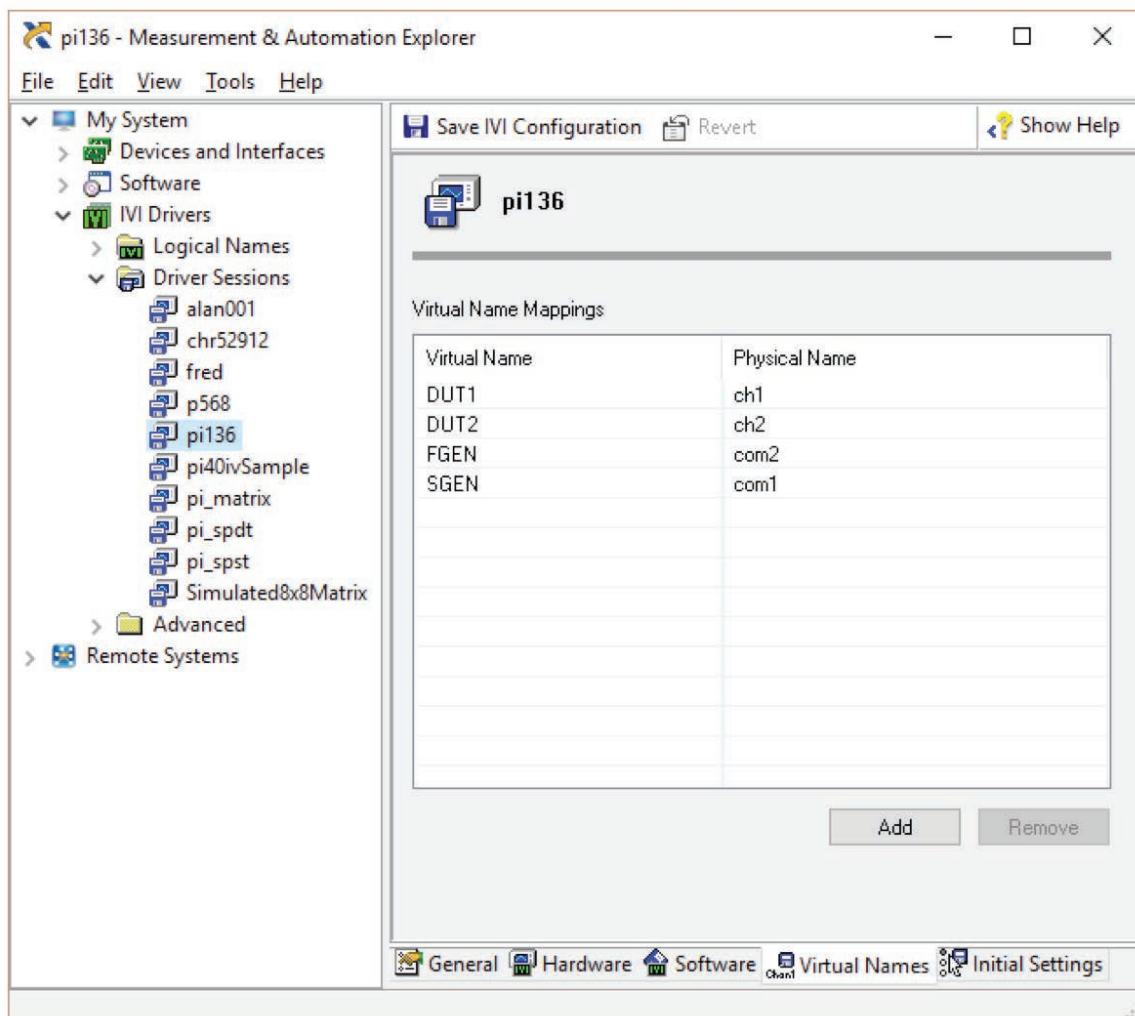
In order to have complete interchangeability these differences in channel naming must also be handled.

6.2 CHANNEL NAMES

To make an application that can be truly interchangeable, differences in channel names between cards must also be handled and the IVI Configuration Store provides a mechanism to do that.

