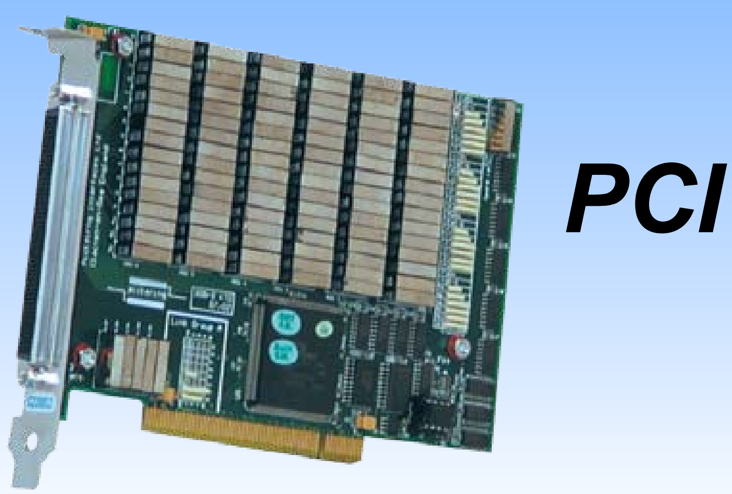
PICKERING INTERFACES

Programming Manual

SYSTEM 40 and 45 PXI SWITCHING MODULES,
SYSTEM 50 PCI SWITCHING MODULES &
SYSTEM 41 PXI INSTRUMENTS (SELECTED MODELS)



www.pickeringtest.com

pickering

ISO 9002
Reg No. FM38792

Pickering Interfaces PXI Programming Manual

for Switching Cards:
System 40, System 45 and System 50

and Instrument Cards:
System 41 (selected models)

Version Date: 08 Oct 2009

Pipx40/Pilpxi driver version: 3.10

Programming options for Pickering Interfaces PXI Cards

Software drivers are supplied for Microsoft Windows 2000/XP operating systems, with specific support for the following development environments:

- Microsoft Visual Basic
- Microsoft Visual C++
- Borland C++
- National Instruments LabWindows/CVI
- National Instruments LabVIEW and LabVIEW RT

Windows drivers are supplied in the form of Dynamic Link Libraries, which should also be usable in any other development environment that supports them.

Three different Windows drivers are available to meet particular system requirements, and should none of these be suitable there is also the option of register-level programming. Drivers are generally 'universal', handling all models in the System 40, 45 and 50 ranges; however some models that are not compliant with the the Iviswtch class cannot be used with the pi40iv IVI driver. The pipx40 and Pilpxi drivers are also applicable to certain models in the System 41 (PXI Instruments) range - see these drivers' System 41 support lists.

Please note that this documentation is available in its most up-to-date form as HTML help files, fully hyperlinked for easy access - both pipx40 and Pilpxi documents are included in the Pipx40vpp software installation.

IVI Driver for Windows - pi40iv
The pi40iv IVI (Interchangeable Virtual Instrument) driver supports all Pickering Interfaces PXI switch cards that are consistent with the Iviswtch class model - as are the great majority of cards in the System 40/45/50 ranges. Based on VISA (Virtual Instrument Software Architecture) it integrates well with LabWindows/CVI and LabVIEW, and is fully compatible with Switch Executive. Provided VISA is available, it is also usable in general-purpose programming environments such as Visual C++ and Visual Basic. Programming information for this driver is not currently included in this manual - please consult separate documentation.

VISA Driver for Windows - pipx40
The pipx40 driver conforms to the VISA (Virtual Instrument Software Architecture) standard for programmable instrumentation. Instrument control environments such as LabVIEW and LabWindows/CVI are based on VISA, and pipx40 support libraries are provided for them. Where VISA is available, pipx40 can also be used in general-purpose programming environments such as Visual C++ and Visual Basic. When IVI is not a system requirement this driver will often yield faster operation than the pi40iv driver.

Direct I/O Driver for Windows - Pilpxi
The Pilpxi driver accesses cards directly, without using the VISA software layer, while offering similar overall functionality to pipx40. It is most commonly used in general-purpose programming environments such as Visual C++ and Visual Basic. Operating speed of the VISA and Direct I/O drivers is generally comparable.

Register-level Programming
Where the supplied drivers are not suitable, register-level programming can be employed - for example:

- if the functionality of the supplied drivers does not meet the application requirements
- if security considerations demand full source-code for the application
- in development environments that have alternate mechanisms for accessing PCIbus
- for operating systems other than Windows

Section 1: PXI VISA Driver - pipx40

Section 2: PXI Direct I/O Driver - Pilpxi

Section 3: Register-level Programming

Pickering Interfaces PXI VISA Driver - pipx40

Table Of Contents

Pickering Interfaces PXI VISA Driver - pipx40

Pickering Interfaces PXI VISA Driver - pipx40

This document describes programming support for Pickering Interfaces PXI cards using the pipx40 VISA (Virtual Instrument Software Architecture) software driver which is applicable to the following families of switching cards:

- System 40 (3U PXI)
- System 45 (6U PXI)
- System 50 (PCI)

Certain System 41 (PXI Instrument) cards are also supported - for models see the System 41 Support List.

Formerly maintained by the VXI*plug&play* Systems Alliance, information on VISA and its specifications is now obtainable through the IVI Foundation http://ivifoundation.org.

The target framework for this driver is WIN32, providing 32 bit application support for Microsoft Windows® versions 2000/XP.

The driver requires **NI-VISA version 4.1 or greater** to be installed on the host system for its operation (VISA version 4.0.0, implementation 4.1.0). The pipx40 driver installation contains the core driver DLLs, support files, help files, sample programs and the Test Panel software.

System 40/45/50 cards offer a wide range of Relay Switching and Digital Input-Output functions in PXI, CompactPCI and PCI formats.

Version date: 08 Oct 2009

pipx40 driver version: 3.10

pipx40 VISA Driver Basics

The pipx40 driver requires the VISA (Virtual Instrument Software Architecture) software layer to have been installed. The driver is implemented in Dynamic Link Library pipx40_32.dll, together with library/header files for each supported programming environment.

pipx40 install location

The install location for pipx40 driver files should default to the appropriate VISA folder, using information obtained from the Windows registry. For legacy VXI*plug&play* installations it is likely to be "C:\VXIPNP\WinNT", while more recent IVI foundation installations will usually be in "C:\Program Files\IVI Foundation\VISA\WinNT".

Alternative drivers

The **Pilpxi** Direct I/O (kernel) driver is also available, giving broadly similar functionality to pipx40 while being independent of the VISA software layer.

A driver compliant with the IVI (Interchangeable Virtual Instruments) standard, **pi40iv**, is also available.

Accessing Cards

Resource names

The VISA resource name supplied to pipx40_init specifies the Pickering card to be opened. A typical example might be "PXI0::15::INSTR", which targets the card at the logical location PXI bus = 0, slot = 15. The VISA environment commonly allows the setting up of aliases, so that for example the name "SWITCH_CARD" can be assigned to represent "PXI0::15::INSTR". A program can then open the card using the resource name "SWITCH_CARD" and if the card's location is changed subsequently it is only necessary to alter the alias, instead of editing the program.

Instrument handles

When a card is successfully opened by pipx40_init, VISA returns an Instrument handle associated with the card. This handle is then used as necessary to specify the card in other function calls.

Sub-units

Pickering PXI cards contain one or more independently addressable functional blocks, or sub-units. Sub-unit numbers begin at 1, and separate sequences are used for input and output functions. This number is used in function calls to access the appropriate block. Generally, sub-unit numbers correspond directly to the bank numbers specified in hardware documentation.

Sub-unit examples:

Model	Configuration	INPUT sub-unit #1	OUTPUT sub-unit #1	OUTPUT sub-unit #2	OUTPUT sub-unit #3
40-110-021	16 SPDT switches	None	16 SPDT switches	None	None
40-290-121	Dual Programmable resistors + 16 SPDT switches	None	Resistor #1	Resistor #2	16 SPDT switches
40-490-001	Digital I/O	16-channel inputs	32-channel outputs	None	None
40-511-021	Dual 12 x 4 matrix	None	12 x 4 matrix #1	12 x 4 matrix #2	None

Sub-unit characteristics

The numbers of input and output sub-units in a card can be obtained using function pipx40_getSubCounts.

Sub-unit type and dimensions can be obtained using functions:

pipx40_getSubType - as a text string

pipx40_getSubInfo - in numerical format

pipx40_getSubType type desc.	**pipx40_getSubInfo type value**	**Characteristics**
INPUT	1	Digital inputs.
SWITCH	1 - pipx40_TYPE_SW	Uncommitted switches. Switches can be selected in any arbitrary pattern.
MUX	2 - pipx40_TYPE_MUX	Multiplexer, single channel. Only one channel can be selected at any time.
MUXM	3 - pipx40_TYPE_MUXM	Multiplexer, multi channel. Any number of channels can be selected simultaneously.
MATRIX	4 - pipx40_TYPE_MAT	Matrix, LF. Multiple crosspoints may be closed on any row or column, though there may be a limit on the total number that can be closed simultaneously. Some matrices intended for RF use are also characterised as this type, though closure of multiple crosspoints on a row or column will inevitably compromise RF performance.
MATRIXR	5 - pipx40_TYPE_MATR	Matrix, RF. A matrix intended for RF use, generally permitting the closure of only one crosspoint on each row and column.
DIGITAL	6 - pipx40_TYPE_DIG	Digital outputs. Outputs can be energised in any arbitrary pattern.
RES	7 - pipx40_TYPE_RES	Programmable resistor.
ATTEN	8 - pipx40_TYPE_ATTEN	Programmable RF attenuator.
PSUDC	9 - pipx40_TYPE_PSUDC	DC power supply.
BATT	10 - pipx40_TYPE_BATT	Battery Simulator.

Data Formats

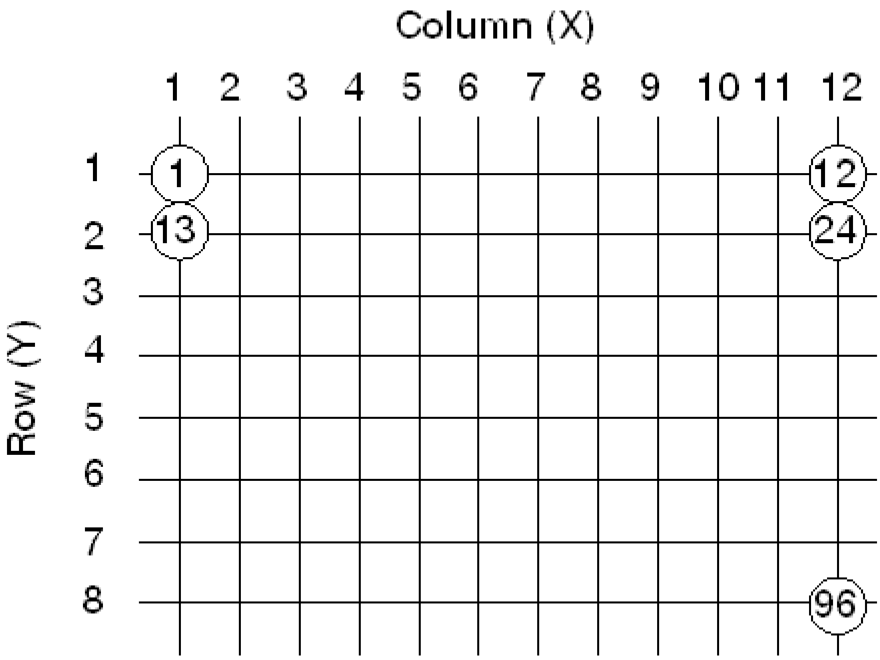
Two basic data formats are used by the driver.

Channel Number

The individual output to be affected by functions such as pipx40_setChannelState is specified by a channel number.

For any sub-unit type other than a matrix, this **unity-based** number directly specifies the affected output channel.

For a matrix sub-unit, the channel number of a crosspoint is determined by folding on the row-axis. For example in a MATRIX(12X8), having 12 columns and 8 rows, channel number 13 represents the crosspoint (row 2, column 1):



Note: matrix operation

More straightforward matrix operation using row/column co-ordinates is provided by functions:

pipx40_setCrosspointState

pipx40_getCrosspointState

pipx40_setCrosspointMask

pipx40_getCrosspointMask

Pattern Array

Functions affecting all of a sub-unit's channels utilise a one-dimensional data array (or vector) of 32-bit (unsigned) longwords. In the array, each bit represents the state of one output channel: '0' for OFF, '1' for ON. The least significant bit in the base element of the array corresponds to channel 1, with more significant bits corresponding to higher-numbered channels.

The minimum number of longwords needed to represent a sub-unit is the integer part of:

((rows * columns) + 31) / 32

For a matrix sub-unit, bit assignments follow the same method as that used to determine channel numbers. Hence for the matrix example above:

Element 0 bit 0 = row 1 column 1

Element 0 bit 11 = row 1 column 12

Element 0 bit 12 = row 2 column 1

Element 2 bit 31 = row 8 column 12

This format is employed by functions:

pipx40_setChannelPattern

pipx40_getChannelPattern

pipx40_setMaskPattern

pipx40_getMaskPattern

pipx40_readInputPattern

Timing Issues

Default mode

In the default mode of operation, driver functions incorporate appropriate delay periods to guarantee safe sequencing of internal events and that switch states will have stabilised prior to returning (fully debounced operation).

Break-before-make action is enforced for all operations, including pattern based functions such as pipx40_setChannelPattern.

No-wait mode

If the option pipx40_MODE_NO_WAIT is invoked using pipx40_setDriverMode all sequencing and settling delays are disabled. This allows other operations to proceed while switches are transitioning - the debounce period for a microwave or high power switch may be 15 milliseconds or more. A sub-unit's debounce period can be discovered using pipx40_getSettlingTime.

It should be borne in mind that for some models the elimination of internal sequencing delays could result in transient illicit states.

When pipx40_MODE_NO_WAIT is set stabilisation of a sub-unit's switches can be determined by polling the result of pipx40_getSubStatus; or stabilisation of all switches on a card by polling with pipx40_getCardStatus. In either case stabilisation is indicated by the pipx40_STAT_BUSY bit being clear.

Function Tree Layout

--VISA Standard Functions--	
Initialise	
Initialise a Pickering card	pipx40_init
Utility	
Convert a numeric error code to a message	pipx40_error_message
Error Query	pipx40_error_query
Reset a card	pipx40_reset
Card driver/firmware revision query	pipx40_revision_query
Self-test a card	pipx40_self_test
Close	
Close a Pickering card	pipx40_close
--Card Specific Functions--	
Information and Status	
Get card ID	pipx40_getCardId
Get card status	pipx40_getCardStatus
Get closure limit	pipx40_getClosureLimit
Get diagnostic information	pipx40_getDiagnostic
Get settling time	pipx40_getSettlingTime
Get card sub-unit counts	pipx40_getSubCounts
Get sub-unit description (numeric format)	pipx40_getSubInfo
Get sub-unit status	pipx40_getSubStatus
Get sub-unit description (string format)	pipx40_getSubType
Switching and General Purpose Output	
Clear all channels of a card	pipx40_clearCard
Clear all channels of a sub-unit	pipx40_clearSub
Get a sub-unit's channel pattern	pipx40_getChannelPattern
Get the state of a single channel	pipx40_getChannelState
Set a sub-unit's channel pattern	pipx40_setChannelPattern
Turn on/off a single channel	pipx40_setChannelState
Specialised Switching	
Get the state of a matrix crosspoint	pipx40_getCrosspointState
Get sub-unit attribute value	pipx40_getSubAttribute
Operate a switch - specialised	pipx40_operateSwitch

Turn on/off of a matrix crosspoint	pipx40_setCrosspointState
Switch Masking	
Clear a sub-unit's mask	pipx40_clearMask
Get the mask state of a matrix crosspoint	pipx40_getCrosspointMask
Get a sub-unit's mask pattern	pipx40_getMaskPattern
Get the mask state of a single channel	pipx40_getMaskState
Mask/unmask a matrix crosspoint	pipx40_setCrosspointMask
Set a sub-unit's mask pattern	pipx40_setMaskPattern
Mask/unmask a single channel	pipx40_setMaskState
Input	
Read a sub-unit's input pattern	pipx40_readInputPattern
Read the state of a single input	pipx40_readInputState
Calibration	
Read an integer calibration value	pipx40_readCalibration
Read a sub-unit's calibration date	pipx40_readCalibrationDate
Read floating-point calibration value(s)	pipx40_readCalibrationFP
Set calibration point	pipx40_setCalibrationPoint
Write an integer calibration value	pipx40_writeCalibration
Write a sub-unit's calibration date	pipx40_writeCalibrationDate
Write floating-point calibration value(s)	pipx40_writeCalibrationFP
Programmable Resistor	
Get resistor information	pipx40_resGetInfo
Get the current resistance setting	pipx40_resGetResistance
Set the resistance value	pipx40_resSetResistance
Programmable RF Attenuator	
Get the current attenuation setting	pipx40_attenGetAttenuation
Get attenuator information (numeric format)	pipx40_attenGetInfo
Get an individual pad's attenuation value	pipx40_attenGetPadValue
Get attenuator description (string format)	pipx40_attenGetType
Set the attenuation value	pipx40_attenSetAttenuation
Power Supplies	
Enable or disable a power supply's output	pipx40_psuEnable
Get power supply information (numeric format)	pipx40_psuGetInfo

Get power supply description (string format)	pipx40_psuGetType
Get power supply output voltage setting	pipx40_psuGetVoltage
Set power supply output voltage	pipx40_psuSetVoltage
Battery Simulator	
Set voltage	pipx40_battSetVoltage
Get voltage	pipx40_battGetVoltage
Set current	pipx40_battSetCurrent
Get current	pipx40_battGetCurrent
Set enable	pipx40_battSetEnable
Get enable	pipx40_battGetEnable
Read interlock state	pipx40_battReadInterlockState
Mode control	
Set driver operating mode	pipx40_setDriverMode

Error Codes

Driver functions return a status code that indicates success or failure of the function call. A status code of zero (VI_SUCCESS) indicates success.

Driver-specific error codes are as follows:

Driver constant	Hexadecimal value	Description
pipx40_ERROR_BAD_SESSION	BFFC0800	No Pickering card is open on the session specified
pipx40_ERROR_NO_INFO	BFFC0801	Cannot obtain information for specified card
pipx40_ERROR_CARD_DISABLED	BFFC0802	Specified card is disabled
pipx40_ERROR_BAD_SUB	BFFC0803	Sub-unit value out-of-range for target card
pipx40_ERROR_BAD_CHANNEL	BFFC0804	Channel number out-of-range for target sub-unit
pipx40_ERROR_NO_CAL_DATA	BFFC0805	Target sub-unit has no calibration data to read/write
pipx40_ERROR_BAD_ARRAY	BFFC0806	SafeArray type, shape or size is incorrect
pipx40_ERROR_MUX_ILLEGAL	BFFC0807	Non-zero write data value is illegal for MUX sub-unit
pipx40_ERROR_EXCESS_CLOSURE	BFFC0808	Execution would cause closure limit to be exceeded
pipx40_ERROR_ILLEGAL_MASK	BFFC0809	One or more of the specified channels cannot be masked
pipx40_ERROR_OUTPUT_MASKED	BFFC080A	Cannot activate an output that is masked
pipx40_ERROR_FAILED_INIT	BFFC080B	Cannot open a Pickering card at the specified location
pipx40_ERROR_READ_FAIL	BFFC080C	Failed read from hardware
pipx40_ERROR_WRITE_FAIL	BFFC080D	Failed write to hardware
pipx40_ERROR_VISA_OP	BFFC080E	VISA operation failure
pipx40_ERROR_VISA_VERSION	BFFC080F	Incompatible VISA version
pipx40_ERROR_SUB_TYPE	BFFC0810	Incompatible with sub-unit type
pipx40_ERROR_BAD_ROW	BFFC0811	Matrix row value out-of-

		range
pipx40_ERROR_BAD_COLUMN	BFFC0812	Matrix column value out-of-range
pipx40_ERROR_BAD_ATTEN	BFFC0813	Attenuation value out-of-range
pipx40_ERROR_BAD_VOLTAGE	BFFC0814	Voltage value out-of-range
pipx40_ERROR_BAD_CAL_INDEX	BFFC0815	Calibration index value out-of-range
pipx40_ERROR_BAD_SEGMENT	BFFC0816	Segment number out of range
pipx40_ERROR_BAD_FUNC_CODE	BFFC0817	Function code value out of range
pipx40_ERROR_BAD_SUBSWITCH	BFFC0818	Subswitch value out of range
pipx40_ERROR_BAD_ACTION	BFFC0819	Action code out of range
pipx40_ERROR_STATE_CORRUPT	BFFC081A	Cannot execute due to corrupt sub-unit state
pipx40_ERROR_BAD_ATTR_CODE	BFFC081B	Unrecognised attribute code
pipx40_ERROR_EEPROM_WRITE_TMO	BFFC081C	Timeout writing to EEPROM
pipx40_ERROR_ILLEGAL_OP	BFFC081D	Operation is illegal in the sub-unit's current state
pipx40_ERROR_BAD_POT	BFFC081E	Unrecognised pot number requested
pipx40_ERROR_MATRIXR_ILLEGAL	BFFC081F	Invalid write pattern for MATRIXR sub-unit
pipx40_ERROR_MISSING_CHANNEL	BFFC0820	Attempted operation on non-existent channel
pipx40_ERROR_CARD_INACCESSIBLE	BFFC0821	Card cannot be accessed (failed/removed/unpowered)
pipx40_ERROR_BAD_FP_FORMAT	BFFC0822	Unsupported internal floating-point format (internal error)
pipx40_ERROR_UNCALIBRATED	BFFC0823	Sub-unit is not calibrated
pipx40_ERROR_BAD_RESISTANCE	BFFC0824	Unobtainable resistance value
pipx40_ERROR_BAD_STORE	BFFC0825	Invalid calibration store number
pipx40_ERROR_BAD_MODE	BFFC0826	Invalid mode value
pipx40_ERROR_SETTINGS_CONFLICT	BFFC0827	Conflicting device settings
pipx40_ERROR_CARD_TYPE	BFFC0828	Function call incompatible with card type or capabilities
pipx40_ERROR_BAD_POLE	BFFC0829	Switch pole value out of

		range
pipx40_ERROR_MISSING_CAPABILITY	BFFC082A	Attempted to activate a non-existent capability
pipx40_ERROR_MISSING_HARDWARE	BFFC082B	Action requires hardware that is not present
pipx40_ERROR_HARDWARE_FAULT	BFFC082C	Faulty hardware
pipx40_ERROR_EXECUTION_FAIL	BFFC082D	Failed to execute (e.g. blocked by a hardware condition)
pipx40_ERROR_BAD_CURRENT	BFFC082E	Current value out of range
pipx40_ERROR_UNKNOWN	BFFC0FFF	Unspecified error

VXI IEEE-488 RS-232 PXI Programmable Switching Systems
www.pickeringtest.com

Contact Pickering

For further assistance, please contact:

Pickering Interfaces Ltd.

Stephenson Road

Clacton-on-Sea

Essex CO15 4NL

UK

Telephone: 44 (0)1255 687900

Fax: 44 (0)1255 425349

Regional contact details are available from our website: http://www.pickeringtest.com

Email (sales): sales@pickeringtest.com

Email (technical support): support@pickeringtest.com

Other useful links

PXI Systems Alliance (PXISA): http://www.pxisa.org

PCI Industrial Computer Manufacturers Group (PICMG): http://www.picmg.com

PCI Special Interest Group (PCI-SIG): http://www.pcisig.com

IVI Foundation (maintainer of the VISA standard): http://ivifoundation.org

System 41 Support List

The following System 41 models are supported by pipx40 driver version 3.10:

- 41-180-021
- 41-180-022
- 41-181-021
- 41-181-022
- 41-660-001
- 41-661-001
- 41-720
- 41-735-001
- 41-750-001
- 41-751-001
- 41-752-001

If your System 41 card does not appear in this list support for it may have been added subsequent to the above release; or it may be supported instead by its own card-specific driver. In either case the appropriate driver version can be downloaded from our website http://www.pickeringtest.com.

Cards with Special Features

Cards with Special Features

Certain cards support special features that are accessed using Input, General Purpose Output or other specific functions. The nature of these features and their methods of operation by the software driver are model-specific:

- 40-170-101, 40-170-102 Current Sensing Switch Cards
- 40-260-001 Precision Resistor
- 40-261 Precision Resistor
- 40-262 RTD Simulator
- 40-265 Strain Gauge Simulator
- 40-297 Precision Resistor
- 40-412-001 Digital Input-Output
- 40-412-101 Digital Input-Output
- 40-413-001 Digital Input-Output
- 40-413-002 Digital Input-Output
- 40-413-003 Digital Input-Output
- 41-750-001 Battery Simulator
- 41-751-001 Battery Simulator
- 41-752-001 Battery Simulator
- 50-297 Precision Resistor

40-170-101/102 Current Sensing Switch Card

The 40-170-101 and 40-170-102 cards contain current sensing circuitry to monitor the current flowing through the main relay contacts. A voltage proportional to the current flowing through the contacts is delivered to the monitor output on the card.