Within the definition of a specific card is an optional table of Virtual Names. By applying user defined names to each channel, the application can become fully interchangeable. In addition the use of user defined Virtual Names can also render the application code considerably more meaningful since instead of obscure references to 'COM1' and 'CH1' the user can create channel names relevant to the application, such as 'DUT1' or 'DMM'.

Below is a screen shot of NI-MAX showing some Virtual Names applied to the channels of a pair of simple switches. Internally the Pickering driver recognises the channels as 'com1', 'ch1', 'com2' and 'ch2'; the IVI Configuration Store contains a table of Virtual Names which allow the user to refer to those channels by application relevant names 'DUT1', 'DUT2', 'FGEN' and 'SGEN'.



It must be said that NI MAX does not make the task of generating Virtual Names easy, the user must have prior knowledge of the driver internal channel names and MAX does not assist in that task.

See the Pickering IVI Wizard for an easier way to set Virtual Names.

THIS PAGE INTENTIONALLY BLANK

SECTION 7 - SHARED MEMORY

7.1 SHARED MEMORY vs PXI DRIVER

The Pickering PXI driver helps the customer to control the card in a PXI chassis. But in this type of driver, single application has access to the card at a point.

Example:

Console application created by the customer using Pickering driver API will be able to access and control the PXI card, but the GSFP won't be able to access the same PXI card to monitor the state of PXI card functionality.

If customer needs access to the PXI card from multiple applications (e.g. General Soft Front Panel, Switch Path Manager) then Pickering shared memory driver should be used.

Example:

The console application can control and access the card, but GSFP at the same time monitor the card functionality along with the console application.

An application that has been created using the API functions of the Pickering PXI driver will work with shared memory driver without any code changes.

Shared Memory driver execution time will be marginally slower when compared to Normal PXI driver due to additional logic execution time.

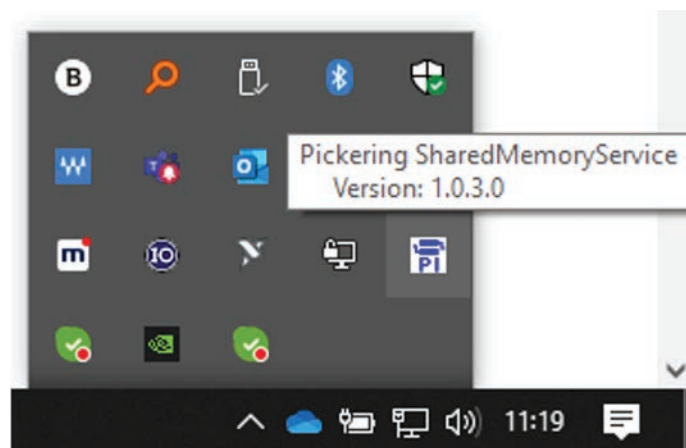
7.2 INSTALLATION INSTRUCTIONS

Please download the shared memory driver using the option "Direct IO and VISA with Shared Memory Service (32 and 64-Bit)" from the link https://downloads.pickeringtest.info/downloads/drivers/PXI_Drivers/

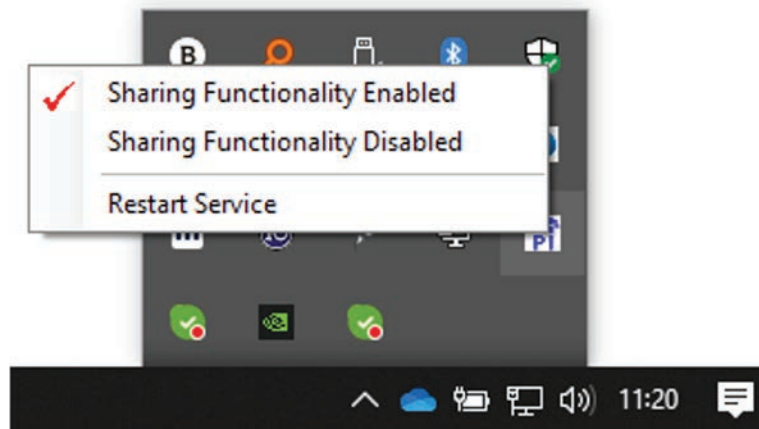
7.3 SHARED MEMORY SERVICE - TIPS TO KEEP IN MIND

a. Activating and De-activating Shared Memory Service

The shared memory service can be restarted, enabled, disabled using the Pickering shared memory tray icon:



Right click on the tray icon to get the options to restart, enable and disable the shared memory service:



b. Adding New Card to Shared Memory

When a PXI card is added to the PXI chassis on the system, after restart the new PXI card will be detected by shared memory service automatically (as long as the shared memory service is enabled).