The card contains the following sub-units:

Output Sub-Units	**Function**
1	2 bit switch, 1 for each relay
2	2-way MUX, controls monitor of relay 1 or relay 2 or cascade if neither relay is selected
3 *	16-bit digital output, used to control current monitor circuit 1
4 *	16-bit digital output, used to control current monitor circuit 2

Input Sub-Units	**Function**
1 *	8-bit port to read result of control commands on circuit 1
2 *	8-bit port to read result of control commands on circuit 2
3 *	8-bit port to read RDAC(0) on circuit 1
4 *	8-bit port to read RDAC(1) on circuit 1
5 *	8-bit port to read RDAC(0) on circuit 2
6 *	8-bit port to read RDAC(1) on circuit 2

The sub-units marked with an asterisk (*) are used for calibration of the current monitoring circuits and are not required for normal operation, refer to the 40-170-101 User Manual for more detail.

40-260-001 Precision Resistor

The 40-260-001 Precision Resistor card contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Units	Applicable functions pipx40_resGetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(28)	Precision resistor 1
2: RES(28)	Precision resistor 2
3: RES(28)	Precision resistor 3

Output Sub-Unit	Applicable functions pipx40_setChannelState pipx40_clearSub pipx40_getChannelPattern
4: MUX(4)	Common reference multiplexer

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	Applicable functions pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP pipx40_writeCalibrationDate	Applicable functions pipx40_setChannelPattern pipx40_getChannelPattern
1: RES(28)	Precision resistor 1	PR1 switched resistance elements
2: RES(28)	Precision resistor 2	PR2 switched resistance elements
3: RES(28)	Precision resistor 3	PR3 switched resistance elements

Output Sub-Unit	Applicable functions pipx40_setChannelState pipx40_clearSub pipx40_getChannelPattern
5: MUX(9)	DMM multiplexer

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern
6: DIGITAL(32)	PR1 digital pot element
7: DIGITAL(32)	PR2 digital pot element
8: DIGITAL(32)	PR3 digital pot element

Refer to the 40-260-001 User Manual for more detail.

40-261 Precision Resistor

The 40-261-001 and 40-261-002 Precision Resistor cards contain an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Units	**Applicable functions** pipx40_resGetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(38)	Precision resistor 1
2: RES(38)	Precision resistor 2

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP pipx40_writeCalibrationDate	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern
1: RES(38)	Precision resistor 1	PR1 switched resistance elements
2: RES(38)	Precision resistor 2	PR2 switched resistance elements

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_clearSub
3: MUX(6)	DMM multiplexer

Refer to the 40-261 User Manual for more detail.

40-262 RTD Simulator

Model 40-262 RTD Simulator cards contain an array of sub-units for control and calibration.

Models 40-262-001, 40-262-002 (18 channels): functions for normal operation

Output Sub-Units	**Applicable functions** pipx40_resGetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(13)	Simulator channel 1
2: RES(13)	Simulator channel 2
3: RES(13)	Simulator channel 3
4: RES(13)	Simulator channel 4
5: RES(13)	Simulator channel 5
6: RES(13)	Simulator channel 6
7: RES(13)	Simulator channel 7
8: RES(13)	Simulator channel 8
9: RES(13)	Simulator channel 9
10: RES(13)	Simulator channel 10
11: RES(13)	Simulator channel 11
12: RES(13)	Simulator channel 12
13: RES(13)	Simulator channel 13
14: RES(13)	Simulator channel 14
15: RES(13)	Simulator channel 15
16: RES(13)	Simulator channel 16
17: RES(13)	Simulator channel 17
18: RES(13)	Simulator channel 18

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_clearSub pipx40_getChannelPattern
19: MUX(4)	Common reference multiplexer

Models 40-262-001, 40-262-002 (18 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output	**Applicable functions**	**Applicable functions**

Sub-Units	pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP pipx40_writeCalibrationDate	pipx40_setChannelPattern pipx40_getChannelPattern
1: RES(13)	Simulator channel 1	Sim chan 1 switched resistance elements
2: RES(13)	Simulator channel 2	Sim chan 2 switched resistance elements
3: RES(13)	Simulator channel 3	Sim chan 3 switched resistance elements
4: RES(13)	Simulator channel 4	Sim chan 4 switched resistance elements
5: RES(13)	Simulator channel 5	Sim chan 5 switched resistance elements
6: RES(13)	Simulator channel 6	Sim chan 6 switched resistance elements
7: RES(13)	Simulator channel 7	Sim chan 7 switched resistance elements
8: RES(13)	Simulator channel 8	Sim chan 8 switched resistance elements
9: RES(13)	Simulator channel 9	Sim chan 9 switched resistance elements
10: RES(13)	Simulator channel 10	Sim chan 10 switched resistance elements
11: RES(13)	Simulator channel 11	Sim chan 11 switched resistance elements
12: RES(13)	Simulator channel 12	Sim chan 12 switched resistance elements
13: RES(13)	Simulator channel 13	Sim chan 13 switched resistance elements
14: RES(13)	Simulator channel 14	Sim chan 14 switched resistance elements
15: RES(13)	Simulator channel 15	Sim chan 15 switched resistance elements
16: RES(13)	Simulator channel 16	Sim chan 16 switched resistance elements
17: RES(13)	Simulator channel 17	Sim chan 17 switched resistance elements
18: RES(13)	Simulator channel 18	Sim chan 18 switched resistance elements

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_clearSub pipx40_getChannelPattern
20: MUX(54)	DMM multiplexer

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern
21: DIGITAL(32)	Sim chan 1 digital pot element
22: DIGITAL(32)	Sim chan 2 digital pot element

23: DIGITAL(32)	Sim chan 3 digital pot element
24: DIGITAL(32)	Sim chan 4 digital pot element
25: DIGITAL(32)	Sim chan 5 digital pot element
26: DIGITAL(32)	Sim chan 6 digital pot element
27: DIGITAL(32)	Sim chan 7 digital pot element
28: DIGITAL(32)	Sim chan 8 digital pot element
29: DIGITAL(32)	Sim chan 9 digital pot element
30: DIGITAL(32)	Sim chan 10 digital pot element
31: DIGITAL(32)	Sim chan 11 digital pot element
33: DIGITAL(32)	Sim chan 12 digital pot element
33: DIGITAL(32)	Sim chan 13 digital pot element
34: DIGITAL(32)	Sim chan 14 digital pot element
35: DIGITAL(32)	Sim chan 15 digital pot element
36: DIGITAL(32)	Sim chan 16 digital pot element
37: DIGITAL(32)	Sim chan 17 digital pot element
38: DIGITAL(32)	Sim chan 18 digital pot element

Models 40-262-101, 40-262-102 (6 channels): functions for normal operation

Output Sub-Units	**Applicable functions** pipx40_res_GetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(13)	Simulator channel 1
2: RES(13)	Simulator channel 2
3: RES(13)	Simulator channel 3
4: RES(13)	Simulator channel 4
5: RES(13)	Simulator channel 5
6: RES(13)	Simulator channel 6

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_clearSub pipx40_getChannelPattern
7: MUX(4)	Common reference multiplexer

Models 40-262-101, 40-262-102 (6 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern

	pipx40_writeCalibrationDate	
1: RES(13)	Simulator channel 1	Sim chan 1 switched resistance elements
2: RES(13)	Simulator channel 2	Sim chan 2 switched resistance elements
3: RES(13)	Simulator channel 3	Sim chan 3 switched resistance elements
4: RES(13)	Simulator channel 4	Sim chan 4 switched resistance elements
5: RES(13)	Simulator channel 5	Sim chan 5 switched resistance elements
6: RES(13)	Simulator channel 6	Sim chan 6 switched resistance elements

Output Sub-Unit	**Applicable functions** pipx40_setChannelState, pipx40_clearSub, pipx40_getChannelPattern
8: MUX(18)	DMM multiplexer

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern
9: DIGITAL(32)	Sim chan 1 digital pot element
10: DIGITAL(32)	Sim chan 2 digital pot element
11: DIGITAL(32)	Sim chan 3 digital pot element
12: DIGITAL(32)	Sim chan 4 digital pot element
13: DIGITAL(32)	Sim chan 5 digital pot element
14: DIGITAL(32)	Sim chan 6 digital pot element

Refer to the 40-262 User Manual for more detail.

40-265 Strain Gauge Simulator

Strain Gauge Simulator model 40-265-006 contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Units	**Applicable functions** pipx40_resGetResistance pipx40_resSetResistance pipx40_resGetInfo pipx40_readCalibrationDate
1: RES(64)	Simulator channel 1
2: RES(64)	Simulator channel 2
3: RES(64)	Simulator channel 3
4: RES(64)	Simulator channel 4
5: RES(64)	Simulator channel 5
6: RES(64)	Simulator channel 6

Output Sub-Units	**Applicable functions** pipx40_setChannelState pipx40_setChannelPattern pipx40_getChannelState pipx40_getChannelPattern pipx40_clearSub
7: SWITCH(4)	Simulator channel 1 auxiliary switches
8: SWITCH(4)	Simulator channel 2 auxiliary switches
9: SWITCH(4)	Simulator channel 3 auxiliary switches
10: SWITCH(4)	Simulator channel 4 auxiliary switches
11: SWITCH(4)	Simulator channel 5 auxiliary switches
12: SWITCH(4)	Simulator channel 6 auxiliary switches

A simulator channel's null-point resistance can be obtained using function:

- pipx40_resGetInfo (in its refRes argument)

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** pipx40_setCalibrationPoint pipx40_readCalibrationFP	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern

	pipx40_writeCalibrationFP pipx40_writeCalibrationDate	
1: RES(64)	Simulator channel 1	Simulator channel 1 resistance elements
2: RES(64)	Simulator channel 2	Simulator channel 2 resistance elements
3: RES(64)	Simulator channel 3	Simulator channel 3 resistance elements
4: RES(64)	Simulator channel 4	Simulator channel 4 resistance elements
5: RES(64)	Simulator channel 5	Simulator channel 5 resistance elements
6: RES(64)	Simulator channel 6	Simulator channel 6 resistance elements

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_getChannelPattern pipx40_clearSub
13: MUX(18)	DMM multiplexer

Refer to the 40-265 User Manual for more detail.

40-297 Precision Resistor

40-297 Precision Resistor cards contain an array of sub-units for control and calibration.

Model 40-297-001 (18 channels): functions for normal operation

Output Sub-Unit	**Applicable functions** pipx40_resGetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(10)	Precision resistor 1
2: RES(10)	Precision resistor 2
3: RES(10)	Precision resistor 3
4: RES(10)	Precision resistor 4
5: RES(10)	Precision resistor 5
6: RES(10)	Precision resistor 6
7: RES(10)	Precision resistor 7
8: RES(10)	Precision resistor 8
9: RES(10)	Precision resistor 9
10: RES(10)	Precision resistor 10
11: RES(10)	Precision resistor 11
12: RES(10)	Precision resistor 12
13: RES(10)	Precision resistor 13
14: RES(10)	Precision resistor 14
15: RES(10)	Precision resistor 15
16: RES(10)	Precision resistor 16
17: RES(10)	Precision resistor 17
18: RES(10)	Precision resistor 18

Model 40-297-001 (18 channels): calibration functions

Output Sub-Unit	**Applicable functions** pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP pipx40_writeCalibrationDate	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern
1: RES(10)	Precision resistor 1	PR1 switched resistance elements
2: RES(10)	Precision resistor 2	PR2 switched resistance elements
3: RES(10)	Precision resistor 3	PR3 switched resistance elements
4: RES(10)	Precision resistor 4	PR4 switched resistance elements
5:	Precision resistor 5	PR5 switched resistance

RES(10)		elements
6: RES(10)	Precision resistor 6	PR6 switched resistance elements
7: RES(10)	Precision resistor 7	PR7 switched resistance elements
8: RES(10)	Precision resistor 8	PR8 switched resistance elements
9: RES(10)	Precision resistor 9	PR9 switched resistance elements
10: RES(10)	Precision resistor 10	PR10 switched resistance elements
11: RES(10)	Precision resistor 11	PR11 switched resistance elements
12: RES(10)	Precision resistor 12	PR12 switched resistance elements
13: RES(10)	Precision resistor 13	PR13 switched resistance elements
14: RES(10)	Precision resistor 14	PR14 switched resistance elements
15: RES(10)	Precision resistor 15	PR15 switched resistance elements
16: RES(10)	Precision resistor 16	PR16 switched resistance elements
17: RES(10)	Precision resistor 17	PR17 switched resistance elements
18: RES(10)	Precision resistor 18	PR18 switched resistance elements

Model 40-297-002 (9 channels): functions for normal operation

Output Sub-Unit	**Applicable functions** pipx40_resGetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(19)	Precision resistor 1
2: RES(19)	Precision resistor 2
3: RES(19)	Precision resistor 3
4: RES(19)	Precision resistor 4
5: RES(19)	Precision resistor 5
6: RES(19)	Precision resistor 6
7: RES(19)	Precision resistor 7
8: RES(19)	Precision resistor 8
9: RES(19)	Precision resistor 9

Model 40-297-002 (9 channels): calibration functions

Output Sub-Unit	**Applicable functions** pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP pipx40_writeCalibrationDate	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern

1: RES(19)	Precision resistor 1	PR1 switched resistance elements
2: RES(19)	Precision resistor 2	PR2 switched resistance elements
3: RES(19)	Precision resistor 3	PR3 switched resistance elements
4: RES(19)	Precision resistor 4	PR4 switched resistance elements
5: RES(19)	Precision resistor 5	PR5 switched resistance elements
6: RES(19)	Precision resistor 6	PR6 switched resistance elements
7: RES(19)	Precision resistor 7	PR7 switched resistance elements
8: RES(19)	Precision resistor 8	PR8 switched resistance elements
9: RES(19)	Precision resistor 9	PR9 switched resistance elements

Model 40-297-003 (6 channels): functions for normal operation

Output Sub-Unit	**Applicable functions** pipx40_resGetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(28)	Precision resistor 1
2: RES(28)	Precision resistor 2
3: RES(28)	Precision resistor 3
4: RES(28)	Precision resistor 4
5: RES(28)	Precision resistor 5
6: RES(28)	Precision resistor 6

Model 40-297-003 (6 channels): calibration functions

Output Sub-Unit	**Applicable functions** pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP pipx40_writeCalibrationDate	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern
1: RES(28)	Precision resistor 1	PR1 switched resistance elements
2: RES(28)	Precision resistor 2	PR2 switched resistance elements
3: RES(28)	Precision resistor 3	PR3 switched resistance elements
4: RES(28)	Precision resistor 4	PR4 switched resistance elements
5: RES(28)	Precision resistor 5	PR5 switched resistance elements
6: RES(28)	Precision resistor 6	PR6 switched resistance elements

Refer to the 40-297 User Manual for more detail.

40-412-001 Digital Input-Output

The 40-412-001 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Units	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub pipx40_setMaskState pipx40_getMaskState pipx40_setMaskPattern pipx40_getMaskPattern pipx40_clearMask
1: DIGITAL(32)	Controls output SINK driver states, each bit: 0 = INACTIVE 1 = ACTIVE
2: DIGITAL(32)	Controls output SOURCE driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
3: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
4: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_getChannelPattern pipx40_clearSub
5: MUX(32)	Input channel selector

Input Sub-Units	**Applicable function** pipx40_readInputPattern
1: INPUT(2)	Gets level of selected input channel (2 bits): 00 = below threshold 2, below threshold 1 01 = below threshold 2, above threshold 1 10 = above threshold 2, below threshold 1 11 = above threshold 2, above threshold 1
2: INPUT(64)	Gets levels of all 32 input channels (2 bits each, as above).

	NOTE: each input channel from 1 to 32 is sampled sequentially. The precise rate of sampling is undefined.

Refer to the 40-412 User Manual for more detail.

40-412-101 Digital Input-Output

The 40-412-101 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Units	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub pipx40_setMaskState pipx40_getMaskState pipx40_setMaskPattern pipx40_getMaskPattern pipx40_clearMask
1: DIGITAL(32)	Controls output SINK driver states, each bit: 0 = INACTIVE 1 = ACTIVE
2: DIGITAL(32)	Controls output SOURCE driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
3: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
4: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Input Sub-Units	**Applicable function** pipx40_readInputPattern
1: INPUT(64)	Gets levels of all 32 input channels, relative to the set thresholds. All input channels are sampled synchronously.

Refer to the 40-412 User Manual for more detail.

40-413-001 Digital Input-Output

The 40-413-001 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub pipx40_setMaskState pipx40_getMaskState pipx40_setMaskPattern pipx40_getMaskPattern pipx40_clearMask
1: DIGITAL(32)	Controls output (SOURCE) driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
2: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
3: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_getChannelPattern pipx40_clearSub
4: MUX(32)	Input channel selector

Input Sub-Units	**Applicable function** pipx40_readInputPattern
1: INPUT(2)	Gets level of selected input channel (2 bits): 00 = below threshold 2, below threshold 1 01 = below threshold 2, above threshold 1 10 = above threshold 2, below threshold 1 11 = above threshold 2, above threshold 1
2: INPUT(64)	Gets levels of all 32 input channels (2 bits each, as above). **NOTE: each input channel from 1 to 32 is sampled sequentially. The precise rate of sampling is**

	undefined.

Refer to the 40-413 User Manual for more detail.

40-413-002 Digital Input-Output

The 40-413-002 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub pipx40_setMaskState pipx40_getMaskState pipx40_setMaskPattern pipx40_getMaskPattern pipx40_clearMask
1: DIGITAL(32)	Controls output (SINK) driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
2: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
3: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_getChannelPattern pipx40_clearSub
4: MUX(32)	Input channel selector

Input Sub-Units	**Applicable function** pipx40_readInputPattern
1: INPUT(2)	Gets level of selected input channel (2 bits): 00 = below threshold 2, below threshold 1 01 = below threshold 2, above threshold 1 10 = above threshold 2, below threshold 1 11 = above threshold 2, above threshold 1
2: INPUT(64)	Gets levels of all 32 input channels (2 bits each, as above). **NOTE: each input channel from 1 to 32 is sampled sequentially. The precise rate of sampling is undefined.**

Refer to the 40-413 User Manual for more detail.

40-413-003 Digital Input-Output

The 40-413-003 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Units	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub pipx40_setMaskState pipx40_getMaskState pipx40_setMaskPattern pipx40_getMaskPattern pipx40_clearMask
1: DIGITAL(32)	Controls output SINK driver states, each bit: 0 = INACTIVE 1 = ACTIVE
2: DIGITAL(32)	Controls output SOURCE driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
3: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
4: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_getChannelPattern pipx40_clearSub
5: MUX(32)	Input channel selector

Input Sub-Units	**Applicable function** pipx40_readInputPattern
1: INPUT(2)	Gets level of selected input channel (2 bits): 00 = below threshold 2, below threshold 1 01 = below threshold 2, above threshold 1 10 = above threshold 2, below threshold 1 11 = above threshold 2, above threshold 1
2: INPUT(64)	Gets levels of all 32 input channels (2 bits each, as above).

	NOTE: each input channel from 1 to 32 is sampled sequentially. The precise rate of sampling is undefined.

Refer to the 40-413 User Manual for more detail.

41-750-001 Battery Simulator

The 41-750-001 Battery Simulator card contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Unit	Applicable functions pipx40_setChannelState pipx40_getChannelState pipx40_getChannelPattern pipx40_clearSub
1: MUX(4)	PIMS multiplexer

Output Sub-Units	Applicable functions pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
2: DIGITAL(96)	Current-sink setting
3: DIGITAL(16)	Voltage output DAC setting

Output Sub-Unit	Applicable functions pipx40_setChannelState pipx40_getChannelState pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
8: DIGITAL(1)	Output on/off control

Input Sub-Unit	Applicable functions pipx40_readInputState pipx40_readInputPattern
1: INPUT(1)	Read the Reg Limit Shutdown PXI Monitor signal

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	Applicable functions pipx40_setChannelPattern pipx40_getChannelPattern

4: DIGITAL(8)	Write RDAC1 register (pot #1 volatile setting)
5: DIGITAL(8)	Write RDAC3 register (pot #3 volatile setting)
6: DIGITAL(8)	Write EEMEM1 register (pot #1 non-volatile setting)
7: DIGITAL(8)	Write EEMEM3 register (pot #3 non-volatile setting)

Input Sub-Units	**Applicable function** pipx40_readInputPattern
2: INPUT(8)	Read RDAC1 register (pot #1 volatile setting)
3: INPUT(8)	Read RDAC3 register (pot #3 volatile setting)

Refer to the 41-750-001 User Manual for more detail.

41-751-001 Battery Simulator

The 41-751-001 Battery Simulator card contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_getChannelPattern pipx40_clearSub
1: MUX(4)	PIMS multiplexer

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
2: DIGITAL(48)	Current-sink setting
3: DIGITAL(16)	Voltage output DAC setting

Output Sub-Unit	**Applicable functions** pipx40_setChannelState pipx40_getChannelState pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
8: DIGITAL(1)	Output on/off control

Input Sub-Unit	**Applicable functions** pipx40_readInputState pipx40_readInputPattern
1: INPUT(2)	Read status signals RLSPM, CDPM

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-	**Applicable functions**

Units	pipx40_setChannelPattern pipx40_getChannelPattern
4: DIGITAL(8)	Write RDAC2 register (pot #2 volatile setting)
5: DIGITAL(8)	Write RDAC3 register (pot #3 volatile setting)
6: DIGITAL(8)	Write EEMEM2 register (pot #2 non-volatile setting)
7: DIGITAL(8)	Write EEMEM3 register (pot #3 non-volatile setting)
9: DIGITAL(8)	Write RDAC1 register (pot #1 volatile setting)
10: DIGITAL(8)	Write EEMEM1 register (pot #1 non-volatile setting)

Input Sub-Units	**Applicable function** pipx40_readInputPattern
2: INPUT(8)	Read RDAC2 register (pot #2 volatile setting)
3: INPUT(8)	Read RDAC3 register (pot #3 volatile setting)
4: INPUT(8)	Read RDAC1 register (pot #1 volatile setting)

Refer to the 41-751-001 User Manual for more detail.

41-752-001 Battery Simulator

The 41-752-001 Battery Simulator card contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Units	Applicable functions pipx40_battSetVoltage pipx40_battGetVoltage pipx40_battSetCurrent pipx40_battGetCurrent pipx40_battSetEnable pipx40_battGetEnable pipx40_battReadInterlockState
1: BATT(14) 2: BATT(14) 3: BATT(14) 4: BATT(14) 5: BATT(14) 6: BATT(14)	Battery simulator channels 1 thru 6

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	Applicable functions pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
1: BATT(14) 2: BATT(14) 3: BATT(14) 4: BATT(14) 5: BATT(14) 6: BATT(14)	Simulator channels 1 thru 6 voltage-setting DACs (direct binary access)

Output Sub-Units	Applicable functions pipx40_writeCalibration pipx40_readCalibration
1: BATT(14)	Simulator channels 1

2: BATT(14) 3: BATT(14) 4: BATT(14) 5: BATT(14) 6: BATT(14)	thru 6 calibration data (14 x 16-bit values per channel)

Output Sub-Units	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern pipx40_clearSub
7: DIGITAL(16) 8: DIGITAL(16) 9: DIGITAL(16) 10: DIGITAL(16) 11: DIGITAL(16) 12: DIGITAL(16)	Simulator channels 1 thru 6 current-setting DACs (direct binary access)

Output Sub-Unit	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern pipx40_setChannelState pipx40_getChannelState pipx40_clearSub
13: DIGITAL(6)	Simulator channels 1 thru 6 enable

Input Sub-Unit	**Applicable functions** pipx40_readInputPattern pipx40_readInputState
1: INPUT(1)	Global interlock state

Refer to the 41-752-001 User Manual for more detail.

50-297 Precision Resistor

50-297 Precision Resistor cards contain an array of sub-units for control and calibration.

Model 50-297-001 (18 channels): functions for normal operation

Output Sub-Unit	**Applicable functions** pipx40_resGetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(10)	Precision resistor 1
2: RES(10)	Precision resistor 2
3: RES(10)	Precision resistor 3
4: RES(10)	Precision resistor 4
5: RES(10)	Precision resistor 5
6: RES(10)	Precision resistor 6
7: RES(10)	Precision resistor 7
8: RES(10)	Precision resistor 8
9: RES(10)	Precision resistor 9
10: RES(10)	Precision resistor 10
11: RES(10)	Precision resistor 11
12: RES(10)	Precision resistor 12
13: RES(10)	Precision resistor 13
14: RES(10)	Precision resistor 14
15: RES(10)	Precision resistor 15
16: RES(10)	Precision resistor 16
17: RES(10)	Precision resistor 17
18: RES(10)	Precision resistor 18

Model 50-297-001 (18 channels): calibration functions

Output Sub-Unit	**Applicable functions** pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP pipx40_writeCalibrationDate	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern
1: RES(10)	Precision resistor 1	PR1 switched resistance elements
2: RES(10)	Precision resistor 2	PR2 switched resistance elements
3: RES(10)	Precision resistor 3	PR3 switched resistance elements
4: RES(10)	Precision resistor 4	PR4 switched resistance elements
5:	Precision resistor 5	PR5 switched resistance

RES(10)		elements
6: RES(10)	Precision resistor 6	PR6 switched resistance elements
7: RES(10)	Precision resistor 7	PR7 switched resistance elements
8: RES(10)	Precision resistor 8	PR8 switched resistance elements
9: RES(10)	Precision resistor 9	PR9 switched resistance elements
10: RES(10)	Precision resistor 10	PR10 switched resistance elements
11: RES(10)	Precision resistor 11	PR11 switched resistance elements
12: RES(10)	Precision resistor 12	PR12 switched resistance elements
13: RES(10)	Precision resistor 13	PR13 switched resistance elements
14: RES(10)	Precision resistor 14	PR14 switched resistance elements
15: RES(10)	Precision resistor 15	PR15 switched resistance elements
16: RES(10)	Precision resistor 16	PR16 switched resistance elements
17: RES(10)	Precision resistor 17	PR17 switched resistance elements
18: RES(10)	Precision resistor 18	PR18 switched resistance elements

Model 50-297-002 (9 channels): functions for normal operation

Output Sub-Unit	**Applicable functions** pipx40_resGetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(19)	Precision resistor 1
2: RES(19)	Precision resistor 2
3: RES(19)	Precision resistor 3
4: RES(19)	Precision resistor 4
5: RES(19)	Precision resistor 5
6: RES(19)	Precision resistor 6
7: RES(19)	Precision resistor 7
8: RES(19)	Precision resistor 8
9: RES(19)	Precision resistor 9

Model 50-297-002 (9 channels): calibration functions

Output Sub-Unit	**Applicable functions** pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP pipx40_writeCalibrationDate	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern

1: RES(19)	Precision resistor 1	PR1 switched resistance elements
2: RES(19)	Precision resistor 2	PR2 switched resistance elements
3: RES(19)	Precision resistor 3	PR3 switched resistance elements
4: RES(19)	Precision resistor 4	PR4 switched resistance elements
5: RES(19)	Precision resistor 5	PR5 switched resistance elements
6: RES(19)	Precision resistor 6	PR6 switched resistance elements
7: RES(19)	Precision resistor 7	PR7 switched resistance elements
8: RES(19)	Precision resistor 8	PR8 switched resistance elements
9: RES(19)	Precision resistor 9	PR9 switched resistance elements

Model 50-297-003 (6 channels): functions for normal operation

Output Sub-Unit	**Applicable functions** pipx40_resGetInfo pipx40_resGetResistance pipx40_resSetResistance pipx40_clearSub pipx40_readCalibrationDate
1: RES(28)	Precision resistor 1
2: RES(28)	Precision resistor 2
3: RES(28)	Precision resistor 3
4: RES(28)	Precision resistor 4
5: RES(28)	Precision resistor 5
6: RES(28)	Precision resistor 6

Model 50-297-003 (6 channels): calibration functions

Output Sub-Unit	**Applicable functions** pipx40_setCalibrationPoint pipx40_readCalibrationFP pipx40_writeCalibrationFP pipx40_writeCalibrationDate	**Applicable functions** pipx40_setChannelPattern pipx40_getChannelPattern
1: RES(28)	Precision resistor 1	PR1 switched resistance elements
2: RES(28)	Precision resistor 2	PR2 switched resistance elements
3: RES(28)	Precision resistor 3	PR3 switched resistance elements
4: RES(28)	Precision resistor 4	PR4 switched resistance elements
5: RES(28)	Precision resistor 5	PR5 switched resistance elements
6: RES(28)	Precision resistor 6	PR6 switched resistance elements

Refer to the 50-297 User Manual for more detail.

VISA Standard Functions

Initialise

pipx40_init

VB	Function	pipx40_init (ByVal rsrcName As String, ByVal id_query As Boolean, ByVal reset_instr As Boolean, ByRef vi As Long) As Long
C++	ViStatus	pipx40_init (ViRsrc rsrcName, ViBoolean id_query, ViBoolean reset_instr, ViPSession vi);

Parameter	I/O	Description
rsrcName	in	Instrument description (resource name)
id_query	in	if VI_TRUE then perform in-system verification; if VI_FALSE then do not perform in-system verification. **See note below.**
reset_instr	in	if VI_TRUE then perform reset operation; if VI_FALSE then do not perform reset operation. **See note below.**
vi	out	Instrument handle

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

This function establishes communications with the instrument.

Remarks

The values of the id_query and reset_instr parameters are ignored: instrument identity is **always** checked, and the instrument is **always** reset when it is opened. No error is given if these options are not specified in the function call.

If the initialisation function encounters an error, an error code return value will be sent, any valid sessions obtained by pipx40_init will be closed and the output parameter vi is set to zero (VI_NULL).

Utility

pipx40_error_message

VB	Function	pipx40_error_message (ByVal vi As Long, ByVal status_code As Long, ByVal message As String) As Long
C++	ViStatus	pipx40_error_message (ViSession vi, ViStatus status_code, ViString message);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
status_code	in	Instrument driver error code
message	out	Error message

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

This function translates the error return value from a Pipx40 instrument driver function to a user-readable string.

pipx40_error_query

VB	Function	pipx40_error_query (ByVal vi As Long, ByRef error_code As Long, ByVal error_message As String) As Long
C++	ViStatus	pipx40_error_query (ViSession vi, ViPInt32 error_code, ViString error_message);

Parameter	I/O	Description
vi	in	Instrument handle
error_code	out	Instrument error code
error_message	out	Error message

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Return an error code and corresponding message from the instrument's error queue.

Remarks

This feature is not supported by the instrument, and the function returns the status code VI_WARN_NSUP_ERROR_QUERY.

pipx40_reset

VB	Function	pipx40_reset (ByVal vi As Long) As Long
C++	ViStatus	pipx40_reset (ViSession vi);

Parameter	I/O	Description
vi	in	Instrument handle

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Resets the instrument to default state.

Remarks

All outputs of all the card's sub-units are:

- cleared, as by pipx40_clearSub
- unmasked, as by pipx40_clearMask

pipx40_revision_query

VB	Function	pipx40_revision_query (ByVal vi As Long, ByVal driver_rev As String, ByVal instr_rev As String) As Long
C++	ViStatus	pipx40_revision_query (ViSession vi, ViString driver_rev, ViString instr_rev);

Parameter	I/O	Description
vi	in	Instrument handle
driver_rev	out	Driver revision
instr_rev	out	Instrument revision

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

This function returns the instrument driver revision and instrument revision codes. The instr_rev value represents the hardware version of the unit - cards have no firmware on-board.

pipx40_self_test

VB	Function	pipx40_self_test (ByVal vi As Long, ByRef test_result As Integer, ByVal test_message As String) As Long
C++	ViStatus	pipx40_self_test (ViSession vi, ViPInt16 test_result, ViString test_message);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
test_result	out	Numeric result from self-test operation
test_message	out	Self-test status message

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

This function causes the instrument to perform a self-test and returns the result of that self-test.

Remarks

The test_result parameter is a numeric code for the test result. The test_message parameter returns a self-test status message. The codes are listed in the table below.

Driver constant	Numeric Value	**Description**
	0	Self-test passed with no errors
pipx40_FAULT_UNKNOWN	1	Unspecified fault
pipx40_FAULT_WRONG_DRIVER	2	Incompatible software driver version

pipx40_FAULT_EEPROM_ERROR	3	EEPROM data error
pipx40_FAULT_HARDWARE	4	Hardware defect
pipx40_FAULT_PARITY	5	Parity error
pipx40_FAULT_CARD_INACCESSIBLE	6	Card cannot be accessed (failed/removed/unpowered)
pipx40_FAULT_UNCALIBRATED	7	One or more sub-units is uncalibrated
pipx40_FAULT_CALIBRATION_DUE	8	One or more sub-units is due for calibration

Diagnostic information on fault conditions indicated in the test result can be obtained using pipx40_getDiagnostic.

Close

pipx40_close

VB	Function	pipx40_close (ByVal vi As Long) As Long
C++	ViStatus	pipx40_close (ViSession vi);

Parameter	**I/O**	**Description**
vi	in	Instrument handle

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Terminate the software connection to the instrument and deallocate system resources associated with the instrument.

Card Specific Functions

Information and Status

Information and Status

This section details the use of functions for obtaining card and sub-unit information. Most of these functions are applicable to all card or sub-unit types.

Functions are provided to:

- Obtain a card's identification string: pipx40_getCardId
- Obtain a card's status flags: pipx40_getCardStatus
- Obtain a card's diagnostic information string: pipx40_getDiagnostic
- Discover the numbers of input and output sub-units on a card: pipx40_getSubCounts
- Obtain sub-unit information (numeric format): pipx40_getSubInfo
- Obtain sub-unit information (string format): pipx40_getSubType
- Obtain an output sub-unit's closure limit value: pipx40_getClosureLimit
- Obtain an output sub-unit's settling time value: pipx40_getSettlingTime

pipx40_getCardId

VB	Function	pipx40_getCardId (ByVal vi As Long, ByVal id As String) As Long
C++	ViStatus	pipx40_getCardId (ViSession vi, ViString id);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
id	out	Instrument identification string

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the identification string of the specified card. The string contains these elements:

PICKERING INTERFACES,<type code>,<serial number>,<revision code>.

The <revision code> value represents the hardware version of the unit - cards have no firmware on-board.

Visual Basic Note

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(id, character_count).

pipx40_getCardStatus

VB	Function	pipx40_getCardStatus (ByVal vi As Long, ByRef status As Long) As Long
C++	ViStatus	pipx40_getCardStatus (ViSession vi, ViPUInt32 status);

Parameter	I/O	Description
vi	in	Instrument handle
status	out	A value representing the card's status flags

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the current status flags for the specified card.

Remarks

The status value is composed of the sum of a number of individual flag bits.

A zero value (pipx40_STAT_OK) indicates that the card is functional and its outputs are stable.

Driver constant	Bit value - hexadecimal	Description
pipx40_STAT_NO_CARD	80000000	The specified session is not associated with a Pickering card
pipx40_STAT_WRONG_DRIVER	40000000	Card requires a more recent version of the software driver

pipx40_STAT_EEPROM_ERR	20000000	Card EEPROM fault
pipx40_STAT_DISABLED	10000000	Card disabled
pipx40_STAT_BUSY	04000000	Card operations not yet completed
pipx40_STAT_HW_FAULT	02000000	Card hardware defect
pipx40_STAT_PARITY_ERROR	01000000	PCIbus parity error
pipx40_STAT_CARD_INACCESSIBLE	00080000	Card cannot be accessed (failed/removed/unpowered)
pipx40_STAT_UNCALIBRATED	00040000	One or more sub-units is uncalibrated
pipx40_STAT_CALIBRATION_DUE	00020000	One or more sub-units is due for calibration

Certain status bits are relevant only for specific classes of sub-unit, or for those having particular characteristics.

Diagnostic information on fault conditions indicated in the status value can be obtained using pipx40_getDiagnostic.

VISA may not allow a card that has experienced a PCIbus parity error to be opened, so in practice pipx40_STAT_PARITY_ERROR can never be reported.

pipx40_getClosureLimit

VB	Function	pipx40_getClosureLimit (ByVal vi As Long, ByVal subUnit As Long, ByRef limit As Long) As Long
C++	ViStatus	pipx40_getClosureLimit (ViSession vi, ViUInt32 subUnit, ViPUInt32 limit);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
limit	out	The maximum number of channel closures permitted

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the maximum number of channels that may be activated simultaneously in the specified sub-unit.

Remarks

A single-channel multiplexer (MUX type) allows only one channel to be closed at any time. In some other models such as high-density matrix types a limit is imposed to prevent overheating; although it is possible to disable the limit for these types (see pipx40_setDriverMode), doing so is not recommended.

pipx40_getDiagnostic

VB	Function	pipx40_getDiagnostic (ByVal vi As Long, ByVal message As String) As Long
C++	ViStatus	pipx40_getDiagnostic (ViSession vi, ViString message);

Parameter	I/O	Description
vi	in	Instrument handle
message	out	Instrument diagnostic string

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the diagnostic string of the specified card, giving expanded information on any fault conditions indicated by the pipx40_getCardStatus value or pipx40_self_test result.

Remarks

The result string may include embedded newline characters, coded as ASCII linefeed (0Ah).

The length of the result string will not exceed the value of the driver constant pipx40_MAX_DIAG_LENGTH.

Warning

Formatting and content of the diagnostic string may change as enhanced diagnostic features are made available. It should therefore not be interpreted programatically.

Visual Basic Notes

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(message, character_count).

If the diagnostic string is to be displayed in Visual Basic, any embedded linefeed characters (0Ah) should be expanded to vbCrLf.

pipx40_getSettlingTime

VB	Function	pipx40_getSettlingTime (ByVal vi As Long, ByVal subUnit As Long, ByRef ti As Long) As Long
C++	ViStatus	pipx40_getSettlingTime (ViSession vi, ViUInt32 subUnit, ViPUInt32 ti);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
ti	out	The settling time value, in microseconds

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a sub-unit's settling time (or debounce period - the time taken for its switches to stabilise).

Remarks

By default, pipx40 driver functions retain control during this period so that switches are guaranteed to have stabilised on completion. This mode of operation can be overridden if required - see pipx40_setDriverMode.

pipx40_getSubCounts

VB	Function	pipx40_getSubCounts (ByVal vi As Long, ByRef inSubs As Long, ByRef outSubs As Long) As Long
C++	ViStatus	pipx40_getSubCounts (ViSession vi, ViPUInt32 inSubs, ViPUInt32 outSubs);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
inSubs	in	pointer/reference to variable to receive the number of INPUT sub-units
outSubs	out	pointer/reference to variable to receive the number of OUTPUT sub-units

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the numbers of input and output sub-units implemented on the specified card.

pipx40_getSubInfo

VB	Function	pipx40_getSubInfo (ByVal vi As Long, ByVal subUnit As Long, ByVal out As Boolean, ByRef subType As Long, ByRef rows As Long, ByRef columns As Long) As Long
C++	ViStatus	pipx40_getSubInfo (ViSession vi, ViUInt32 subUnit, ViBoolean out, ViPUInt32 subType, ViPUInt32 rows, ViPUInt32 columns);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
out	in	sub-unit function: 0 for INPUT, 1 for OUTPUT
subType	out	pointer to variable to receive type code
rows	out	pointer to variable to receive row count
columns	out	pointer to variable to receive column count

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a type description of a sub-unit, as numeric values.

Remarks

Row and column values give the dimensions of the sub-unit. For all types other than matrices the column value contains the significant dimension: their row value is always 1.

Input sub-units are always indicated with a type code of 1.

Output sub-unit type codes are:

Driver constant	subType value	**Description**
pipx40_TYPE_SW	1	Uncommitted switches
pipx40_TYPE_MUX	2	Multiplexer, single-channel only
pipx40_TYPE_MUXM	3	Multiplexer, multi-channel
pipx40_TYPE_MAT	4	Matrix, LF
pipx40_TYPE_MATR	5	Matrix, RF
pipx40_TYPE_DIG	6	Digital outputs
pipx40_TYPE_RES	7	Programmable resistor
pipx40_TYPE_ATTEN	8	Programmable RF attenuator
pipx40_TYPE_PSUDC	9	Power supply, DC
pipx40_TYPE_BATT	10	Battery Simulator

Note that for some types additional information is obtainable using alternate functions:

- Programmable RF attenuator: pipx40_attenGetInfo

- Power supply: pipx40_psuGetInfo

pipx40_getSubStatus

VB	Function	pipx40_getSubStatus (ByVal vi As Long, ByVal subUnit As Long, ByRef status As Long) As Long
C++	ViStatus	pipx40_getSubStatus (ViSession vi, ViUInt32 subUnit, ViPUInt32 status);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
status	out	A value representing the sub-unit's status flags

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the current status flags for the specified sub-unit.

Remarks

The status value is composed of the sum of a number of individual flag bits.

A zero value (pipx40_STAT_OK) indicates that the sub-unit is functional and its outputs are stable.

Driver constant	**Bit value - hexadecimal**	**Description**
pipx40_STAT_NO_CARD	80000000	The specified session is not associated with a Pickering card
pipx40_STAT_WRONG_DRIVER	40000000	Card requires a more recent version of the

		software driver
pipx40_STAT_EEPROM_ERR	20000000	Card EEPROM fault
pipx40_STAT_DISABLED	10000000	Card disabled
pipx40_STAT_NO_SUB	08000000	Card has no sub-unit with specified number
pipx40_STAT_BUSY	04000000	Sub-unit operations not yet completed
pipx40_STAT_HW_FAULT	02000000	Card hardware defect
pipx40_STAT_PARITY_ERROR	01000000	PCIbus parity error
pipx40_STAT_PSU_INHIBITED	00800000	PSU sub-unit - supply is disabled (by software)
pipx40_STAT_PSU_SHUTDOWN	00400000	PSU sub-unit - supply is shutdown (due to overload)
pipx40_STAT_PSU_CURRENT_LIMIT	00200000	PSU sub-unit - supply is operating in current-limited mode
pipx40_STAT_CORRUPTED	00100000	Sub-unit logical state is corrupted
pipx40_STAT_CARD_INACCESSIBLE	00080000	Card cannot be accessed (failed/removed/unpowered)
pipx40_STAT_UNCALIBRATED	00040000	Sub-unit is uncalibrated
pipx40_STAT_CALIBRATION_DUE	00020000	Sub-unit is due for calibration

Certain status bits are relevant only for specific classes of sub-unit, or for those having particular characteristics.

Note that certain card-level conditions that affect the sub-unit's functionality are also reported.

Diagnostic information on fault conditions indicated in the status value can be obtained using pipx40_getDiagnostic.

VISA may not allow a card that has experienced a PCIbus parity error to be opened, so in practice pipx40_STAT_PARITY_ERROR can never be reported.

pipx40_getSubType

VB	Function	pipx40_getSubType (ByVal vi As Long, ByVal subUnit As Long, ByVal out As Boolean, ByVal subType As String) As Long
C++	ViStatus	pipx40_getSubType (ViSession vi, ViUInt32 subUnit, ViBoolean out, ViString type);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
out	in	sub-unit function: 0 for INPUT, 1 for OUTPUT
subType	out	character string to receive the result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a type description of a sub-unit, as a text string.

subType string	**Description**
INPUT(<size>)	Digital inputs
SWITCH(<size>)	Uncommitted switches

MUX(<size>)	Multiplexer, single-channel only
MUXM(<size>)	Multiplexer, multi-channel
MATRIX(<columns>X<rows>)	Matrix, LF
MATRIXR(<columns>X<rows>)	Matrix, RF
DIGITAL(<size>)	Digital outputs
RES(<size>)	Programmable resistor
ATTEN(<number of pads>)	Programmable RF attenuator
PSUDC(0)	Power supply, DC
BATT(<voltage DAC resolution, bits>)	Battery Simulator

Note that for some types additional information is obtainable using alternate functions:

- Programmable RF attenuator: pipx40_attenGetType
- Power supply: pipx40_psuGetType

Visual Basic Note

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(subType, character_count).

Switching and General Purpose Output

Switching and General Purpose Output

This section details the use of functions that are applicable to most output sub-unit types.

Note that although these functions may be used with them, some sub-unit types - for example matrix and programmable RF attenuator - are also served by specific functions offering more straightforward control.

Functions are provided to:

- Clear all output channels of a Pickering card: pipx40_clearCard
- Clear all output channels of a sub-unit: pipx40_clearSub
- Open or close a single output channel: pipx40_setChannelState
- Set a sub-unit's output pattern: pipx40_setChannelPattern
- Obtain the state of a single output channel: pipx40_getChannelState
- Obtain a sub-unit's output pattern: pipx40_setChannelPattern

pipx40_clearCard

VB	Function	pipx40_clearCard (ByVal vi As Long) As Long
C++	ViStatus	pipx40_clearCard (ViSession vi);

Parameter	I/O	Description
vi	in	Instrument handle

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Clears (de-energises or sets to logic '0') all output channels of all sub-units on the card.

pipx40_clearSub

VB	Function	pipx40_clearSub (ByVal vi As Long, ByVal subUnit As Long) As Long
C++	ViStatus	pipx40_clearSub (ViSession vi, ViUInt32 subUnit);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Clears (de-energises or sets to logic '0') all output channels of a sub-unit.

pipx40_getChannelPattern

VB	Function	pipx40_getChannelPattern (ByVal vi As Long, ByVal subUnit As Long, ByRef pattern As Long) As Long
C++	ViStatus	pipx40_getChannelPattern (ViSession vi, ViUInt32 subUnit, ViAUInt32 pattern);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
pattern	out	pointer/reference to the one-dimensional array (vector) to receive the result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the state of all output channels of a sub-unit.

Remarks

The result fills the number of least significant bits corresponding to the size of the sub-unit.

For a Matrix sub-unit, the result is folded into the vector on its row-axis. See Data formats.

Warning

The data array pointed to must contain sufficient bits to hold the bit-pattern for the specified sub-unit, or adjacent memory will be overwritten, causing data corruption and/or a program crash.

Visual Basic Note

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable for the result:

```
pipx40_getChannelPattern(vi, subUnit, pattern)
```

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

```
pipx40_getChannelPattern(vi, subUnit, pattern(0))
```

Example Code

See the description of pipx40_setChannelPattern for example code using a pattern-based function.

pipx40_getChannelState

VB	Function	pipx40_getChannelState (ByVal vi As Long, ByVal subUnit As Long, ByVal channel As Long, ByRef state As Boolean) As Long
C++	ViStatus	pipx40_getChannelState (ViSession vi, ViUInt32 subUnit, ViUInt32 channel, ViPBoolean state);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
channel	in	Channel whose state is to be reported
state	out	Pointer/reference to variable to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Reads the current state of the specified output channel (VI_OFF = open or logic '0', VI_ON = closed or logic '1').

pipx40_setChannelPattern

VB	Function	pipx40_setChannelPattern (ByVal vi As Long, ByVal subUnit As Long, ByRef pattern As Long) As Long
C++	ViStatus	pipx40_setChannelPattern (ViSession vi, ViUInt32 subUnit, ViAUInt32 pattern);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
pattern	out	pointer/reference to the one-dimensional array (vector) containing the bit-pattern to be written

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Sets all output channels of a sub-unit to the supplied bit-pattern.

Remarks

The number of least significant bits corresponding to the size of the sub-unit are written.

For a Matrix sub-unit, the data is folded into the vector on its row-axis. See Data formats.

In some high-density matrix cards the number of simultaneous crosspoint closures that can be made is restricted in order to prevent overheating. If the number of channel closures specified would exceed this limit an error is reported. The maximum number of channel closures permitted can be obtained using pipx40_getClosureLimit. Limit values are such that they should not impact on normal matrix operations. Although it is possible to override the closure limit using pipx40_setDriverMode this is **not** recommended as overheating could endanger both the card itself and the system in which it is installed.

In the case of a single-channel multiplexer (MUX type) sub-unit this function will only permit writing an array of nulls to clear it. MUX sub-units are more conveniently operated using pipx40_setChannelState and pipx40_clearSub.

Warning

The data array pointed to must contain sufficient bits to represent the bit-pattern for the specified sub-unit, or undefined data will be written to the more significant bits.

Visual Basic Note

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable containing the bit-pattern:

```
pipx40_setChannelPattern(vi, subUnit, pattern)
```

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

```
pipx40_setChannelPattern(vi, subUnit, pattern(0))
```

Example Code

Visual Basic Code Sample

Visual C++ Code Sample

pipx40_setChannelState

VB	Function	`pipx40_setChannelState (ByVal vi As Long, ByVal subUnit As Long, ByVal channel As Long, ByVal state As Boolean) As Long`
C++	ViStatus	`pipx40_setChannelState (ViSession vi, ViUInt32 subUnit, ViUInt32 channel, ViBoolean state);`

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
channel	in	Numeric variable indicating which channel will be affected
state	in	A Boolean indicating type of action, VI_ON to energise, VI_OFF to de-energise

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Energises or de-energises a single output channel. For a digital output, state = VI_ON sets logic '1'.

Remarks

For a single-channel multiplexer (MUX type), closing a channel results in automatic disconnection of the previously closed channel, if any.

In some high-density matrix cards the number of simultaneous crosspoint closures that can be made is restricted in order to prevent overheating. If this limit is exceeded no further crosspoints can be set and an error is reported. The maximum number of channel closures permitted can be obtained using pipx40_getClosureLimit. Limit values are such that they should not impact on normal matrix operations. Although it is possible to override the closure limit using pipx40_setDriverMode this is **not** recommended as overheating could endanger both the card itself and the system in which it is installed.

Specialised Switching

Specialised Switching

This section details the use of functions specific to particular types of switching sub-unit (uncommitted switches, multiplexer, matrix and digital output types).

Matrix operations

Open or close a single matrix crosspoint: pipx40_setCrosspointState

Obtain the state of a single matrix crosspoint: pipx40_getCrosspointState

Individual switch operations, complex matrix sub-units

Obtain/set the state of an individual switch: pipx40_operateSwitch

Attribute information, complex matrix sub-units

Obtain sub-unit attribute values: pipx40_getSubAttribute

pipx40_getCrosspointState

VB	Function	pipx40_getCrosspointState (ByVal vi As Long, ByVal subUnit As Long, ByVal row As Long, ByVal column As Long, ByRef state As Boolean) As Long
C++	ViStatus	pipx40_getCrosspointState (ViSession vi, ViUInt32 subUnit, ViUInt32 row, ViUInt32 column, ViPBoolean state);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
row	in	Row (Y) location of the crosspoint whose state is to be reported
column	in	Column (X) location of the crosspoint whose state is to be reported
state	out	Pointer/reference to variable to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Reads the current state of the specified matrix crosspoint (VI_OFF = open, VI_ON = closed).