If you are using a thunderbolt connectivity between your chassis and host PC:

After the PXI card is added to chassis and you are not restarting the PC, then please restart the shared memory service using the tray icon so that the new card is detected correctly by shared memory.

c. PXI Simulator and Shared Memory

When you add a new card to the PXI Simulator. Please make sure all the applications that are linked to the PXI driver (for example General Soft Front Panel, LabVIEW or custom applications using PILPXI or PIPX40 API functions) is closed and the shared memory service is restarted.

d. Opening Cards While Using Shared Memory Driver

When shared memory driver is used, as card is opened using respective API functions for use in an application the previous state of the card will not be cleared.

(For Non-shared memory driver every time a card is opened, the state of the card will be cleared)

Depending on the application if the state of the card must be cleared to initial known state while using shared memory driver:

a. Direct IO driver (PILPXI)

Execute the `PIL_ClearCard()` function after `PIL_OpenSpecifiedCard()` or `PIL_OpenCards()` function.

b. VISA based driver (PIPX40)

Please use "reset_instr" parameter of the `pipx40_init` function.

SECTION 8 - PROTECTION RESET UTILITY

8.1 PROTECTION RESET OVERVIEW

This driver feature provides a means of automated card reset on the server side (LXI Controller) if a user defined time has elapsed.

The purpose is to protect interconnected UUTs to the cards by resetting them in case of failing or freezing control applications, freezing PCs or broken Ethernet links.

There are two functions available:

- PICMLX_ProtectionResetEnable(SessionID, Bus, Device, CardsToProtect, Timeout)
- PICMLX_ProtectionResetDisable(SessionID)

With the ProtectionResetEnable() and the given timeout a timer on the server side gets started. If this function is not called again within that set timeout a reset on the server side takes place.

After an active server side reset any consecutive ProtectionResetEnable call returns an error.
error = ER_PICMLX_SRV_PROTECTIONRESET_PERFORMED = "Protection Reset was performed"

Call ProtectionResetDisable() to clear the error or disable the timer.

Note:

- VX Instrument cards (from APIs LXI_PXM78xx_*, LXI_PX773x_*, LXI_PXA72xx) and 41-743-XXX and 41-620-XXX cards are excluded from the above feature.
- If CardsToProtect == 0 then it does not matter which value you have defined in the bus and device arrays, all cards are reset
If CardsToProtect >0 then bus and device numbers in the arrays have to be valid, and only the valid ones are reset

Function Declaration (C API)

<summary>

Enable protection reset feature. If the timer will exceed (no subsequent call was obtained to reset the timer) all or dedicated cards will be reset. If Bus number>0 and Device number> only the cards with this valid address will be reset.

<param name="SID" >Handle of current session

<param name="Bus" >The array of card's logical bus location.

<param name="Device" >The array of card's logical device location.

<param name="CardsToProtect">The number of cards to protect. Note: If the value is set to 0, all cards will be protected by reset. CardsToProtect should match the length of Bus/Device array.

</param>

<param name="Timeout" >Timeout in seconds. When timeout exceeds all cards will be reset. Min value: 1, Max value: 60

<returns>Zero for success or non-zero error code for failure

DWORD PICMLX_API PICMLX_ProtectionResetEnable(SESSION SID, DWORD Bus[], DWORD Device[], DWORD CardsToProtect, DWORD Timeout);

<summary>

Disables protection reset feature and/or clears error ER_PICMLX_SRV_PROTECTIONRESET_PERFORMED

<param name="SID">Handle of current session.</param>

<returns>Zero for success or non-zero error code for failure.</returns>

DWORD PICMLX_API PICMLX_ProtectionResetDisable(SESSION SID);

THIS PAGE INTENTIONALLY BLANK