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized channel-number method employed by pipx40_getChannelState. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

pipx40_operateSwitch

VB	Function	pipx40_operateSwitch (ByVal vi As Long, ByVal subUnit As Long, ByVal switchFunc As Long, ByVal segNum As Long, ByVal switchNum As Long, ByVal subSwitch As Long, ByVal switchAction As Long, ByRef state As Boolean) As Long
C++	ViStatus	pipx40_operateSwitch (ViSession vi, ViUInt32 subUnit, ViUInt32 switchFunc, ViUInt32 segNum, ViUInt32 switchNum, ViUInt32 subSwitch, ViUInt32 switchAction, ViPBoolean state);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
switchFunc	in	A code indicating the functional group of the switch
segNum	in	The segment in which the switch is located
switchNum	in	The logical number of the switch
subSwitch	in	The logical sub-switch
switchAction	in	A code indicating the action to perform
state	out	The state of the switch (after performing any action)

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

This function obtains, and optionally sets, the state of a switch. It allows explicit access to the individual switches making up a sub-unit, in types where their operation is normally handled automatically by the driver. The main purpose of this is in implementing fault diagnostic programs for such types; it can also be used where normal automated behaviour does not suit an application.

Applicable sub-unit types

This function is only usable with matrix sub-units having auto-isolation and/or auto-loopthru features. For further information see: segmented matrix, unsegmented matrix.

switchFunc Value

A value indicating the functional group of the switch to be accessed.

Value	Ident	Function
0	pipx40_SW_FUNC_CHANNEL	A channel (matrix crosspoint) switch
1	pipx40_SW_FUNC_X_ISO	A matrix X-isolation switch
2	pipx40_SW_FUNC_Y_ISO	A matrix Y-isolation switch
3	pipx40_SW_FUNC_X_LOOPTHRU	A matrix X-loopthru switch
4	pipx40_SW_FUNC_Y_LOOPTHRU	A matrix Y-loopthru switch

segNum Value

The segment location of the switch. The numbers and sizes of segments on each matrix axis can be obtained using pipx40_getSubAttribute.

In an unsegmented matrix, use segNum = 1.

In a segmented matrix, segment numbers for crosspoint and isolation switches are determined logically.

switchNum Value

The number of the switch in its functional group (unity-based).

For channel (crosspoint) switches, the switch number can be either:

- if segNum is zero, the global channel number of the switch (see channel number)
- if segNum is non-zero, the segment-local number of the switch, calculated in a similar way to the above

subSwitch Value

The number of the subswitch to operate (unity-based). This parameter caters for a situation in which a logical channel, isolation or loopthru switch is served by more than one physical relay (as for example when 2-pole operation is implemented using independently-driven single-pole relays).

The numbers of subswitches for each functional group can be obtained using pipx40_getSubAttribute.

switchAction Value

A code indicating the action to be performed.

Value	**Ident**	**Function**
0	pipx40_SW_ACT_NONE	No switch change - just set state result
1	pipx40_SW_ACT_OPEN	Open switch
2	pipx40_SW_ACT_CLOSE	Close switch

Loopthru switches

Loopthru switches are initialised by the driver to a closed state, which may mean that they are either energised or de-energised depending upon their type. In normal automated operation loopthru switches open when any crosspoint on their associated line is closed. Actions pipx40_SW_ACT_CLOSE and pipx40_SW_ACT_OPEN close or open loopthru switch contacts as their names imply.

Operational considerations

This function can be used to alter a pre-existing switch state in a sub-unit, set up by fuctions such as pipx40_setChannelState or pipx40_setChannelPattern. However once the state of any switch is changed by pipx40_operateSwitch the logical state of the sub-unit is considered to have been destroyed. This condition is flagged in the result of pipx40_getSubStatus (bit pipx40_STAT_CORRUPTED). Subsequent attempts to operate it using 'ordinary' switch functions such as pipx40_setChannelState, pipx40_getChannelState etc. will fail (result pipx40_ERROR_STATE_CORRUPT). Normal operation can be restored by clearing the sub-unit using pipx40_clearSub or pipx40_clearCard.

pipx40_setCrosspointState

VB	Function	pipx40_setCrosspointState (ByVal vi As Long, ByVal subUnit As Long, ByVal row As Long, ByVal column As Long, ByVal state As Boolean) As Long
C++	ViStatus	pipx40_setCrosspointState (ViSession vi, ViUInt32 subUnit, ViUInt32 row, ViUInt32 column, ViBoolean state);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
row	in	Numeric variable indicating the row (Y) location of the crosspoint to be affected
column	in	Numeric variable indicating the column (X) location of the crosspoint to be affected
state	in	A Boolean indicating type of action, VI_ON to energise, VI_OFF to de-energise

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Energises or de-energises a single matrix crosspoint.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized channel-number method employed by pipx40_setChannelState. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

Related Matrix Functions

pipx40_getCrosspointState

pipx40_setCrosspointMask

pipx40_getCrosspointMask

Remarks

In some high-density matrix cards the number of simultaneous crosspoint closures that can be made is restricted in order to prevent overheating. If this limit is exceeded no further crosspoints can be set and an error is reported. The maximum number of channel closures permitted can be obtained using pipx40_getClosureLimit. Limit values are such that they should never impact on normal matrix operations. Although it is possible to override the closure limit using pipx40_setDriverMode this is **not** recommended as overheating could endanger both the card itself and the system in which it is installed.

pipx40_getSubAttribute

VB	Function	pipx40_getSubAttribute (ByVal vi As Long, ByVal subUnit As Long, ByVal out As Boolean, ByVal attrCode As Long, ByRef attrValue As Long) As Long
C++	ViStatus	pipx40_getSubAttribute (ViSession vi, ViUInt32 subUnit, ViBoolean out, ViUInt32 attrCode, ViPUInt32 attrValue);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
out	in	sub-unit function: 0 for INPUT (unsupported), 1 for OUTPUT
attrCode	in	A numeric code indicating the attribute to be queried - see below
attrValue	out	The value of the selected attribute

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains attributes describing the internal organisation of sub-units having auto-isolation and/or auto-loopthru features, to facilitate operation by pipx40_operateSwitch.

attrCode values

Value	**Ident**	**Function**

1	pipx40_SUB_ATTR_CHANNEL_SUBSWITCHES	Gets number of subswitches per logical channel (matrix crosspoint)
2	pipx40_SUB_ATTR_X_ISO_SUBSWITCHES	Gets number of subswitches per logical X-isolator
3	pipx40_SUB_ATTR_Y_ISO_SUBSWITCHES	Gets number of subswitches per logical Y-isolator
4	pipx40_SUB_ATTR_X_LOOPTHRU_SUBSWITCHES	Gets number of subswitches per logical X-loopthru
5	pipx40_SUB_ATTR_Y_LOOPTHRU_SUBSWITCHES	Gets number of subswitches per logical Y-loopthru
0x100	pipx40_SUB_ATTR_NUM_X_SEGMENTS	Gets number of X-axis segments
0x101	pipx40_SUB_ATTR_X_SEGMENT01_SIZE	Gets size of X-axis segment 1
0x102	pipx40_SUB_ATTR_X_SEGMENT02_SIZE	Gets size of X-axis segment 2
0x103	pipx40_SUB_ATTR_X_SEGMENT03_SIZE	Gets size of X-axis segment 3
0x104	pipx40_SUB_ATTR_X_SEGMENT04_SIZE	Gets size of X-axis segment 4
0x105	pipx40_SUB_ATTR_X_SEGMENT05_SIZE	Gets size of X-axis segment 5

		segment 5
0x106	pipx40_SUB_ATTR_X_SEGMENT06_SIZE	Gets size of X-axis segment 6
0x107	pipx40_SUB_ATTR_X_SEGMENT07_SIZE	Gets size of X-axis segment 7
0x108	pipx40_SUB_ATTR_X_SEGMENT08_SIZE	Gets size of X-axis segment 8
0x109	pipx40_SUB_ATTR_X_SEGMENT09_SIZE	Gets size of X-axis segment 9
0x10A	pipx40_SUB_ATTR_X_SEGMENT10_SIZE	Gets size of X-axis segment 10
0x10B	pipx40_SUB_ATTR_X_SEGMENT11_SIZE	Gets size of X-axis segment 11
0x10C	pipx40_SUB_ATTR_X_SEGMENT12_SIZE	Gets size of X-axis segment 12
0x200	pipx40_SUB_ATTR_NUM_Y_SEGMENTS	Gets number of Y-axis segments
0x201	pipx40_SUB_ATTR_Y_SEGMENT01_SIZE	Gets size of y-axis segment 1
0x202	pipx40_SUB_ATTR_Y_SEGMENT02_SIZE	Gets size of y-axis segment 2

Switch Masking

Switch Masking

This section details the use of switch masking functions.

Masking permits disabling operation of chosen switch channels by the pipx40_setChannelState, pipx40_setCrosspointState and pipx40_setChannelPattern functions. These functions report error pipx40_ERROR_OUTPUT_MASKED if an attempt is made to activate a masked channel.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Masking functions, all switching sub-unit types

Clear a sub-unit's mask: pipx40_clearMask

Mask or unmask a single output channel: pipx40_setMaskState

Set a sub-unit's mask pattern: pipx40_setMaskPattern

Obtain the mask state of a single output channel: pipx40_getMaskState

Obtain a sub-unit's mask pattern: pipx40_getMaskPattern

Masking functions, matrix sub-units

Mask or unmask a single matrix crosspoint: pipx40_setCrosspointMask

Obtain the mask state of a single matrix crosspoint: pipx40_getCrosspointMask

Note

Masking only allows output channels to be disabled in the OFF state; applying a mask to a channel that is already turned ON forces it OFF.

pipx40_clearMask

VB	Function	pipx40_clearMask (ByVal vi As Long, ByVal subUnit As Long) As Long
C++	ViStatus	pipx40_clearMask (ViSession vi, ViUInt32 subUnit);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Clears a sub-unit's switch mask, enabling operation of all output channels by the pipx40_setChannelState and pipx40_setChannelPattern functions.

pipx40_getCrosspointMask

VB	Function	pipx40_getCrosspointMask (ByVal vi As Long, ByVal subUnit As Long, ByVal row As Long, ByVal column As Long, ByRef state As Boolean) As Long
C++	ViStatus	pipx40_getCrosspointMask (ViSession vi, ViUInt32 subUnit, ViUInt32 row, ViUInt32 column, ViPBoolean state);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
row	in	Row (Y) location of the crosspoint whose mask state is to be reported
column	in	Column (X) location of the crosspoint whose mask state is to be reported
state	out	Pointer/reference to variable to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Reads the current state of the specified matrix crosspoint's mask (VI_OFF = unmasked, VI_ON = masked).

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized channel-number method employed by pipx40_getMaskState. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

pipx40_getMaskPattern

VB	Function	pipx40_getMaskPattern (ByVal vi As Long, ByVal subUnit As Long, ByRef pattern As Long) As Long
C++	ViStatus	pipx40_getMaskPattern (ViSession vi, ViUInt32 subUnit, ViPUInt32 pattern);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
pattern	out	Pointer/reference to the one-dimensional array (vector) to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the switch mask of a sub-unit.

Remarks

The result fills the number of least significant bits corresponding to the size of the sub-unit.

For a Matrix sub-unit, the result is folded into the vector on its row-axis. See Data formats.

Warning

The data array pointed to must contain sufficient bits to hold the bit-pattern for the specified sub-unit, or adjacent memory will be overwritten, causing data corruption and/or a program crash.

Visual Basic Note

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable for the result:

```
pipx40_getMaskPattern(vi, subUnit, pattern)
```

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

```
pipx40_getMaskPattern(vi, subUnit, pattern(0))
```

Example Code

See the description of pipx40_setChannelPattern for example code using a pattern-based function.

pipx40_getMaskState

VB	Function	pipx40_getMaskState (ByVal vi As Long, ByVal subUnit As Long, ByVal channel As Long, ByRef state As Boolean) As Long
C++	ViStatus	pipx40_getMaskState (ViSession vi, ViUInt32 subUnit, ViUInt32 channel, ViPBoolean state);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
channel	in	Channel whose mask state is to be reported
state	out	Pointer/reference to variable to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Reads the current state of the specified output channel's mask (VI_OFF = unmasked, VI_ON = masked).

pipx40_setCrosspointMask

VB	Function	pipx40_setCrosspointMask (ByVal vi As Long, ByVal subUnit As Long, ByVal row As Long, ByVal column As Long, ByVal state As Boolean) As Long
C++	ViStatus	pipx40_setCrosspointMask (ViSession vi, ViUInt32 subUnit, ViUInt32 row, ViUInt32 column, ViBoolean state);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
row	in	Row (Y) location of the crosspoint to be affected
column	in	Column (X) location of the crosspoint to be affected
state	in	VI_ON to mask, VI_OFF to unmask

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Mask or unmask a single matrix crosspoint.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized channel-number method employed by pipx40_setMaskState. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

Remarks

Masking disables the corresponding crosspoint for the pipx40_setChannelState, pipx40_setCrosspointState and pipx40_setChannelPattern functions. An error is reported by those functions if an attempt is made to activate a masked channel.

This facility is particularly useful to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

pipx40_setMaskPattern

VB	Function	pipx40_setMaskPattern (ByVal vi As Long, ByVal subUnit As Long, ByRef pattern As Long) As Long
C++	ViStatus	pipx40_setMaskPattern (ViSession vi, ViUInt32 subUnit, ViPUInt32 pattern);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
pattern	in	Pointer/reference to the one-dimensional array (vector) containing the mask pattern to be set

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Sets a sub-unit's switch mask to the supplied bit-pattern.

Remarks

The number of least significant bits corresponding to the size of the sub-unit are written into the mask. A '1' bit in the mask disables the corresponding switch for the pipx40_setChannelState, pipx40_setCrosspointState and pipx40_setChannelPattern functions.

An error is reported by those functions if an attempt is made to activate a masked channel.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

For a Matrix sub-unit, the mask data is folded into the vector on its row-axis. See Data formats.

Certain single-channel multiplexer (MUX type) sub-units have a default channel (that is, a channel that is connected when the sub-unit is in a 'cleared' state). This channel cannot be masked, and error pipx40_ERROR_ILLEGAL_MASK is given if an attempt is made to mask it.

Warning

The data array pointed to must contain sufficient bits to represent the mask pattern for the specified sub-unit, or undefined data will be written to the more significant bits.

Visual Basic Note

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable containing the bit-pattern:

```
pipx40_setMaskPattern(vi, subUnit, pattern)
```

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

```
pipx40_setMaskPattern(vi, subUnit, pattern(0))
```

Example Code

See the description of pipx40_setChannelPattern for example code using a pattern-based function.

pipx40_setMaskState

VB	Function	pipx40_setMaskState (ByVal vi As Long, ByVal subUnit As Long, ByVal channel As Long, ByVal state As Boolean) As Long
C++	ViStatus	pipx40_setMaskState (ViSession vi, ViUInt32 subUnit, ViUInt32 channel, ViBoolean state);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
channel	in	Channel to be affected
state	in	VI_ON to mask, VI_OFF to unmask

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Mask or unmask a single output channel.

Remarks

Masking disables the corresponding channel for the pipx40_setChannelState, pipx40_setCrosspointState and pipx40_setChannelPattern functions. An error is reported by those functions if an attempt is made to activate a masked channel.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Certain single-channel multiplexer (MUX type) sub-units have a default channel (that is, a channel that is connected when the sub-unit is in a 'cleared' state). This channel cannot be masked, and error pipx40_ERROR_ILLEGAL_MASK is given if an attempt is made to mask it.

Input

Input

This section details the use of functions specific to input sub-units.

Specific functions are provided to:

- Obtain the state of a single input: pipx40_readInputState
- Obtain a sub-unit's input pattern: pipx40_readInputPattern

pipx40_readInputPattern

VB	Function	pipx40_readInputPattern (ByVal vi As Long, ByVal subUnit As Long, ByRef pattern As Long) As Long
C++	ViStatus	pipx40_readInputPattern (ViSession vi, ViUInt32 subUnit, ViAUInt32 pattern);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
pattern	out	Pointer/reference to the one-dimensional array (vector) to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the current state of all inputs of a sub-unit.

Warning

The data array pointed to must contain sufficient bits to hold the bit-pattern for the specified sub-unit, or adjacent memory will be overwritten, causing data corruption and/or a program crash.

Visual Basic Note

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable for the result:

```
pipx40_readInputPattern(vi, subUnit, pattern)
```

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

```
pipx40_readInputPattern(vi, subUnit, pattern(0))
```

Example Code

See the description of pipx40_setChannelPattern for example code using a pattern-based function.

pipx40_readInputState

VB	Function	`pipx40_readInputState (ByVal vi As Long, ByVal subUnit As Long, ByVal channel As Long, ByRef state As Boolean) As Long`
C++	ViStatus	`pipx40_readInputState (ViSession vi, ViUInt32 subUnit, ViUInt32 channel, ViPBoolean state);`

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
channel	in	Channel to be read
state	out	Pointer/reference to variable to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Reads the current state of the specified input channel (VI_OFF = logic '0', VI_ON = logic '1').

Calibration

Calibration

This section details the use of functions associated with storing calibration values in a card's non-volatile (EEPROM) memory. This facility is only available for certain sub-unit types, such as programmable resistors.

Specific functions are provided to:

- Retrieve an integer calibration value from non-volatile memory: pipx40_readCalibration
- Store an integer calibration value in non-volatile memory: pipx40_writeCalibration
- Retrieve floating-point calibration value(s) from non-volatile memory: pipx40_readCalibrationFP
- Store floating-point calibration value(s) in non-volatile memory: pipx40_writeCalibrationFP
- Retrieve a sub-unit's calibration date from non-volatile memory: pipx40_readCalibrationDate
- Store a sub-unit's calibration date in non-volatile memory: pipx40_writeCalibrationDate
- Set a calibration point: pipx40_setCalibrationPoint

pipx40_readCalibration

VB	Function	pipx40_readCalibration (ByVal vi As Long, ByVal subUnit As Long, ByVal idx As Long, ByRef data As Long) As Long
C++	ViStatus	pipx40_readCalibration (ViSession vi, ViUInt32 subUnit, ViUInt32 idx, ViPUInt32 data);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
idx	in	Index of the calibration value to be affected - see below
data	out	Pointer/reference to variable to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Reads an integer calibration value from on-card non-volatile (EEPROM) memory.

Remarks

This function is usable only with sub-units that support integer calibration data.

In simple programmable resistor models such as:

40-280

40-281

40-282

40-290

40-291

40-295

40-296

50-295

the pipx40 driver places no interpretation on the stored value - an application program can utilise it in any way it wishes.

In some other models, including:

41-735-001

41-752-001

stored values are utilised by specific pipx40 driver functions, and they should only be overwritten by an appropriate calibration utility.

For programmable resistors supporting this function the valid range of idx values corresponds to the number of bits, i.e. to the range of valid output channel numbers. A 16-bit resistor sub-unit typically provides 16 x 16-bit values.

The storage capacity of other types supporting this feature is determined by their functionality.

pipx40_readCalibrationDate

VB	Function	pipx40_readCalibrationDate (ByVal vi As Long, ByVal subUnit As Long, ByVal store As Long, ByRef year As Long, ByRef day As Long, ByRef interval As Long) As Long
C++	ViStatus	pipx40_readCalibrationDate (ViSession vi, ViUInt32 subUnit, ViUInt32 store, ViPUInt32 year, ViPUInt32 day, ViPUInt32 interval);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
store	in	Numeric variable indicating which store to access (see below)
year	out	Pointer/reference to variable to receive the year of calibration
day	out	Pointer/reference to variable to receive the day in the year of calibration
interval	out	Pointer/reference to variable to receive calibration interval (in days)

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Reads a sub-unit's calibration date and interval from on-card non-volatile (EEPROM) memory.

Remarks

This function is only applicable to sub-units that support floating-point calibration data; it can be used to discover when the sub-unit was last calibrated, and when recalibration will become due. Bit pipx40_STAT_CALIBRATION_DUE in the result of pipx40_getCardStatus or pipx40_getSubStatus indicates the need for recalibration.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "store" parameter	**Ident**	**Function**
0	pipx40_CAL_STORE_USER	Access user calibration store
1	pipx40_CAL_STORE_FACTORY	Access factory calibration store

pipx40_readCalibrationFP

VB	Function	pipx40_readCalibrationFP (ByVal vi As Long, ByVal subUnit As Long, ByVal store As Long, ByVal offset As Long, ByVal numValues As Long, ByRef data As Double) As Long
C++	ViStatus	pipx40_readCalibrationFP (ViSession vi, ViUInt32 subUnit, ViUInt32 store, ViUInt32 offset, ViUInt32 numValues, ViAReal64 data);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
store	in	Numeric variable indicating which store to access (see below)
offset	in	Offset in the calibration store of the first value to be read
numValues	in	The number of calibration values to read
data	out	Pointer/reference to array to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Reads one or more floating-point calibration values from on-card non-volatile (EEPROM) memory.

Remarks

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as pipx40_resSetResistance. The number of values stored and their purpose is specific to the target sub-unit.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "store" parameter	**Ident**	**Function**
0	pipx40_CAL_STORE_USER	Access user calibration store
1	pipx40_CAL_STORE_FACTORY	Access factory calibration store

pipx40_setCalibrationPoint

VB	Function	pipx40_setCalibrationPoint (ByVal vi As Long, ByVal subUnit As Long, ByVal idx As Long) As Long
C++	ViStatus	pipx40_setCalibrationPoint (ViSession vi, ViUInt32 subUnit, ViUInt32 idx);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
idx	in	Numeric variable indicating the calibration point (see below)

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Sets a sub-unit to a state corresponding to one of its defined calibration points.

Notes

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as pipx40_resSetResistance. The number of calibration points supported is specific to the target sub-unit.

The idx value used by this function corresponds directly to the offset in the sub-unit's calibration store at which the value is to be stored and retrieved, using pipx40_writeCalibrationFP and pipx40_readCalibrationFP.

WARNING

Selection of a calibration point causes the sub-unit to change state; the resulting state may be outside its normally desired range of operation. On completion of a calibration sequence, pipx40_resSetResistance can be used to normalise the setting.

pipx40_writeCalibration

VB	Function	pipx40_writeCalibration (ByVal vi As Long, ByVal subUnit As Long, ByVal idx As Long, ByVal data As Long) As Long
C++	ViStatus	pipx40_writeCalibration (ViSession vi, ViUInt32 subUnit, ViUInt32 idx, ViUInt32 data);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
idx	in	Index of the calibration value to be affected - see below
data	in	The calibration value to be written

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Writes an integer calibration value into on-card non-volatile (EEPROM) memory.

Remarks

This function is usable only with sub-units that support integer calibration data.

In simple programmable resistor models such as:

40-280

40-281

40-282

40-290

40-291

40-295

40-296

50-295

the pipx40 driver places no interpretation on the stored value - an application program can utilise it in any way it wishes.

In some other models, including:

41-735-001

41-752-001

stored values are utilised by specific pipx40 driver functions, and they should only be overwritten by an appropriate calibration utility.

The number of bits actually stored is specific to the target sub-unit - any redundant high-order bits of the supplied data value are ignored.

For programmable resistors supporting this function the valid range of idx values corresponds to the number of bits, i.e. to the range of valid output channel numbers. A 16-bit resistor sub-unit typically provides 16 x 16-bit values.

The storage capacity of other types supporting this feature is determined by their functionality.

pipx40_writeCalibrationDate

VB	Function	pipx40_writeCalibrationDate (ByVal vi As Long, ByVal subUnit As Long, ByVal store As Long, ByVal interval As Long) As Long
C++	ViStatus	pipx40_writeCalibrationDate (ViSession vi, ViUInt32 subUnit, ViUInt32 store, ViUInt32 interval);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
store	in	Numeric variable indicating which store to access (see below)
interval	in	The desired calibration interval (in days)

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Writes a sub-unit's calibration date and interval into on-card non-volatile (EEPROM) memory. Date information is obtained from the current system date.

Remarks

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as pipx40_resSetResistance. The number of values stored and their purpose is specific to the target sub-unit.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "store" parameter	**Ident**	**Function**
0	pipx40_CAL_STORE_USER	Access user calibration store
1	pipx40_CAL_STORE_FACTORY	Access factory calibration store

pipx40_writeCalibrationFP

VB	Function	pipx40_writeCalibrationFP (ByVal vi As Long, ByVal subUnit As Long, ByVal store As Long, ByVal offset As Long, ByVal numValues As Long, ByRef data As Double) As Long
C++	ViStatus	pipx40_writeCalibrationFP (ViSession vi, ViUInt32 subUnit, ViUInt32 store, ViUInt32 offset, ViUInt32 numValues, ViAReal64 data);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
store	in	Numeric variable indicating which store to access (see below)
offset	in	Offset in the calibration store of the first value to be written
numValues	in	The number of calibration values to write
data	out	Pointer/reference to array containing the values to write

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Writes one or more floating-point calibration values into on-card non-volatile (EEPROM) memory.

Remarks

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as pipx40_resSetResistance. The number of values stored and their purpose is specific to the target sub-unit.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "store" parameter	**Ident**	**Function**
0	pipx40_CAL_STORE_USER	Access user calibration store
1	pipx40_CAL_STORE_FACTORY	Access factory calibration store

WARNING

Writing new values will affect the sub-unit's calibration.

Programmable Resistor

Programmable Resistor

This section details the use of functions specific to programmable resistor sub-units.

Detailed information about a programmable resistor sub-unit, if available, can be obtained using function pipx40_resGetInfo.

Precision models

Precision programmable resistor models such as 40-260-001 are supported by functions:

- pipx40_resGetResistance
- pipx40_resSetResistance

which allow chosen resistance values to be set.

Simple models

In models not supported by the above functions general purpose output functions such as pipx40_setChannelPattern must be used to program resistance values by setting bit-patterns explicitly.

Models 40-280, 40-281 and 40-282 are configured as simple resistor/switch arrays and programming should be straightforward.

In models employing a series resistor chain - such as 40-290, 40-291 and 40-295 - each of a card's programmable resistors is implemented as a separate logical sub-unit and is constructed from a series chain of individual fixed resistor elements, each element having an associated shorting switch. In the cleared state all switches are open, giving the programmable resistor its maximum value. A nominal value of zero ohms is obtained by turning all switches ON; other values by turning on an appropriate pattern of switches.

In standard models the individual fixed resistors are arranged in a binary sequence, the least significant bit of the least significant element in the array passed to pipx40_setChannelPattern corresponding to the lowest value resistor element. For example, in a standard 16-bit resistor of 32768 ohms:

Data[0] bit 0 (value 0x0001) corresponds to the 0R5 resistor element

Data[0] bit 1 (value 0x0002) corresponds to the 1R0 resistor element

thru...

Data[0] bit 15 (value 0x8000) corresponds to the 16384R resistor element

Setting a nominal value of 68 ohms (= 64 + 4 ohms) therefore requires Data[0] set to 0xFF77 (the inverse of the binary pattern 0000 0000 1000 1000).

Special models may have some other arrangement, and may also include a fixed offset resistor that is permanently in circuit.

Non-volatile (EEPROM) storage of calibration values is supported through the functions pipx40_readCalibration and pipx40_writeCalibration.

See the application note on Simple Programmable Resistor Cards.

Summary of functions for normal operation of "Programmable Resistor" cards

Model(s)	Class	Functions
40-260-001	Precision	pipx40_resSetResistance
		pipx40_resGetResistance
		pipx40_readCalibrationDate
40-260-999	Precision	pipx40_setChannelPattern
		pipx40_getChannelPattern
40-261	Precision	pipx40_resSetResistance
		pipx40_resGetResistance
		pipx40_readCalibrationDate
40-262	Precision	pipx40_resSetResistance
		pipx40_resGetResistance
		pipx40_readCalibrationDate
40-265	Precision	pipx40_resSetResistance
		pipx40_resGetResistance
		pipx40_readCalibrationDate
40-280, 40-281, 40-282	Simple	pipx40_setChannelState
		pipx40_getChannelState
		pipx40_setChannelPattern
		pipx40_getChannelPattern
		pipx40_readCalibration
		pipx40_writeCalibration
40-290, 40-291	Simple	pipx40_setChannelPattern
		pipx40_getChannelPattern
		pipx40_readCalibration
		pipx40_writeCalibration
40-295	Simple	pipx40_setChannelPattern
		pipx40_getChannelPattern

		pipx40_readCalibration
		pipx40_writeCalibration
40-296	Simple	pipx40_setChannelPattern
		pipx40_getChannelPattern
		pipx40_readCalibration
		pipx40_writeCalibration
40-297	Precision	pipx40_resSetResistance
		pipx40_resGetResistance
		pipx40_readCalibrationDate
50-295	Simple	pipx40_setChannelPattern
		pipx40_getChannelPattern
		pipx40_readCalibration
		pipx40_writeCalibration
50-297	Precision	pipx40_resSetResistance
		pipx40_resGetResistance
		pipx40_readCalibrationDate
...		

pipx40_resGetInfo

VB	Function	pipx40_resGetInfo (ByVal vi As Long, ByVal subUnit As Long, ByRef MinRes As Double, ByRef MaxRes As Double, ByRef refRes As Double, ByRef precPC As Double, ByRef precDelta As Double, ByRef int1 As Double, ByRef intDelta As Double, ByRef capabilities As Long) As Long
C++	ViStatus	pipx40_resGetInfo (ViSession vi, ViUInt32 subUnit, ViPReal64 minRes, ViPReal64 maxRes, ViPReal64 refRes, ViPReal64 precPC, ViPReal64 precDelta, ViPReal64 int1, ViPReal64 intDelta, ViPUInt32 capabilities);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
minRes	out	pointer to variable to receive minimum resistance setting
maxRes	out	pointer to variable to receive maximum resistance setting
refRes	out	pointer to variable to receive reference resistance value
precPC	out	pointer to variable to receive percentage precision (+/- percent)
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 Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains detailed information about a programmable resistor sub-unit.

Capabilities flag bit indications are:

Driver constant	**Bit value - hexadecimal**	**Description**
pipx40_RES_CAP_REF	00000008	Supports reference calibration value
pipx40_RES_CAP_INF	00000004	Supports infinity setting
pipx40_RES_CAP_ZERO	00000002	Supports "zero ohms" setting
pipx40_RES_CAP_PREC	00000001	Precision resistor - supporting function pipx40_resSetResistance etc.
pipx40_RES_CAP_NONE	00000000	No special capablities

Remarks

minRes and maxRes are the minimum and maximum values that can be set in the sub-unit's continuous range of adjustment. If capability pipx40_RES_CAP_ZERO is flagged a setting of "zero ohms" is also possible. If pipx40_RES_CAP_INF is flagged an open-circuit setting is also possible.

If capability pipx40_RES_CAP_REF is flagged, refRes is the reference resistance value - such as in model 40-265, where it gives the balanced state resistance.

precPC and precDelta represent the sub-unit's precision specification, such as (±0.2%, ±0.1 ohms).

intDelta is the notional precision to which the sub-unit works internally; this value will be less than or equal to the figure indicated by PrecPC and PrecDelta, indicating greater internal precision.

Where information is not available for the sub-unit concerned, null values are returned.

pipx40_resGetResistance

VB	Function	pipx40_resGetResistance (ByVal vi As Long, ByVal subUnit As Long, ByRef resistance As Double) As Long
C++	ViStatus	pipx40_resGetResistance (ViSession vi, ViUInt32 subUnit, ViPReal64 resistance);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
resistance	in	The current resistance setting, in ohms

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the current resistance setting of the specified programmable resistor. This function is only usable with programmable resistor models that support it; such capability is indicated in the result of pipx40_resGetInfo.

Remarks

The value obtained for a resistance setting of infinity, if the sub-unit permits this, is HUGE_VAL (in C language, #include <math.h>).

pipx40_resSetResistance

VB	Function	pipx40_resSetResistance (ByVal vi As Long, ByVal subUnit As Long, ByVal mode As Long, ByVal resistance As Double) As Long
C++	ViStatus	pipx40_resSetResistance (ViSession vi, ViUInt32 subUnit, ViUInt32 mode, ViReal64 resistance);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
mode	in	The resistance setting mode (see below)
resistance	in	The resistance value, in ohms

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Sets a programmable resistor to the closest available setting to the value specified. This function is only usable with programmable resistor models that support it: such capability is indicated in the result of pipx40_resGetInfo.

mode Value

A value indicating how the given resistance value is to be applied. Only one mode is currently supported:

Value	Ident	Function
0	pipx40_RES_MODE_SET	Set resistance to the specifed value

Remarks

If the sub-unit permits, the resistance value can be set to:

- zero ohms (nominally), by passing the resistance value 0.0
- infinity, by passing the resistance value HUGE_VAL (in C language, #include <math.h>); or alternatively by using function pipx40_clearSub

The resistance value actually set can be found using pipx40_resGetResistance.

In programmable resistor models having gapped ranges, resistance values falling within such gaps are not coerced. For example, in a unit supporting settings:

- zero ohms
- 100 - 200 ohms continuously variable
- infinity

attempting to set values above zero but below 100 ohms, or above 200 ohms but less than infinity, gives error pipx40_ERROR_BAD_RESISTANCE.

Programmable Potentiometer

Programmable Potentiometer

This section details the use of functions specific to programmable potentiometer sub-units.

No potentiometer-specific functions are currently provided.

A potentiometer such as model 40-296 is represented logically as a programmable resistor (RES type) having twice the number of switched bits as its nominal resolution, i.e. a 24-bit potentiometer returns the type description RES(48). To make the unit behave correctly appropriate bit-patterns must be set in the upper and lower halves using general purpose output function pipx40_setChannelPattern. Transient effects must be expected when changing the wiper position; provided pipx40_MODE_NO_WAIT is not in force resistance values can only be transiently high.

Note that a potentiometer's state at power-up and when cleared is as a device of twice the nominal resistance with its wiper centred.

WARNING

Mis-programming can result in the potentiometer presenting a lower than normal resistance between its end terminals - in the worst case zero ohms.

Non-volatile (EEPROM) storage of calibration values is supported through the functions pipx40_readCalibration and pipx40_writeCalibration.

Programmable RF Attenuator

Programmable RF Attenuator

This section details the use of functions specific to programmable RF attenuator sub-units.

Specific functions are provided to:

- Obtain attenuator information, in numeric format: pipx40_attenGetInfo
- Obtain attenuator description, in string format: pipx40_attenGetType
- Set an attenuation level, in dB: pipx40_attenSetAttenuation
- Obtain the current attenuation setting, in dB: pipx40_attenGetAttenuation
- Obtain the value of each individual attenuator pad, in dB: pipx40_attenGetPadValue

pipx40_attenGetAttenuation

VB	Function	pipx40_attenGetAttenuation (ByVal vi As Long, ByVal subUnit As Long, ByRef atten As Single) As Long
C++	ViStatus	pipx40_attenGetAttenuation (ViSession vi, ViUInt32 subUnit, ViPReal32 atten);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
atten	out	The sub-unit's attenuation setting, in dB

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains an attenuator sub-unit's current attenuation setting.

pipx40_attenGetInfo

VB	Function	pipx40_attenGetInfo (ByVal vi As Long, ByVal subUnit As Long, ByRef typeNum As Long, ByRef numSteps As Long, ByRef stepSize As Single) As Long
C++	ViStatus	pipx40_attenGetInfo (ViSession vi, ViUInt32 subUnit, ViPUInt32 typeNum, ViPUInt32 numSteps, ViPReal32 stepSize);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
typeNum	out	pointer to variable to receive type code
numSteps	out	pointer to variable to receive step count
stepSize	out	pointer to variable to receive step size, in dB

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a type description of an attenuator sub-unit, as numeric values.

Attenuator sub-unit type codes are:

Driver constant	typeNum value	**Description**
pipx40_TYPE_ATTEN	8	Programmable RF attenuator

Remarks

The description obtained by this function is a *logical* one; a *physical* description indicating the number of discrete pads in the attenuator can be obtained using pipx40_getSubInfo.

pipx40_attenGetPadValue

VB	Function	pipx40_attenGetPadValue (ByVal vi As Long, ByVal subUnit As Long, ByVal padNum As Long, ByRef atten As Single) As Long
C++	ViStatus	pipx40_attenGetPadValue (ViSession vi, ViUInt32 subUnit, ViUInt32 padNum, ViPReal32 atten);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
padNum	in	The number of the pad whose value is to be queried
atten	out	Pointer to variable to receive the pad's attenuation value, in dB

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the attenuation value associated with an individual pad of an attenuator sub-unit.

Remarks

This function facilitates explicit pad selection using pipx40_setChannelState or pipx40_setChannelPattern, if the selections made by pipx40_attenSetAttenuation are not optimal for the application.

The number of pads in the sub-unit can be found using pipx40_getSubInfo.

pipx40_attenGetType

VB	Function	pipx40_attenGetType (ByVal vi As Long, ByVal subUnit As Long, ByVal subType As String) As Long
C++	ViStatus	pipx40_attenGetType (ViSession vi, ViUInt32 subUnit, ViString subType);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
subType	out	Character string to receive the result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a type description of an attenuator sub-unit, as a text string.

subType string	**Description**
ATTEN(<number of steps>,<step size in dB>)	Programmable RF attenuator

Visual Basic Note

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(subType, character_count).

Remarks

The description obtained by this function is a *logical* one; a *physical* description indicating the number of discrete pads in the attenuator can be obtained using pipx40_getSubType.

pipx40_attenSetAttenuation

VB	Function	pipx40_attenSetAttenuation (ByVal vi As Long, ByVal subUnit As Long, ByVal atten As Single) As Long
C++	ViStatus	pipx40_attenSetAttenuation (ViSession vi, ViUInt32 subUnit, ViReal32 atten);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
atten	in	The attenuation value to set, in dB

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Sets an attenuator sub-unit's attenuation level.

Remarks

The combination of pads inserted to achieve the desired attenuation level is determined by the driver for best all-round performance. In some models it may be possible to optimise particular aspects of attenuator performance by setting other pad combinations explicitly using pipx40_setChannelState or pipx40_setChannelPattern. The pad value associated with each output channel can be discovered with pipx40_attenGetPadValue.

Power Supplies

Power Supplies

This section details the use of functions specific to power supply sub-units.

Specific functions are provided to:

- Obtain power supply description, in string format: pipx40_psuGetType
- Obtain power supply information, in numeric format: pipx40_psuGetInfo
- Set power supply output voltage: pipx40_psuSetVoltage
- Obtain a power supply's output voltage setting: pipx40_psuGetVoltage
- Enable/disable a power supply's output: pipx40_psuEnable

Other functions that are relevant to operation of power supply sub-units include:

- Clear a power supply (restore start-up state): pipx40_clearSub
- Obtain power supply status information: pipx40_getSubStatus
- Retrieve a calibration value from non-volatile memory (some models): pipx40_readCalibration
- Store a calibration value in non-volatile memory (some models): pipx40_writeCalibration

pipx40_psuEnable

VB	Function	pipx40_psuEnable (ByVal vi As Long, ByVal subUnit As Long, ByVal state As Boolean) As Long
C++	ViStatus	pipx40_psuEnable (ViSession vi, ViUInt32 subUnit, ViBoolean state);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
state	in	A Boolean indicating type of action, VI_ON to enable, VI_OFF to disable

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Enables or disables the output of a power supply sub-unit.

pipx40_psuGetInfo

VB	Function	pipx40_psuGetInfo (ByVal vi As Long, ByVal subUnit As Long, ByRef typeNum As Long, ByRef voltage As Double, ByRef current As Double, ByRef precision As Long, ByRef capabilities As Long) As Long
C++	ViStatus	pipx40_psuGetInfo (ViSession vi, ViUInt32 subUnit, ViPUInt32 typeNum, ViPReal64 voltage, ViPReal64 current, ViPUInt32 precision, ViPUInt32 capabilities);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
typeNum	out	pointer to variable to receive type code
voltage	out	pointer to variable to receive voltage rating
current	out	pointer to variable to receive current rating
precision	out	pointer to variable to receive precision (the number of bits resolution - for programmable supplies only)
capabilities	out	pointer to variable to receive capability flags - see below

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a type description of a power supply sub-unit, as numeric values.

Power supply sub-unit type codes are:

Driver constant	**typeNum value**	**Description**
pipx40_TYPE_PSUDC	9	Power supply, DC

The capabilities value is the sum of a number of individual bit-flags, as follows:

Driver constant	**Bit value - hexadecimal**	**Description**
pipx40_PSU_CAP_CURRENT_MODE_SENSE	00000010	Can sense if operating in current-limited mode
pipx40_PSU_CAP_PROG_CURRENT	00000008	Output current is programmable
pipx40_PSU_CAP_PROG_VOLTAGE	00000004	Output voltage is programmable
pipx40_PSU_CAP_OUTPUT_SENSE	00000002	Has logic-level sensing of output active state
pipx40_PSU_CAP_OUTPUT_CONTROL	00000001	Has output on/off control

Certain driver functions are only usable with power supply sub-units having appropriate capabilities - examples being:

pipx40_psuEnable

pipx40_psuSetVoltage

pipx40_psuGetType

VB	Function	pipx40_psuGetType (ByVal vi As Long, ByVal subUnit As Long, ByVal subType As String) As Long
C++	ViStatus	pipx40_psuGetType (ViSession vi, ViUInt32 subUnit, ViString subType);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
subType	out	Character string to receive the result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a type description of a power supply sub-unit, as a text string.

subType string	**Description**
PSUDC(<voltage rating>,<current rating>)	Power supply, DC

Visual Basic Note

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(subType, character_count).

pipx40_psuGetVoltage

VB	Function	pipx40_psuGetVoltage (ByVal vi As Long, ByVal subUnit As Long, ByRef voltage As Double) As Long
C++	ViStatus	pipx40_psuGetVoltage (ViSession vi, ViUInt32 subUnit, ViPReal64 voltage);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
voltage	out	The sub-unit's output voltage setting

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a power supply sub-unit's current output voltage setting.

The result is the nominal value to which the output has been set, not necessarily the actual voltage being output (which may be affected by device tolerances, current-limit conditions etc.).

This function is also usable with fixed-voltage supplies, returning the nominal output voltage.

pipx40_psuSetVoltage

VB	Function	pipx40_psuSetVoltage (ByVal vi As Long, ByVal subUnit As Long, ByVal voltage As Double) As Long
C++	ViStatus	pipx40_psuSetVoltage (ViSession vi, ViUInt32 subUnit, ViReal64 voltage);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
voltage	in	The output voltage value to set

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Sets a power supply sub-unit's output voltage.

The voltage value specified is rounded to the precision of the supply's DAC. The actual voltage setting can be obtained using pipx40_psuGetVoltage.

This function is usable only with sub-units having the capability pipx40_PSU_CAP_PROG_VOLTAGE - see pipx40_psuGetInfo.

Battery Simulator

Battery Simulator

This section details the use of functions specific to battery simulator models.

Models 41-750-001 and 41-751-001

No special-purpose functions are implemented for these models - they are operable using general-purpose input-output functions. See:

40-750-001

40-751-001

Model 41-752-001

Model 41-752-001 is implemented as an array of BATT sub-units, employing the following special-purpose functions for normal operation:

- Set output voltage: pipx40_battSetVoltage
- Obtain the present output voltage setting: pipx40_battGetVoltage
- Set sink current: pipx40_battSetCurrent
- Obtain the present sink current setting: pipx40_battGetCurrent
- Set output enable states: pipx40_battSetEnable
- Obtain present output enable states: pipx40_battGetEnable
- Obtain the present state of the hardware interlock: pipx40_battReadInterlockState

pipx40_battSetVoltage

VB	Function	pipx40_battSetVoltage (ByVal vi As Long, ByVal subUnit As Long, ByVal voltage As Double) As Long
C++	ViStatus	pipx40_battSetVoltage (ViSession vi, ViUInt32 subUnit, ViReal64 voltage);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
voltage	in	The output voltage value to set

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Sets battery simulator output voltage.

When subUnit corresponds to a BATT sub-unit, the function sets the voltage of that sub-unit alone.

If subUnit = 0 (pipx40_BATT_ALL_BATT_SUB_UNITS), all of the card's BATT sub-units are set to the given voltage.

The voltage value specified is rounded to the precision of the sub-unit's DAC. The actual voltage setting can be obtained using pipx40_battGetVoltage.

pipx40_battGetVoltage

VB	Function	pipx40_battGetVoltage (ByVal vi As Long, ByVal subUnit As Long, ByRef voltage As Double) As Long
C++	ViStatus	pipx40_battGetVoltage (ViSession vi, ViUInt32 subUnit, ViPReal64 voltage);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
voltage	out	The sub-unit's output voltage setting

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a battery simulator (BATT type) sub-unit's output voltage setting, as set by pipx40_battSetVoltage.

The result is the nominal value to which the output has been set, not necessarily the actual voltage being output (which could be affected by conditions such as current-limiting).

pipx40_battSetCurrent

VB	Function	pipx40_battSetCurrent (ByVal vi As Long, ByVal subUnit As Long, ByVal current As Double) As Long
C++	ViStatus	pipx40_battSetCurrent (ViSession vi, ViUInt32 subUnit, ViReal64 current);

Parameter	I/O	Description
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
current	in	The output sink current value to set

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Sets battery simulator output sink current.

When subUnit corresponds to a BATT sub-unit, the function sets the sink current of that sub-unit alone.

If subUnit = 0 (pipx40_BATT_ALL_BATT_SUB_UNITS), all of the card's BATT sub-units are set to the given current.

For non-zero values, output sink current is set to the nearest available value **greater** than that specified, typically using a low-precision DAC (e.g. 4-bit). The actual sink current setting can be obtained using pipx40_battGetCurrent.

pipx40_battGetCurrent

VB	Function	pipx40_battGetCurrent (ByVal vi As Long, ByVal subUnit As Long, ByRef current As Double) As Long
C++	ViStatus	pipx40_battGetCurrent (ViSession vi, ViUInt32 subUnit, ViPReal64 current);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
current	out	The sub-unit's output sink current setting

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains a battery simulator (BATT type) sub-unit's output sink current setting, as set by pipx40_battSetCurrent.

pipx40_battSetEnable

VB	Function	pipx40_battSetEnable (ByVal vi As Long, ByVal subUnit As Long, ByVal pattern As Long) As Long
C++	ViStatus	pipx40_battSetEnable (ViSession vi, ViUInt32 subUnit, ViUInt32 pattern);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
pattern	in	The output enable pattern to set

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Enables/disables battery simulator outputs.

When subUnit corresponds to a BATT sub-unit, the function sets the output enable state of that sub-unit alone according to the least significant bit of Pattern (0 = OFF, 1 = ON).

If subUnit = 0 (pipx40_BATT_ALL_BATT_SUB_UNITS), enable states of all the card's BATT sub-units are set; bits in the supplied pattern are utilised in ascending order of BATT sub-unit, i.e.

Pattern bit 0 = enable state of lowest numbered BATT sub-unit (0 = OFF, 1 = ON)

Pattern bit 1 = enable state of next numbered BATT sub-unit (0 = OFF, 1 = ON)

etc.

Note that the operation can fail (returning pipx40_ERROR_EXECUTION_FAIL) if a necessary hardware interlock is disconnected.

The present enable pattern can be obtained using pipx40_battGetEnable.

pipx40_battGetEnable

VB	Function	pipx40_battGetEnable (ByVal vi As Long, ByVal subUnit As Long, ByRef pattern As Long) As Long
C++	ViStatus	pipx40_battGetEnable (ViSession vi, ViUInt32 subUnit, ViPUInt32 pattern);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating the sub-unit for which information is to be obtained
pattern	out	The sub-unit's output enable pattern

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Obtains the enabled/disabled status of battery simulator sub-units.

When subUnit corresponds to a BATT sub-unit, the function gets the output enable state of that sub-unit alone in the least significant bit of Pattern (0 = OFF, 1 = ON).

If subUnit = 0 (pipx40_BATT_ALL_BATT_SUB_UNITS), enable states of all the card's BATT sub-units are obtained; bits in the pattern are assigned in ascending order of BATT sub-unit, i.e.

> pattern bit 0 = enable state of lowest numbered BATT sub-unit (0 = OFF, 1 = ON)
>
> pattern bit 1 = enable state of next numbered BATT sub-unit (0 = OFF, 1 = ON)

etc.

pipx40_battReadInterlockState

VB	Function	pipx40_battReadInterlockState (ByVal vi As Long, ByVal subUnit As Long, ByRef interlock As Boolean) As Long
C++	ViStatus	pipx40_battReadInterlockState (ViSession vi, ViUInt32 subUnit, ViPBoolean interlock);

Parameter	**I/O**	**Description**
vi	in	Instrument handle
subUnit	in	Numeric variable indicating in which sub-unit actions will take place
interlock	out	Pointer/reference to variable to receive result

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Reads battery simulator hardware interlock state.

When SubNum corresponds to a BATT sub-unit, the function gets the state of the hardware interlock associated with that sub-unit:

0 = VI_OFF = interlock is "down"

1 = VI_ON = interlock is "up"

If SubNum = 0 (pipx40_BATT_ALL_BATT_SUB_UNITS), the function gets the summary state of all BATT sub-unit interlocks:

0 = VI_OFF = one or more interlocks is "down"

1 = VI_ON = all interlocks are "up"

Model 41-752-001 has a single global interlock affecting all channels, and both modes above yield the same result.

Interlock "up" state is hardware-latched from the physical wired interlock by the action of pipx40_battSetEnable, when that function succeeds. Hence:

- If the "up" state is indicated, the physical interlock has remained intact and outputs are enabled as previously set by pipx40_battSetEnable.
- If the "down" state is indicated, the physical interlock has been broken and all outputs will have been disabled automatically through hardware.

Mode Control

Mode Control

This section details the use of functions controlling the driver's operation.

This feature is implemented through a single function: pipx40_setDriverMode.

pipx40_setDriverMode

VB	Function	pipx40_setDriverMode (ByVal newMode As Long, ByRef previousMode As Long) As Long
C++	ViStatus	pipx40_setDriverMode (ViUInt32 newMode, ViPUInt32 previousMode);

Parameter	**I/O**	**Description**
newMode	in	New value for driver mode flags
previousMode	out	The driver's mode flags prior to executing this function.

Return Value

0 = Successful operation. Negative values are error codes and positive values are warnings. To get a description of the error, pass the error code to pipx40_error_message.

Description

Allows control flags affecting the driver's global behaviour to be set and read. This function gives access to low-level control features of the pipx40 driver and is intended for 'expert' use only - the default driver behaviour should be satisfactory for the great majority of applications.

Remarks

Individual bits in the mode value control various aspects of driver operation.

Setting the value zero (pipx40_MODE_DEFAULT) clears all special driver modes.

Control bit values may be summed to enable multiple driver features.

Driver constant	**Bit value - hexadecimal**	**Description**
pipx40_MODE_NO_WAIT	00000001	Function calls return without waiting for card operations to complete

pipx40_MODE_UNLIMITED	00000002	Disable maximium closure limits - see **Warning** below
pipx40_MODE_IGNORE_TEST	00000008	Enable card operation even if selftest fails - see **Warning** below

Warning - pipx40_MODE_UNLIMITED

Use of pipx40_MODE_UNLIMITED to disable the maximum number of crosspoint closures permitted on high-density cards is **not** recommended, because it carries the danger of overheating and consequent damage to both the card itself and the system in which it is installed. See Closure Limits.

Warning - pipx40_MODE_IGNORE_TEST

The pipx40_MODE_IGNORE_TEST feature should be used with **extreme caution**. If a defective card is forcibly enabled, under some fault conditions a large number of outputs could be energised spuriously, resulting in overheating and consequent damage to both the card itself and the system in which it is installed. The intended purpose of this feature is to allow continued operation of a BRIC unit from which a daughtercard has been removed for maintenance. See BRIC Operation.

Utility Programs

Utility Programs

The pipx40 driver is supported by a number of utility programs:

- Test Panels
- Terminal Monitor
- Diagnostic Utility

Test Panels

The Test Panels application allows any combination of cards to be controlled using a graphical interface.

Please note that the Test Panels access cards using the Pilpxi Direct I/O driver, so their I/O operations are not visible to VISA-based utilities such as NI-Spy.

Terminal Monitor

PILMon is a simple terminal monitor program for Pickering PXI cards. Use the HE command within PILMon to obtain help.

PILMon has a number of command-line options when starting the program. For instructions, in a Command Prompt window with the current directory set to that containing PILMon, type:

PILMON -?

```
C:\Pickering\Utils>pilmon -?

Program:   PIL PXI Monitor

Syntax:    PILMon [-cN] [-r] [-n]

Arguments: -cN specifies the number of the COM port (1 thru 9) to use

                in lieu of the console. COM settings are 9600/8/N/1.

           -r specifies that when run PILMon should attempt to open

              the cards without clearing them. This may or may not be

              possible.

           -n specifies that when run PILMon should NOT automatically

              open the cards. Overrides -r if both are used.

Options are accepted in any order.

Example:   PILMon -c2 -r -n
```

Please note that PILMon accesses cards using the Pilpxi Direct I/O driver, so its I/O operations are not visible to VISA-based utilities such as NI-Spy.

Although it does not employ the VISA driver, the action of many PILMon commands corresponds closely to pipx40 card specific functions:

--Card Specific Functions--	**Corresponding PILMon command**

Card ID, Properties & Status		
Get card ID	pipx40_getCardId	See note
Get card status	pipx40_getCardStatus	ST
Get closure limit	pipx40_getClosureLimit	CL
Get diagnostic information	pipx40_getDiagnostic	DI
Get settling time	pipx40_getSettlingTime	SE
Get card sub-unit counts	pipx40_getSubCounts	See note
Get sub-unit description (string format)	pipx40_getSubType	See note
Get sub-unit description (numeric format)	pipx40_getSubInfo	See note
Output control		
Clear all channels of a card	pipx40_clearCard	AR
Clear all channels of a sub-unit	pipx40_clearSub	CS
Turn on/off a single channel	pipx40_setChannelState	SC and SO
Turn on/off of a matrix crosspoint	pipx40_setCrosspointState	XC and XO
Set a sub-unit's channel pattern	pipx40_setChannelPattern	SB
Get the state of a single channel	pipx40_getChannelState	SV
Get the state of a matrix crosspoint	pipx40_getCrosspointState	XV
Get a sub-unit's channel pattern	pipx40_getChannelPattern	BV
Output masking		
Clear a sub-unit's mask	pipx40_clearMask	CM
Mask/unmask a single channel	pipx40_setMaskState	SM
Mask/unmask a matrix crosspoint	pipx40_setCrosspointMask	XM
Set a sub-unit's mask pattern	pipx40_setMaskPattern	MB
Get the mask state of a single channel	pipx40_getMaskState	MS
Get the mask state of a matrix crosspoint	pipx40_getCrosspointMask	XS
Get a sub-unit's mask	pipx40_getMaskPattern	MV

pattern		
Output calibration		
Read a channel's calibration value	pipx40_readCalibration	RC
Write a channel's calibration value	pipx40_writeCalibration	WC
Input		
Read the state of a single input	pipx40_readInputState	IS
Read a sub-unit's input pattern	pipx40_readInputPattern	BR
Mode control		
Set driver operating mode	pipx40_setDriverMode	DM

Note

Where noted, the information obtained by this function is displayed as part of the output from the PILMon LS command; though the Pilpxi card identification string omits the "PICKERING INTERFACES," manufacturer identification that is returned by pipx40_getCardId.

Diagnostic Utility

The Plug & Play functionality of PXI cards generally ensures trouble-free installation. However in the event of any problems, it may be helpful to know how cards have been configured in the system. The PipxDiag Windows diagnostic utility generates an extensive report showing the allocations of PCI/PXI system resources and specific details of installed Pickering cards, highlighting any potential configuration issues.

In the diagnostic report, all the installed Pickering cards should be listed in the "Pilpxi information" section - if one or more cards is missing it may be possible to determine the reason by referring to the PCI configuration dump contained in the report, but interpretation of this information is far from straightforward, and the best course is to contact Pickering support: support@pickeringtest.com, if possible including a copy of the diagnostic report.

In the "VISA information" section, if VISA is not installed its absence will be reported. The pipx40 driver cannot function without VISA. VISA is a component of National Instruments LabWindows/CVI and LabVIEW, or is available as a standalone environment. If the installed VISA version is reported as too old to operate Pickering cards, you should contact National Instruments for an updated version - upgrades are normally available from the National Instruments website http://www.ni.com.

If VISA is present and is of a sufficiently recent version, the section "Pipx40 information" should contain a listing similar to "Pilpxi information".

Please note that the Diagnostic Utility cannot access cards if they are currently opened by some other application, such as the Test Panels or Terminal Monitor.

Application Notes

Application Notes

This section contains application notes on the following topics:

- BRIC Operation
- Closure Limits
- Execution Speed
- Isolation Switching
- Multiprocessing and Multithreading
- Simple programmable Resistor Cards
- Segmented Matrix
- Unsegmented Matrix

BRIC Operation

BRIC closure limits

As with other high-density units, for a BRIC the pipx40 driver imposes a limit on the maximum number of channel closures - see Closure Limits. Although pipx40_setDriverMode offers a means of disabling this limit, the extraordinarily high packing density in BRIC units makes observation of maximum closure limits particularly important. The consequences of turning on an excessive number of crosspoints can be appreciated from the fact that each activated crosspoint may consume around 10mA at 5V (50mW, or 1W per 20 crosspoints). The power consumption of a large BRIC with all crosspoints energised would be beyond the capacity of the system power supply and backplane connectors, never mind its cooling capabilities. For this reason BRIC units are fuse-protected against overcurrent. However, it cannot protect against local hot-spots within a BRIC if too large a block of physically adjacent crosspoints is energised. Although the fuse is self-resetting under moderate overload, a massive overload may cause it to rupture permanently.

BRIC daughtercard removal

In the event of a BRIC daughtercard being removed for servicing, operation of the entire unit is normally disabled. It is possible to allow continued operation in spite of this fault condition using the pipx40_MODE_IGNORE_TEST option bit in pipx40_setDriverMode. When this mode is set, the tests performed when the card is opened will still detect the fault and flag it in the card's pipx40_getCardStatus value (bit pipx40_STAT_HW_FAULT = set); however it will no longer be flagged as disabled (bit pipx40_STAT_DISABLED = clear), allowing continued operation.

Multifunction BRICs

Multifunction BRICs have independently controlled isolation switches. In operating these units it is advised that where hot-switching occurs programmers ensure that matrix crosspoint relays hot-switch, and isolation relays cold-switch. This avoids concentrating the contact wear caused by hot-switching in the isolation relays, which could lead to a reduction in their operational life. The preferred operating sequences for hot-switching are:

- When closing a crosspoint, first close the isolation switch, then the crosspoint switch
- When opening a crosspoint, first open the crosspoint, then the isolation switch

Closure Limits

The high switch density attained in certain System 40/45/50 cards, particularly high-density matrix types, necessitates close packing of relays and airflow is quite restricted. If excessive numbers of relays were energised for a prolonged period overheating could occur. For example, in model 40-531 simultaneous energisation of all 256 relays would yield a power dissipation of around 17W. In BRIC units the situation is even more extreme - see BRIC Operation. To guard against this danger the software driver places a limit on the number of crosspoints that can be energised simultaneously. The limits imposed by the driver are set with regard to operating temperature levels and will not cause any difficulty for typical matrix usage, where only a small proportion of crosspoints are simultaneously ON. A sub-unit's closure limit can be discovered using the pipx40_getClosureLimit function.

In some models, energisation of too many relays would cause the card's supply current to exceed the maximum available from the system backplane, with the potential for overheating and damage to the card and backplane connectors.

The software driver does however provide a method of disabling this protection. Calling the function pipx40_setDriverMode with the bit pipx40_MODE_UNLIMITED set allows an unlimited number of crosspoints to be energised simultaneously. This feature should be used with **EXTREME CAUTION**. Although it may be safe to energise larger numbers of crosspoints where ON times are short and duty cycle is low, it must be borne in mind that if the user's program were to halt in the ON state (for example at a breakpoint when debugging) the danger of overheating is present.

Some models incorporate fuses to protect against simultaneous activation of a hugely excessive number of channels. These are self-resetting in moderate overload, and operation will be restored when the fault condition clears.

Execution Speed

Internal optimisations

Generally, the pipx40 driver optimises a card's internal switch operations as far as possible. For example in a single-channel multiplexer (MUX type) with isolation switching, if a channel-change is requested the isolation switch is not cycled. This saves both time and mechanical wear on the switch.

Break-before-make action

By default, the pipx40 driver enforces Break-Before-Make (BBM) action and settling delays (to cope with contact bounce) on all switching operations. This ensures 'clean' switching actions and minimises the danger of switch damage due to conflicting contact closures.

For time-critical applications the driver can be set to omit all sequencing delays using the pipx40_MODE_NO_WAIT option of pipx40_setDriverMode. This causes the driver to return control to the application program in the shortest possible time. The function pipx40_getCardStatus can then be used at a later time to determine when operations on a particular card have completed (indicated by the bit pipx40_STAT_BUSY becoming clear). By this method a number of switching operations (and/or other program activity) can be executed in parallel rather than sequentially. However the programmer must guard against switch conflicts that might transiently cause, say, the shorting of a power supply and consequent switch damage.

In some cards (for example model 40-745), making an individual channel selection involves several physical relays. Normally, sequencing delays are imposed to ensure that no unwanted transient connections occur. Setting pipx40_MODE_NO_WAIT bypasses these delays, and the programmer must bear in mind the potential for transient conflicts.

Default driver action is restored by executing pipx40_setDriverMode with the pipx40_MODE_NO_WAIT bit clear.

Many System 40/45/50 relay cards exhibit very short basic execution times in the order of a few tens of microseconds; however BBM and settling delays associated with relays may extend from a few hundred microseconds (for small reed relays) to some tens of milliseconds (for microwave switches). Here, setting pipx40_MODE_NO_WAIT and appropriate programming can free a significant amount of CPU time for other purposes.

There are some exceptions to the above: for example digital outputs generally have zero settling time and pipx40_MODE_NO_WAIT offers no performance advantage.

To summarise, where execution speed is of paramount importance setting pipx40_MODE_NO_WAIT can offer significant advantages for many cards; however it is more demanding for the programmer, requiring an understanding of the operational characteristics of specific card types and taking greater account of conditions in the switched circuits.

Processor speed

A faster processor might be expected to yield faster operation. However for many cards much of a function's execution time is spent waiting for switch contacts to stabilise, so unless pipx40_MODE_NO_WAIT is invoked little improvement will be seen. Further, modern processors are capable of operating many cards near or beyond their hardware limits, and the pipx40 driver includes timing control to ensure reliable operation. Therefore increases in processor speed beyond about 3GHz may well give no actual improvement in operating speed.

Isolation Switching

Isolation switching is incorporated in particular models for a variety of reasons:

- Reducing capacitive loading on a node. In low-frequency units, reduced capacitive loading gives faster response times when medium to high impedance signals are being carried.
- Reducing circuit leakage current. Reduced leakage current in the switch circuits is advantageous where low-current measurements are involved.
- Reducing the length of circuit stubs on a node. In high-frequency units, reduced stub lengths give better RF performance.
- Providing alternate switching functionality. Some versatile models utilise isolation switching to support additional operating modes.

Automatic isolation switching

Isolation and loopthru switches are normally controlled automatically by the pipx40 driver, and their operation is entirely transparent to the user.

In some applications or for fault diagnostic purposes it may be desirable to control isolation and loopthru switches independently. There are two ways of achieving this:

1. In matrix types having auto-isolation and/or auto-loopthru, function pipx40_operateSwitch permits explicit control of individual switches.
2. Cards can usually be reconfigured to allow independent control of isolation or loopthru switches using the ordinary control functions - if you have such a requirement please contact support@pickeringtest.com.

Multiprocessing and Multithreading

Multiprocessing involves operation of cards by multiple software processes (i.e. programs); multithreading uses multiple execution threads within a single program. Multithreading is a feature of programming environments such as LabVIEW, and can also be managed through the standard Windows API.

Process-safety

The pipx40 driver is process-safe.

Note that a card is automatically cleared when opened by pipx40_init, irrespective of the value of the reset_instr parameter. The reason for this is that on initialisation a card has no means of reporting its current output state to the driver, which must therefore initialise it to a known state.

Thread-safety

The pipx40 driver is thread-safe.

Execution of a pipx40 driver function by one thread simply blocks its execution by other threads or processes. This includes any settling delay periods, ensuring that no unwanted overlaps occur in operation.

Function pipx40_setDriverMode

The settings made by pipx40_setDriverMode are process-specific, i.e. multiple processes can operate with different settings.

Simple Programmable Resistor Cards

Applicable to models:

- 40-290
- 40-291
- 40-295
- 40-296
- 50-295

Simple programmable resistor cards employ a series chain of individual fixed resistors, each having an associated shorting switch. In standard models the fixed resistor values are arranged in a binary sequence. The discussion below relates to 16-bit models; some considerations may be either more or less significant in models with higher or lower resolution.

Application considerations: 16-bit models

The binary resistor chain employed in a 16-bit programmable resistor card provides a notional resolution of about 0.002% (or 15ppm) of the total resistance.

In exploiting this high resolution there are a number of factors which should be taken into account:

- The absolute accuracy of the resistors fitted may be only 1% or 0.5% (i.e. less than 8 bits).
- For 'custom' resistor-chain values, components having the precise nominal values required may be unobtainable, and the nearest available preferred values may have to be used.
- The resistors have a non-zero temperature coefficient, typically of ±50ppm/°C, though values down to ±15ppm/°C may be obtainable.
- The closed-contact resistance of the switch shunting each resistor is of the order of 100 milliohms. In the reed switches employed in these cards it is highly stable, provided they are not subjected to overcurrent. This includes transient currents, such as discharging a long cable that is pre-charged to a significant voltage.
- Wiring and connectors impose a small resistance in series with the resistor chain, of perhaps 200 milliohms.

Some implications of these factors are:

- The relationship between the switch pattern and the programmed resistance value is not guaranteed to be monotonic (i.e. a change in switch pattern that might be expected to yield an increase in resistance value may in fact decrease it, and vice-versa).

- A resistance value of zero ohms is unobtainable. The lowest value that can be achieved is composed of the closed-contact resistances of 16 relays in series, together with wiring and connector resistance. A value of around 1.8 ohms is typical.
- Temperature effects can significantly exceed the notional resolution. For example, a temperature change of only 5°C may cause a resistance change of ±250ppm, or 17 times the notional resolution. The resistance of wiring and closed switch contacts is also affected by temperature.

The cards have the facility to store in non-volatile memory a 16-bit value associated with each resistor. These values can be used to calibrate the card to provide greater setting accuracy than the basic absolute accuracy of the resisors employed in the chain. Usage and interpretation of stored values is entirely user-specific: the software driver merely provides a mechanism (functions pipx40_writeCalibration and pipx40_readCalibration) for storing and retrieving them.

A possible scheme for utilising the stored calibration values might be:

- Employ the stored values to somehow represent the deviation of each resistor's actual value from its nominal value (say, as a percentage: treated as a signed quantity the 16-bit value might be chosen to represent a range of ±32.767%).
- Use a calibration procedure to obtain and store an appropriate value for each individual resistor.
- Software must then make use of the stored calibration data when programming specific resistance values, taking into account extraneous circuit resistances. Because of the non-monotonic relationship between switch pattern and resistance value, some calculation is necessary to obtain a pattern matching a chosen value. A simple C program ProgResFind.c demonstrates a possible approach to this.

ProgResFind.c

This program demonstrates a possible algorithm for use in obtaining a specific resistance value in a 16-bit programmable resistor card, using stored calibration values for enhanced accuracy.

```
/* Program: ProgResFind.c */

/* Programmable resistor: find a 16-bit code to give a particular
resistance value */

/* D.C.H  16/8/01 */

/* Overall accuracy is determined by the accuracy of the calibration
values employed */


#include <stdio.h>


/* To  output debug info... */

/* *** #define DEBUG */


/* === SEARCH VALUES
====================================================== */

/* The resistance value to search for, ohms */

double search_res = 1000.0;

/* The required tolerance (fractional) */

double search_tol    = 0.0005;        /* = 0.05% */


/* === CALIBRATION VALUES
================================================= */

/* Offset resistance value, ohms: includes connector and wiring.

   This example includes a 50R offset resistor. */

/* For accuracy, this should ideally be a CALIBRATED value */

double res_offset = 50.2;
```

```
/* The installed resistor values, ohms */

/* For accuracy better than resistor tolerance these must be
CALIBRATED values,

   not NOMINAL ones. */

double res_value[16] =

{

    0.12,

    0.22,

    0.56,

    1.13,

    2.26,

    4.42,

    8.2,

    18.0,

    37.4,

    71.5,

    143.0,

    287.0,

    576.0,

    1130.0,

    2260.0,

    4530.0

};



/* Relay closed-contact resistance, ohms: assumed identical for all
relays */

double res_contact = 0.1;

/*
======================================================================
=== */
```

```
/* Prototype */

long find_code(double value, double tolerance);


int main(void)

{

    long code;

    printf("Programmable Resistor Code Finder\n");

    printf("=================================\n");

    printf("D.C.H  16/8/01\n\n");

    printf("Search for %8.2f ohms (+/- %1.3f%%)...\n", search_res,
search_tol * 100);

    code = find_code(search_res, search_tol);

    if (code < 0)

        printf("No code matches this value within the specified
tolerance\n");

    else

        printf("Code 0x%04X\n", code);

    return 0;

}


/* Function: parallel resistor calculation */

double parallel_resistance(double r1, double r2)

{

    return ((r1 * r2) / (r1 + r2));

}


/* Function: find the first code whose actual value  matches the
search value
```

```
   within the specified tolerance band.

   Returns the code (0x0000 thru 0xFFFF).

   If no code generates a value that lies within the specified
tolerance band,

   returns -1.

   The method simply searches all codes - some optimisation is
possible. */

long find_code(double value, double tolerance)

{

    long code;

    int bit;

    double res;

    /* Search all codes */

    for (code = 0; code < 0x10000L; code++)

    {

        res = res_offset;

        for (bit = 0; bit < 16; bit++)

        {

            if (code & (1 << bit))

            {

                /* This bit is ON (switch closed) */

                res += parallel_resistance(res_value[bit],
res_contact);

            }

            else

            {

                /* This bit is OFF (switch open) */

                res += res_value[bit];

            }
```

```
        }

        if ( res > (value * (1.0 - tolerance)) && res < (value * (1.0
+ tolerance)) )

        {

#ifdef DEBUG

            printf("Code 0x%04X = %8.2f ohms\n", code, res);

#endif

            return code;

        }

    }

    return -1;

}
```

Segmented Matrix

Segmented Matrix

A segmented matrix is one in which groups of lines on an axis are served by separate sets of isolation switches on the opposing axis.

When operated by functions such as:

- pipx40_setChannelState
- pipx40_setChannelPattern
- pipx40_setCrosspointState

isolation switching is handled automatically by the driver, and the sub-unit's internal structure is immaterial to a user; use of pipx40_operateSwitch however requires an understanding of this.

Examples of segmented matrix types:

- 40-725-511: 8 x 9, segmented on both axes
- 40-726-751-LT: 12 x 8, segmented on both axes, with loopthru on Y-axis only
- 40-560-021: 50 x 8 specimen BRIC configuration, segmented on X-axis (Y-isolation only)

Segmented Matrix 40-725-511

40-725-511 is an 8 x 9 matrix, segmented on both axes.

In its standard configuration as a single 8 x 9 matrix sub-unit, when channel selections are made using functions such as:

- pipx40_setChannelState
- pipx40_setChannelPattern
- pipx40_setCrosspointState

operation of isolation switches is automated to optimise connections for X - Y signal routing. pipx40_operateSwitch allows access to individual switches for other routing schemes or fault diagnostic purposes.

Note that an alternate logical configuration treats the card as multiple sub-units, giving independent access to all switches via the ordinary control functions: for that configuration pipx40_operateSwitch is not applicable.

Attribute values

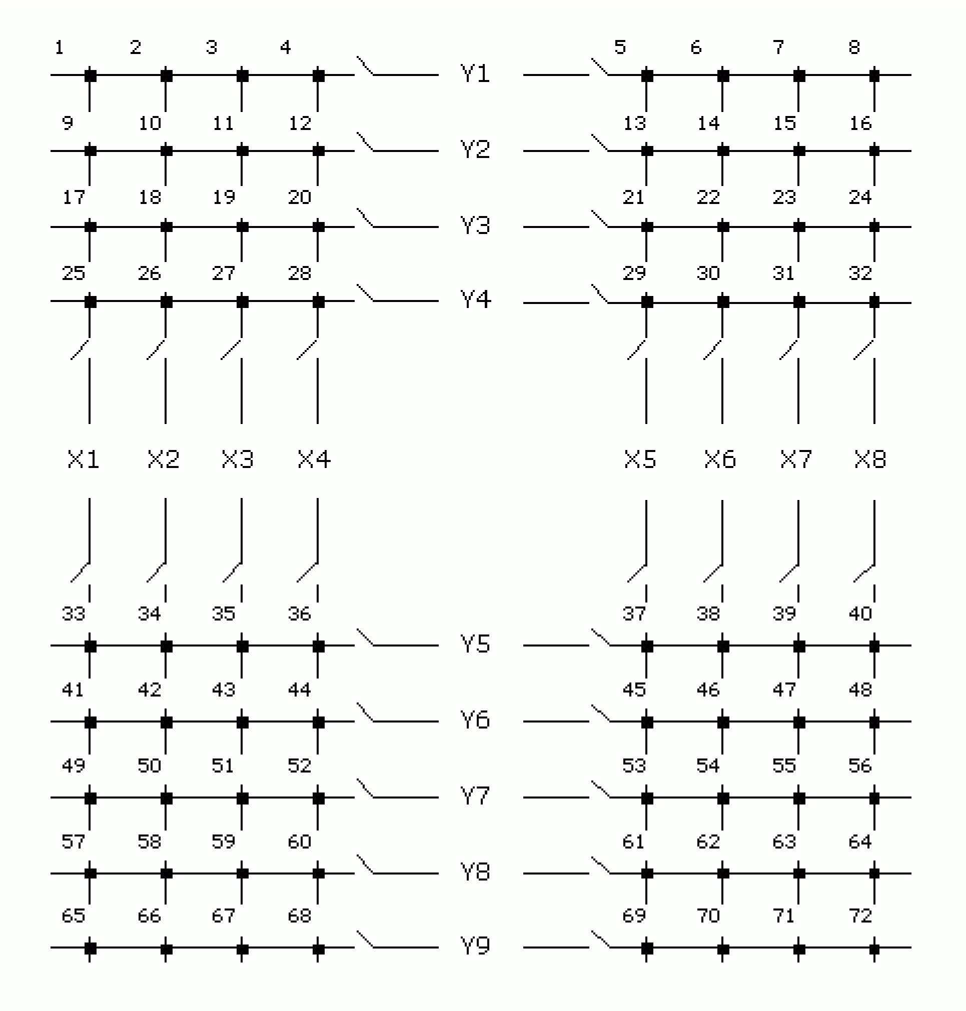
The relevant values obtained by pipx40getSubAttribute when configured for auto-isolation are:

Attribute code	Attribute value
pipx40_SUB_ATTR_CHANNEL_SUBSWITCHES	1
pipx40_SUB_ATTR_X_ISO_SUBSWITCHES	1
pipx40_SUB_ATTR_Y_ISO_SUBSWITCHES	1
pipx40_SUB_ATTR_NUM_X_SEGMENTS	2
pipx40_SUB_ATTR_X_SEGMENT01_SIZE	4
pipx40_SUB_ATTR_X_SEGMENT02_SIZE	4
pipx40_SUB_ATTR_NUM_Y_SEGMENTS	2
pipx40_SUB_ATTR_Y_SEGMENT01_SIZE	4
pipx40_SUB_ATTR_Y_SEGMENT02_SIZE	5

Global crosspoint switch numbers

These numbers correspond to the channel numbers used with pipx40_setChannelState and are valid for pipx40_operateSwitch when:

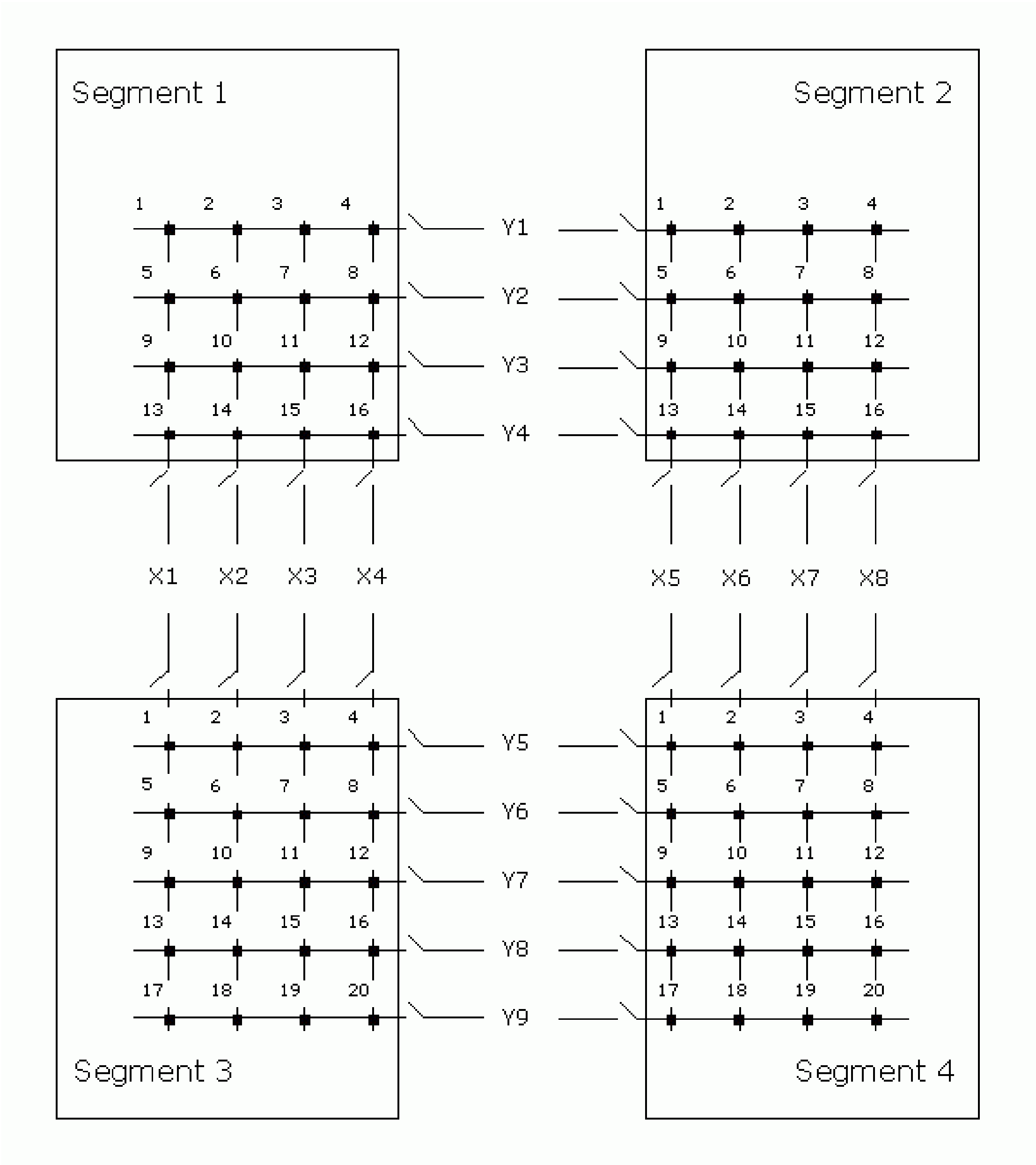
- switchFunc = pipx40_SW_FUNC_CHANNEL
- segNum = 0



Segment-local crosspoint switch numbers

These switch numbers are valid for pipx40_operateSwitch when:

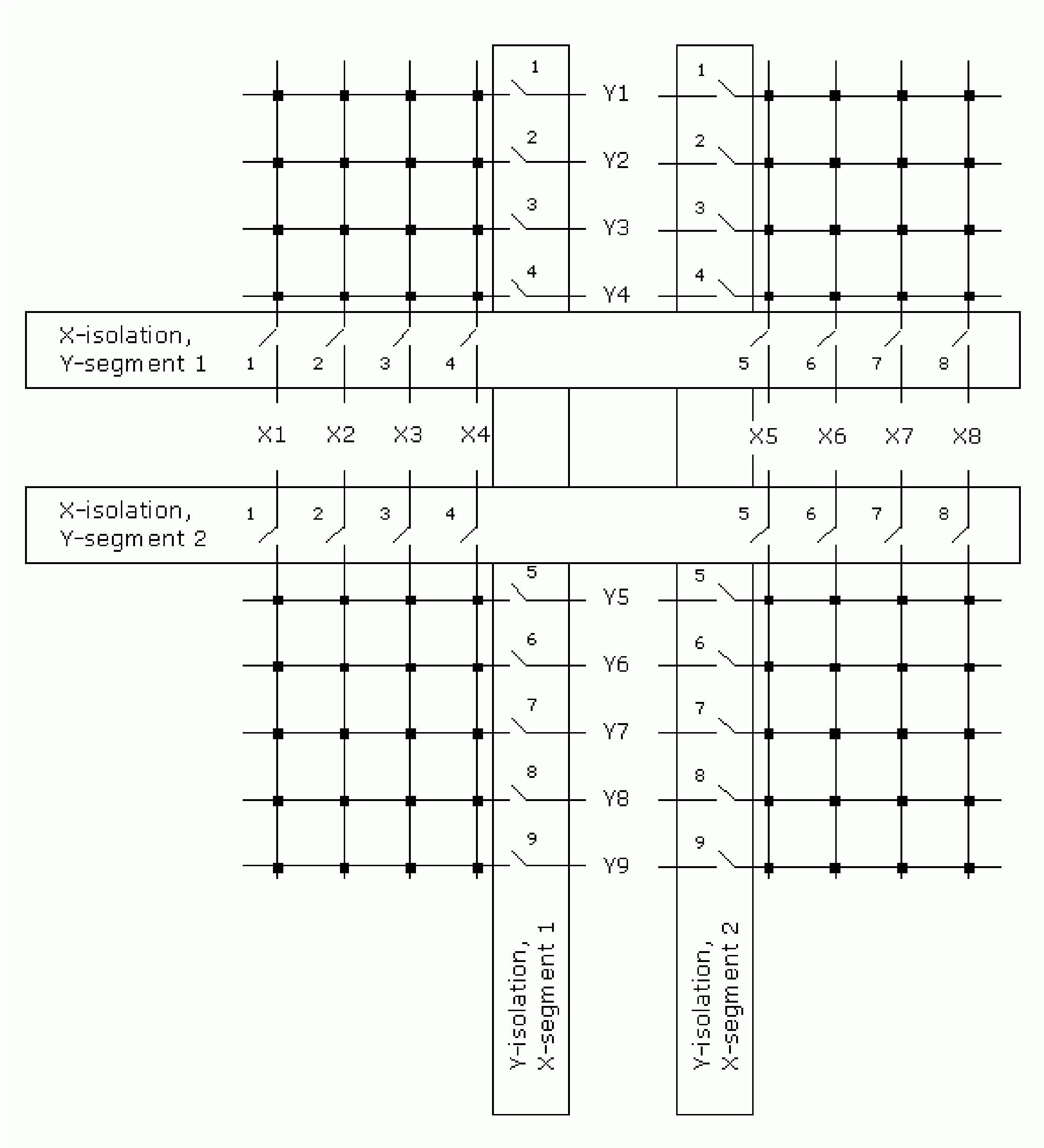
- switchFunc = pipx40_SW_FUNC_CHANNEL
- segNum = 1 thru 4



Isolation switch numbers

These switch numbers are valid for pipx40_operateSwitch when:

- switchFunc = pipx40_SW_FUNC_X_ISO or pipx40_SW_FUNC_Y_ISO
- segNum = 1 or 2


Y1
Y2
Y3
Y4
X-isolation,
Y-segment 1
X1
X2
X3
X4
X5
X6
X7
X8
X-isolation,
Y-segment 2
Y5
Y6
Y7
Y8
Y9
Y-isolation,
X-segment 1
Y-isolation,
X-segment 2

Segmented Matrix 40-726-751-LT

Operation of this model's crosspoint and isolation switches by pipx40_operateSwitch is similar to that of 40-725-511, which only differs dimensionally - the size of each segment in 40-726 being 6 x 4.

In addition, this model incorporates loopthru switches on all lines of its Y-axis.

Note that an alternate logical configuration treats the card as multiple sub-units, giving independent access to all switches via the ordinary control functions: for that configuration pipx40_operateSwitch is not applicable.

Attribute values

The relevant values obtained by pipx40_getSubAttribute when configured for auto-isolation and auto-loopthru are:

Attribute code	**Attribute value**
pipx40_SUB_ATTR_CHANNEL_SUBSWITCHES	1
pipx40_SUB_ATTR_X_ISO_SUBSWITCHES	1
pipx40_SUB_ATTR_Y_ISO_SUBSWITCHES	1
pipx40_SUB_ATTR_X_LOOPTHRU_SUBSWITCHES	0
pipx40_SUB_ATTR_Y_LOOPTHRU_SUBSWITCHES	1
pipx40_SUB_ATTR_NUM_X_SEGMENTS	2
pipx40_SUB_ATTR_X_SEGMENT01_SIZE	6
pipx40_SUB_ATTR_X_SEGMENT02_SIZE	6
pipx40_SUB_ATTR_NUM_Y_SEGMENTS	2
pipx40_SUB_ATTR_Y_SEGMENT01_SIZE	4
pipx40_SUB_ATTR_Y_SEGMENT02_SIZE	4

Segmented Matrix 40-560-021

This documents a specimen 40-560-021 BRIC configuration, as a 50 x 8 matrix using two 46 x 8 daughtercards; the second daughtercard being partially populated as 4 x 8. This design is segmented only on the X-axis (each daughtercard having Y-isolation switches only).

In its standard configuration as a single 50 x 8 matrix sub-unit, when channel selections are made using functions such as:

- pipx40_setChannelState
- pipx40_setChannelPattern
- pipx40_setCrosspointState

operation of isolation switches is automated to optimise connections for X - Y signal routing. pipx40_operateSwitch allows access to individual switches for other routing schemes or fault diagnostic purposes.

Note that an alternate logical configuration is possible, the unit being treated as multiple sub-units and giving independent access to all switches via the ordinary control functions: for that configuration pipx40_operateSwitch would not be applicable.

In a unit employing a larger number of daughtercards, the number of X-segments is correspondingly increased.

Attribute values

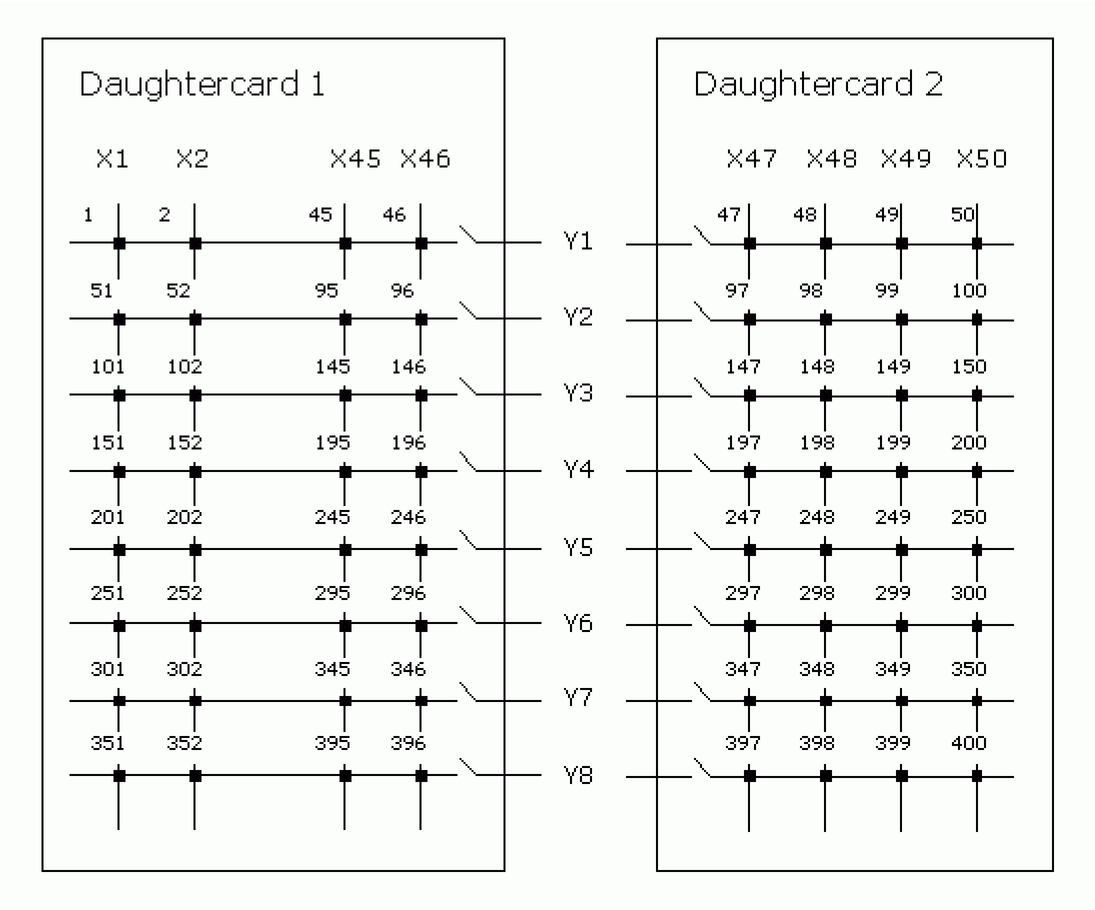
The relevant values obtained by pipx40_getSubAttribute when configured for auto-isolation are:

Attribute code	Attribute value
pipx40_SUB_ATTR_CHANNEL_SUBSWITCHES	1
pipx40_SUB_ATTR_X_ISO_SUBSWITCHES	0
pipx40_SUB_ATTR_Y_ISO_SUBSWITCHES	1
pipx40_SUB_ATTR_NUM_X_SEGMENTS	2
pipx40_SUB_ATTR_X_SEGMENT01_SIZE	46
pipx40_SUB_ATTR_X_SEGMENT02_SIZE	4
pipx40_SUB_ATTR_NUM_Y_SEGMENTS	1
pipx40_SUB_ATTR_Y_SEGMENT01_SIZE	8

Global crosspoint switch numbers

These numbers correspond to the channel numbers used with pipx40_setChannelState and are valid for pipx40_operateSwitch when:

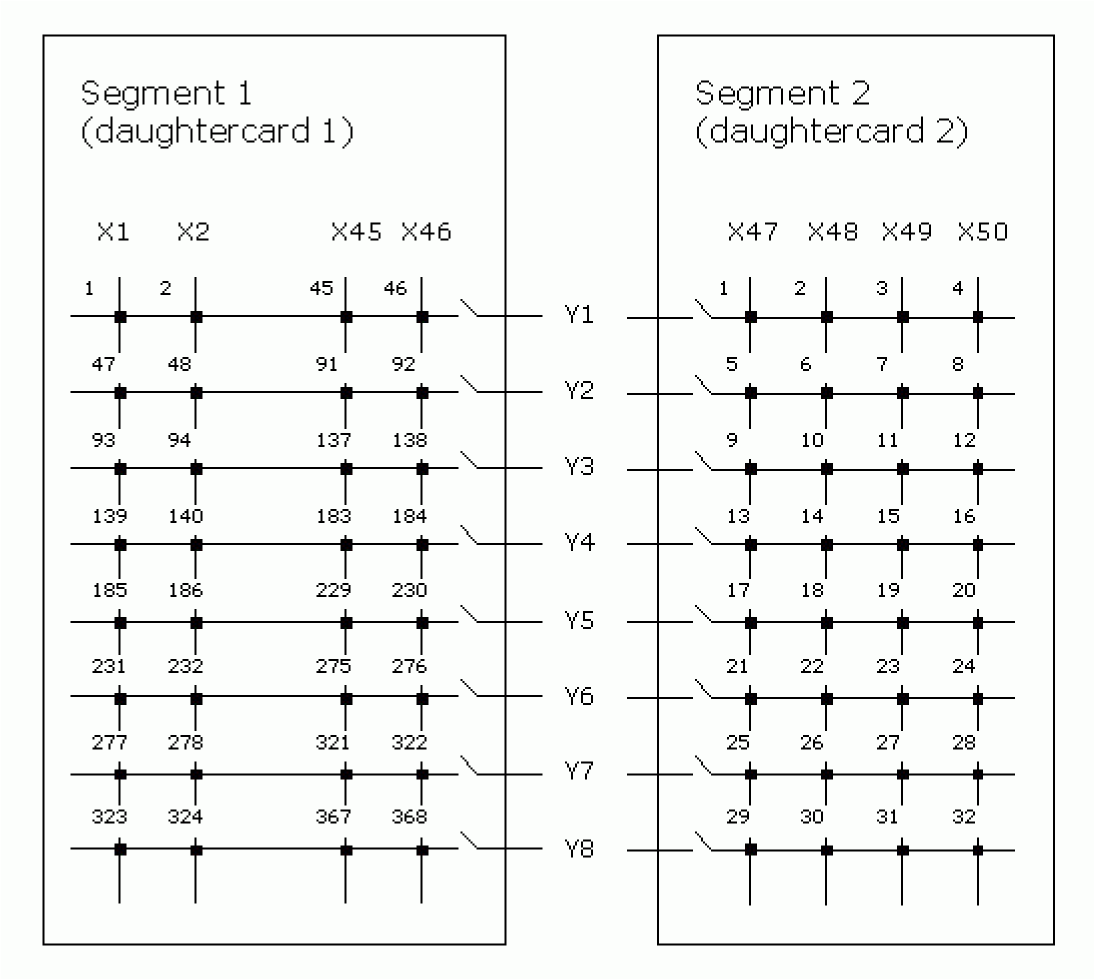
- switchFunc = pipx40_SW_FUNC_CHANNEL
- segNum = 0



Segment-local crosspoint switch numbers

These switch numbers are valid for pipx40_operateSwitch when:

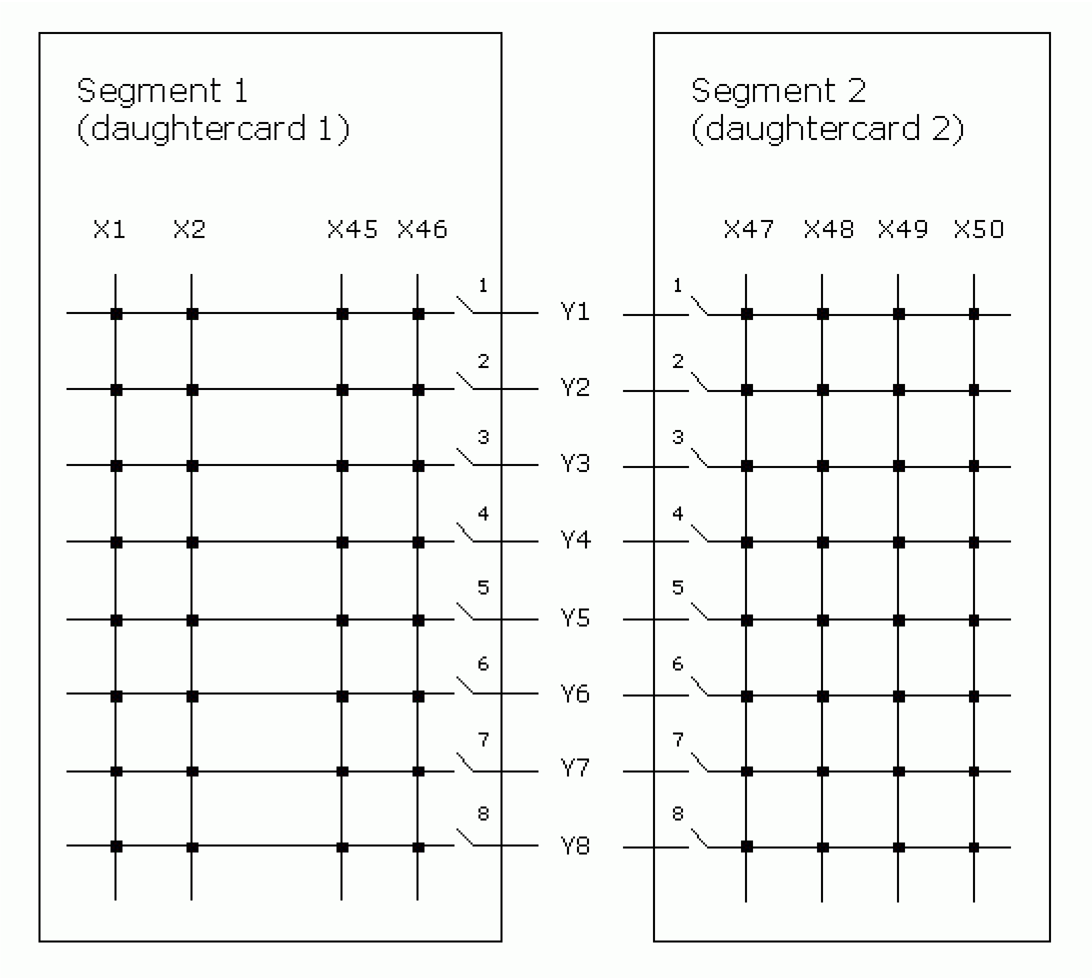
- switchFunc = pipx40_SW_FUNC_CHANNEL
- segNum = 1 or 2



Isolation switch numbers

These switch numbers are valid for pipx40_operateSwitch when:

- switchFunc = pipx40_SW_FUNC_Y_ISO
- segNum = 1 or 2


Segment 1
(daughtercard 1)
X1
X2
X45
X46
Segment 2
(daughtercard 2)
X47
X48
X49
X50
Y1
Y2
Y3
Y4
Y5
Y6
Y7
Y8
1
2
3
4
5
6
7
8

Unsegmented Matrix

An unsegmented matrix is one in which all lines on an axis are served by a single set of isolation switches on the opposing axis.

Examples:

- there is currently no real example of this configuration

Index

R

S

T

Pickering Interfaces PXI Direct I/O Driver - Pilpxi

Table Of Contents

Pickering Interfaces PXI Direct I/O Driver - Pilpxi

VXI IEEE-488 RS-232 PXI Programmable Switching Systems
www.pickeringtest.com

Pickering Interfaces PXI Direct I/O Driver - Pilpxi

This document describes programming support and diagnostic utilities for Pickering Interfaces PXI cards using the Pilpxi Direct I/O (kernel) driver, which is applicable to the following families of switching cards:

- System 40 (3U PXI)
- System 45 (6U PXI)
- System 50 (PCI)

Certain System 41 (PXI Instrument) cards are also supported - for models see the System 41 Support List.

System 40/45/50 cards offer a wide range of Relay Switching and Digital Input-Output functions in PXI, CompactPCI and PCI formats.

Version date: 08 Oct 2009

Pilpxi driver version: 3.10

Pilpxi Direct I/O Driver Basics

The Pilpxi Direct I/O driver is a 'kernel' driver, and works independently of indirected I/O schemes such as VISA (Virtual Instrument Software Architecture). The driver is implemented in Dynamic Link Library Pilpxi.dll, together with library/header files for each supported programming environment.

Alternative drivers

If a VISA-based solution is preferred the **pipx40** driver is available, offering broadly similar functionality to Pilpxi.

A driver compliant with the IVI (Interchangeable Virtual Instruments) standard, **pi40iv**, is also available.

Accessing Cards

Opening Cards

The Pilpxi driver supports two mechanisms for opening and closing Pickering cards - see function reference for Visual Basic / Visual C++.

Card Numbers

When opened by PIL_OpenCards, each Pickering card is accessed using a logical card number, starting from 1. Note that the logical number associated with any card may change if the number of installed switch cards is changed, or if cards are moved to different slot positions. Function PIL_CardLoc can be used to obtain the logical bus/slot location associated with a logical card number, and PIL_CardId to discover the card's identity.

When opened by PIL_OpenSpecifiedCard, the logical card number associated with a card is the value returned in the CardNum argument of the PIL_OpenSpecifiedCard call that opened it. PIL_CardId obtains the card's identity.

Sub-units

All Pickering cards contain one or more independently addressable functional blocks, or sub-units. Sub-unit numbers begin at 1, and separate sequences are used for input and output functions. This number is used in function calls to access the appropriate block. Generally, sub-unit numbers correspond directly to the bank numbers specified in hardware documentation.

Sub-unit examples:

Model	**Configuration**	**INPUT sub-unit #1**	**OUTPUT sub-unit #1**	**OUTPUT sub-unit #2**	**OUTPUT sub-unit #3**
40-110-021	16 SPDT switches	None	16 SPDT switches	None	None
40-290-121	Dual Programmable resistors + 16 SPDT switches	None	Resistor #1	Resistor #2	16 SPDT switches
40-490-001	Digital I/O	16-channel inputs	32-channel outputs	None	None
40-511-021	Dual 12 x 4 matrix	None	12 x 4 matrix #1	12 x 4 matrix #2	None

Sub-unit characteristics

The numbers of input and output sub-units in a card can be obtained using function PIL_EnumerateSubs.

Sub-unit type and dimensions can be obtained using functions:

PIL_SubType - as a text string

PIL_SubInfo - in numerical format

PIL_SubType type desc.	**PIL_SubInfo type value**	**Characteristics**
INPUT	1	Digital inputs.
SWITCH	1 - TYPE_SW	Uncommitted switches. Switches can be selected in any arbitrary pattern.
MUX	2 - TYPE_MUX	Multiplexer, single channel. Only one channel can be selected at any time.
MUXM	3 - TYPE_MUXM	Multiplexer, multi channel. Any number of channels can be selected simultaneously.
MATRIX	4 - TYPE_MAT	Matrix, LF. Multiple crosspoints may be closed on any row or column, though there may be a limit on the total number that can be closed simultaneously. Some matrices intended for RF use are also characterised as this type, though closure of multiple crosspoints on a row or column will inevitably compromise RF performance.
MATRIXR	5 - TYPE_MATR	Matrix, RF. A matrix intended for RF use, generally permitting the closure of only one crosspoint on each row and column.
DIGITAL	6 - TYPE_DIG	Digital outputs. Outputs can be energised in any arbitrary pattern.
RES	7 - TYPE_RES	Programmable resistor.
ATTEN	8 - TYPE_ATTEN	Programmable RF attenuator.
PSUDC	9 - TYPE_PSUDC	DC power supply.
BATT	10 - TYPE_BATT	Battery Simulator.

Data Formats

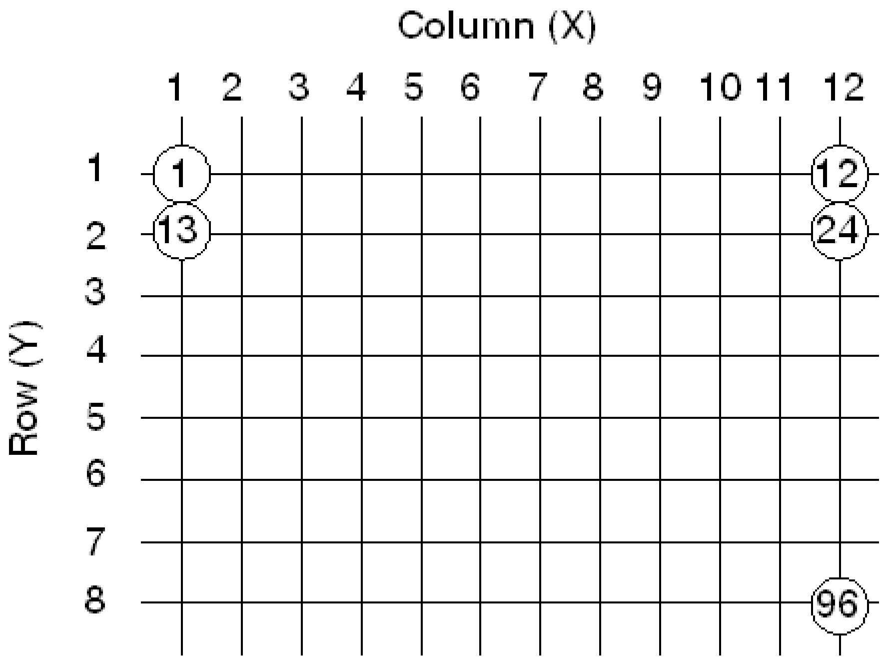
Two basic data formats are used by the driver.

Bit Number

The individual output to be affected by functions such as PIL_OpBit is specified by a bit number value.

For any sub-unit type other than a matrix, this **unity-based** number directly specifies the affected output channel.

For a matrix sub-unit, the bit number of a crosspoint is determined by folding on the row-axis. For example in a MATRIX(12X8), having 12 columns and 8 rows, bit number 13 represents the crosspoint (row 2, column 1):



Note: matrix operation

More straightforward matrix operation using row/column co-ordinates is provided by functions:

PIL_OpCrosspoint

PIL_ViewCrosspoint

PIL_MaskCrosspoint

PIL_ViewMaskCrosspoint

Data Array

Functions affecting all of a sub-unit's channels utilise a one-dimensional data array (or vector) of 32-bit (unsigned) longwords. In the array, each bit represents the state of one output channel: '0' for OFF, '1' for ON. The least significant bit in the base element of the array corresponds to channel 1, with more significant bits corresponding to higher-numbered channels.

The minimum number of longwords needed to represent a sub-unit is the integer part of:

((rows * columns) + 31) / 32

For a matrix sub-unit, bit assignments follow the same pattern as that used to determine bit numbers. Hence for the matrix example above:

Element 0 bit 0 = row 1 column 1

Element 0 bit 11 = row 1 column 12

Element 0 bit 12 = row 2 column 1

Element 2 bit 31 = row 8 column 12

This format is employed by functions:

PIL_WriteSub

PIL_ViewSub

PIL_WriteSubArray

PIL_ViewSubArray

PIL_WriteMask

PIL_ViewMask

PIL_WriteMaskArray

PIL_ViewMaskArray

PIL_ReadSub

Timing Issues

Default mode

In the default mode of operation, driver functions incorporate appropriate delay periods to guarantee safe sequencing of internal events and that switch states will have stabilised prior to returning (fully debounced operation).

Break-before-make action is enforced for all operations, including pattern based functions such as PIL_WriteSub.

No-wait mode

If the option MODE_NO_WAIT is invoked using PIL_SetMode all sequencing and settling delays are disabled. This allows other operations to proceed while switches are transitioning - the debounce period for a microwave or high power switch may be 15 milliseconds or more. A sub-unit's debounce period can be discovered using PIL_SettleTime.

It should be borne in mind that for some models the elimination of internal sequencing delays could result in transient illicit states.

When MODE_NO_WAIT is set stabilisation of a sub-unit's switches can be determined by polling the result of PIL_SubStatus; or stabilisation of all switches on a card by polling with PIL_Status. In either case stabilisation is indicated by the STAT_BUSY bit being clear.

Error Codes

Many of the Pilpxi.dll functions return a numeric error code that indicates success or failure of the function call.

A string describing an error code can be obtained using PIL_ErrorMessage - see function reference for Visual Basic / Visual C++.

Error codes are as follows:

0	NO_ERR	Success
1	ER_NO_CARD	No card present with specified number
2	ER_NO_INFO	Card information unobtainable - hardware problem
3	ER_CARD_DISABLED	Card disabled - hardware problem
4	ER_BAD_SUB	Card has no sub-unit with specified number
5	ER_BAD_BIT	Sub-unit has no bit with specified number
6	ER_NO_CAL_DATA	Sub-unit has no calibration data to write/read
7	ER_BAD_ARRAY	Array type, size or shape is incorrect
8	ER_MUX_ILLEGAL	Non-zero write data is illegal for MUX sub-unit
9	ER_EXCESS_CLOSURE	Sub-unit closure limit exceeded
10	ER_ILLEGAL_MASK	One or more of the specified channels cannot be masked
11	ER_OUTPUT_MASKED	Cannot activate an output that is masked
12	ER_BAD_LOCATION	Cannot open a Pickering card at the specified location
13	ER_READ_FAIL	Failed read from hardware
14	ER_WRITE_FAIL	Failed write to hardware
15	ER_DRIVER_OP	Hardware driver failure
16	ER_DRIVER_VERSION	Incompatible hardware driver version
17	ER_SUB_TYPE	Function call incompatible with sub-unit type or capabilities
18	ER_BAD_ROW	Matrix row value out of range
19	ER_BAD_COLUMN	Matrix column value out of range
20	ER_BAD_ATTEN	Attenuation value out of range
21	ER_BAD_VOLTAGE	Voltage value out of range
22	ER_BAD_CAL_INDEX	Calibration reference out of range
23	ER_BAD_SEGMENT	Segment number out of range

24	ER_BAD_FUNC_CODE	Function code value out of range
25	ER_BAD_SUBSWITCH	Subswitch value out of range
26	ER_BAD_ACTION	Action code out of range
27	ER_STATE_CORRUPT	Cannot execute due to corrupt sub-unit state
28	ER_BAD_ATTR_CODE	Unrecognised attribute code
29	ER_EEPROM_WRITE_TMO	Timeout writing to EEPROM
30	ER_ILLEGAL_OP	Operation is illegal in the sub-unit's current state
31	ER_BAD_POT	Unrecognised pot number requested
32	ER_MATRIXR_ILLEGAL	Invalid write pattern for MATRIXR sub-unit
33	ER_MISSING_CHANNEL	Attempted operation on non-existent channel
34	ER_CARD_INACCESSIBLE	Card cannot be accessed (failed/removed/unpowered)
35	ER_BAD_FP_FORMAT	Unsupported internal floating-point format (internal error)
36	ER_UNCALIBRATED	Sub-unit is not calibrated
37	ER_BAD_RESISTANCE	Unobtainable resistance value
38	ER_BAD_STORE	Invalid calibration store number
39	ER_BAD_MODE	Invalid mode value
40	ER_SETTINGS_CONFLICT	Conflicting device settings
41	ER_CARD_TYPE	Function call incompatible with card type or capabilities
42	ER_BAD_POLE	Switch pole value out of range
43	ER_MISSING_CAPABILITY	Attempted to activate a non-existent capability
44	ER_MISSING_HARDWARE	Action requires hardware that is not present
45	ER_HARDWARE_FAULT	Faulty hardware
46	ER_EXECUTION_FAIL	Failed to execute (e.g. blocked by a hardware condition)
47	ER_BAD_CURRENT	Current value out of range

For Visual Basic, corresponding global constants are provided in Pilpxi.bas.

For Visual C++, corresponding enumerated constants are provided in Pilpxi.h.

Contact Pickering

For further assistance, please contact:

Pickering Interfaces Ltd.
Stephenson Road
Clacton-on-Sea

Essex CO15 4NL
UK

Telephone: 44 (0)1255 687900

Fax: 44 (0)1255 425349

WWW: http://www.pickeringtest.com

Email (sales): sales@pickeringtest.com

Email (technical support): support@pickeringtest.com

System 41 Support List

The following System 41 models are supported by Pilpxi driver version 3.10:

- 41-180-021
- 41-180-022
- 41-181-021
- 41-181-022
- 41-660-001
- 41-661-001
- 41-720
- 41-735-001
- 41-750-001
- 41-751-001
- 41-752-001

If your System 41 card does not appear in this list support for it may have been added subsequent to the above release; or it may be supported instead by its own card-specific driver. In either case the appropriate driver version can be downloaded from our website http://www.pickeringtest.com.

Cards with Special Features

Cards with Special Features

Certain cards support special features that are accessed using Input, General Purpose Output or other specific functions. The nature of these features and their methods of operation by the software driver are model-specific:

- 40-170-101, 40-170-102 Current Sensing Switch Cards
- 40-260-001 Precision Resistor
- 40-261 Precision Resistor
- 40-262 RTD Simulator
- 40-265 Strain Gauge Simulator
- 40-297 Precision Resistor
- 40-412-001 Digital Input-Output
- 40-412-101 Digital Input-Output
- 40-413-001 Digital Input-Output
- 40-413-002 Digital Input-Output
- 40-413-003 Digital Input-Output
- 41-750-001 Battery Simulator
- 41-751-001 Battery Simulator
- 41-752-001 Battery Simulator
- 50-297 Precision Resistor

40-170-101/102 Current Sensing Switch Card

The 40-170-101 and 40-170-102 cards contain current sensing circuitry to monitor the current flowing through the main relay contacts. A voltage proportional to the current flowing through the contacts is delivered to the monitor output on the card.

The card contains the following sub-units:

Output Sub-Units	**Function**
1	2 bit switch, 1 for each relay
2	2-way MUX, controls monitor of relay 1 or relay 2 or cascade if neither relay is selected
3 *	16-bit digital output, used to control current monitor circuit 1
4 *	16-bit digital output, used to control current monitor circuit 2

Input Sub-Units	**Function**
1 *	8-bit port to read result of control commands on circuit 1
2 *	8-bit port to read result of control commands on circuit 2
3 *	8-bit port to read RDAC(0) on circuit 1
4 *	8-bit port to read RDAC(1) on circuit 1
5 *	8-bit port to read RDAC(0) on circuit 2
6 *	8-bit port to read RDAC(1) on circuit 2

The sub-units marked with an asterisk (*) are used for calibration of the current monitoring circuits and are not required for normal operation, refer to the 40-170-101 User Manual for more detail.

40-260-001 Precision Resistor

The 40-260-001 Precision Resistor card contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(28)	Precision resistor 1
2: RES(28)	Precision resistor 2
3: RES(28)	Precision resistor 3

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewSub PIL_ClearSub
4: MUX(4)	Common reference multiplexer

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_SetCalPoint PIL_ReadCalFP PIL_WriteCalFP PIL_WriteCalDate	**Applicable functions** PIL_WriteSub PIL_ViewSub
1: RES(28)	Precision resistor 1	PR1 switched resistance elements
2: RES(28)	Precision resistor 2	PR2 switched resistance elements
3: RES(28)	Precision resistor 3	PR3 switched resistance elements

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewSub PIL_ClearSub

5: MUX(9)	DMM multiplexer

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub
6: DIGITAL(32)	PR1 digital pot element
7: DIGITAL(32)	PR2 digital pot element
8: DIGITAL(32)	PR3 digital pot element

Refer to the 40-260-001 User Manual for more detail.

40-261 Precision Resistor

The 40-261-001 and 40-261-002 Precision Resistor cards contain an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Units	Applicable functions PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(38)	Precision resistor 1
2: RES(38)	Precision resistor 2

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	Applicable functions PIL_SetCalPoint PIL_ReadCalFP PIL_WriteCalFP PIL_WriteCalDate	Applicable functions PIL_WriteSub PIL_ViewSub
1: RES(38)	Precision resistor 1	PR1 switched resistance elements
2: RES(38)	Precision resistor 2	PR2 switched resistance elements

Output Sub-Unit	Applicable functions PIL_OpBit PIL_ViewSub PIL_ClearSub
3: MUX(6)	DMM multiplexer

Refer to the 40-261 User Manual for more detail.

40-262 RTD Simulator

Model 40-262 RTD Simulator cards contain an array of sub-units for control and calibration.

Models 40-262-001, 40-262-002 (18 channels): functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(13)	Simulator channel 1
2: RES(13)	Simulator channel 2
3: RES(13)	Simulator channel 3
4: RES(13)	Simulator channel 4
5: RES(13)	Simulator channel 5
6: RES(13)	Simulator channel 6
7: RES(13)	Simulator channel 7
8: RES(13)	Simulator channel 8
9: RES(13)	Simulator channel 9
10: RES(13)	Simulator channel 10
11: RES(13)	Simulator channel 11
12: RES(13)	Simulator channel 12
13: RES(13)	Simulator channel 13
14: RES(13)	Simulator channel 14
15: RES(13)	Simulator channel 15
16: RES(13)	Simulator channel 16
17: RES(13)	Simulator channel 17
18: RES(13)	Simulator channel 18

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewSub PIL_ClearSub
19: MUX(4)	Common reference multiplexer

Models 40-262-001, 40-262-002 (18 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output	**Applicable**	**Applicable functions**

Sub-Units	**functions** PIL_SetCalPoint PIL_ReadCalFP PIL_WriteCalFP PIL_WriteCalDate	PIL_WriteSub PIL_ViewSub
1: RES(13)	Simulator channel 1	Sim chan 1 switched resistance elements
2: RES(13)	Simulator channel 2	Sim chan 2 switched resistance elements
3: RES(13)	Simulator channel 3	Sim chan 3 switched resistance elements
4: RES(13)	Simulator channel 4	Sim chan 4 switched resistance elements
5: RES(13)	Simulator channel 5	Sim chan 5 switched resistance elements
6: RES(13)	Simulator channel 6	Sim chan 6 switched resistance elements
7: RES(13)	Simulator channel 7	Sim chan 7 switched resistance elements
8: RES(13)	Simulator channel 8	Sim chan 8 switched resistance elements
9: RES(13)	Simulator channel 9	Sim chan 9 switched resistance elements
10: RES(13)	Simulator channel 10	Sim chan 10 switched resistance elements
11: RES(13)	Simulator channel 11	Sim chan 11 switched resistance elements
12: RES(13)	Simulator channel 12	Sim chan 12 switched resistance elements
13: RES(13)	Simulator channel 13	Sim chan 13 switched resistance elements
14: RES(13)	Simulator channel 14	Sim chan 14 switched resistance elements
15: RES(13)	Simulator channel 15	Sim chan 15 switched resistance elements
16: RES(13)	Simulator channel 16	Sim chan 16 switched resistance elements
17: RES(13)	Simulator channel 17	Sim chan 17 switched resistance elements
18: RES(13)	Simulator channel 18	Sim chan 18 switched resistance elements

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewSub PIL_ClearSub
20: MUX(54)	DMM multiplexer

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub
21: DIGITAL(32)	Sim chan 1 digital pot element
22: DIGITAL(32)	Sim chan 2 digital pot element

23: DIGITAL(32)	Sim chan 3 digital pot element
24: DIGITAL(32)	Sim chan 4 digital pot element
25: DIGITAL(32)	Sim chan 5 digital pot element
26: DIGITAL(32)	Sim chan 6 digital pot element
27: DIGITAL(32)	Sim chan 7 digital pot element
28: DIGITAL(32)	Sim chan 8 digital pot element
29: DIGITAL(32)	Sim chan 9 digital pot element
30: DIGITAL(32)	Sim chan 10 digital pot element
31: DIGITAL(32)	Sim chan 11 digital pot element
33: DIGITAL(32)	Sim chan 12 digital pot element
33: DIGITAL(32)	Sim chan 13 digital pot element
34: DIGITAL(32)	Sim chan 14 digital pot element
35: DIGITAL(32)	Sim chan 15 digital pot element
36: DIGITAL(32)	Sim chan 16 digital pot element
37: DIGITAL(32)	Sim chan 17 digital pot element
38: DIGITAL(32)	Sim chan 18 digital pot element

Models 40-262-101, 40-262-102 (6 channels): functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(13)	Simulator channel 1
2: RES(13)	Simulator channel 2
3: RES(13)	Simulator channel 3
4: RES(13)	Simulator channel 4
5: RES(13)	Simulator channel 5
6: RES(13)	Simulator channel 6

Output Sub-Units	**Applicable functions** PIL_OpBit PIL_ViewSub PIL_ClearSub
7: MUX(4)	Common reference multiplexer

Models 40-262-101, 40-262-102 (6 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_SetCalPoint PIL_ReadCalFP	**Applicable functions** PIL_WriteSub PIL_ViewSub

	PIL_WriteCalFP PIL_WriteCalDate	
1: RES(13)	Simulator channel 1	Sim chan 1 switched resistance elements
2: RES(13)	Simulator channel 2	Sim chan 2 switched resistance elements
3: RES(13)	Simulator channel 3	Sim chan 3 switched resistance elements
4: RES(13)	Simulator channel 4	Sim chan 4 switched resistance elements
5: RES(13)	Simulator channel 5	Sim chan 5 switched resistance elements
6: RES(13)	Simulator channel 6	Sim chan 6 switched resistance elements

Output Sub-Units	**Applicable functions** PIL_OpBit PIL_ViewSub PIL_ClearSub
8: MUX(18)	DMM multiplexer

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub
9: DIGITAL(32)	Sim chan 1 digital pot element
10: DIGITAL(32)	Sim chan 2 digital pot element
11: DIGITAL(32)	Sim chan 3 digital pot element
12: DIGITAL(32)	Sim chan 4 digital pot element
13: DIGITAL(32)	Sim chan 5 digital pot element
14: DIGITAL(32)	Sim chan 6 digital pot element

Refer to the 40-262 User Manual for more detail.

40-265 Strain Gauge Simulator

Strain Gauge Simulator model 40-265-006 contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ReadCalDate
1: RES(64)	Simulator channel 1
2: RES(64)	Simulator channel 2
3: RES(64)	Simulator channel 3
4: RES(64)	Simulator channel 4
5: RES(64)	Simulator channel 5
6: RES(64)	Simulator channel 6

Output Sub-Units	**Applicable functions** PIL_OpBit PIL_WriteSub PIL_ViewBit PIL_ViewSub PIL_ClearSub
7: SWITCH(4)	Simulator channel 1 auxiliary switches
8: SWITCH(4)	Simulator channel 2 auxiliary switches
9: SWITCH(4)	Simulator channel 3 auxiliary switches
10: SWITCH(4)	Simulator channel 4 auxiliary switches
11: SWITCH(4)	Simulator channel 5 auxiliary switches
12: SWITCH(4)	Simulator channel 6 auxiliary switches

A simulator channel's null-point resistance can be obtained using function:

- PIL_ResInfo (in its RefRes argument)

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_SetCalPoint	**Applicable functions** PIL_WriteSub PIL_ViewSub

	PIL_ReadCalFP PIL_WriteCalFP PIL_WriteCalDate	
1: RES(64)	Simulator channel 1	Simulator channel 1 resistance elements
2: RES(64)	Simulator channel 2	Simulator channel 2 resistance elements
3: RES(64)	Simulator channel 3	Simulator channel 3 resistance elements
4: RES(64)	Simulator channel 4	Simulator channel 4 resistance elements
5: RES(64)	Simulator channel 5	Simulator channel 5 resistance elements
6: RES(64)	Simulator channel 6	Simulator channel 6 resistance elements

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewBit PIL_ViewSub PIL_ClearSub
13: MUX(18)	DMM multiplexer

Refer to the 40-265 User Manual for more detail.

40-297 Precision Resistor

40-297 Precision Resistor cards contain an array of sub-units for control and calibration.

Model 40-297-001 (18 channels): functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(10)	Precision resistor 1
2: RES(10)	Precision resistor 2
3: RES(10)	Precision resistor 3
4: RES(10)	Precision resistor 4
5: RES(10)	Precision resistor 5
6: RES(10)	Precision resistor 6
7: RES(10)	Precision resistor 7
8: RES(10)	Precision resistor 8
9: RES(10)	Precision resistor 9
10: RES(10)	Precision resistor 10
11: RES(10)	Precision resistor 11
12: RES(10)	Precision resistor 12
13: RES(10)	Precision resistor 13
14: RES(10)	Precision resistor 14
15: RES(10)	Precision resistor 15
16: RES(10)	Precision resistor 16
17: RES(10)	Precision resistor 17
18: RES(10)	Precision resistor 18

Model 40-297-001 (18 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_SetCalPoint PIL_ReadCalFP PIL_WriteCalFP PIL_WriteCalDate	**Applicable functions** PIL_WriteSub PIL_ViewSub
1: RES(10)	Precision resistor 1	PR1 switched resistance elements
2: RES(10)	Precision resistor 2	PR2 switched resistance elements

3: RES(10)	Precision resistor 3	PR3 switched resistance elements
4: RES(10)	Precision resistor 4	PR4 switched resistance elements
5: RES(10)	Precision resistor 5	PR5 switched resistance elements
6: RES(10)	Precision resistor 6	PR6 switched resistance elements
7: RES(10)	Precision resistor 7	PR7 switched resistance elements
8: RES(10)	Precision resistor 8	PR8 switched resistance elements
9: RES(10)	Precision resistor 9	PR9 switched resistance elements
10: RES(10)	Precision resistor 10	PR10 switched resistance elements
11: RES(10)	Precision resistor 11	PR11 switched resistance elements
12: RES(10)	Precision resistor 12	PR12 switched resistance elements
13: RES(10)	Precision resistor 13	PR13 switched resistance elements
14: RES(10)	Precision resistor 14	PR14 switched resistance elements
15: RES(10)	Precision resistor 15	PR15 switched resistance elements
16: RES(10)	Precision resistor 16	PR16 switched resistance elements
17: RES(10)	Precision resistor 17	PR17 switched resistance elements
18: RES(10)	Precision resistor 18	PR18 switched resistance elements

Model 40-297-002 (9 channels): functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(19)	Precision resistor 1
2: RES(19)	Precision resistor 2
3: RES(19)	Precision resistor 3
4: RES(19)	Precision resistor 4
5: RES(19)	Precision resistor 5
6: RES(19)	Precision resistor 6
7: RES(19)	Precision resistor 7
8: RES(19)	Precision resistor 8
9: RES(19)	Precision resistor 9

Model 40-297-002 (9 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_SetCalPoint PIL_ReadCalFP PIL_WriteCalFP PIL_WriteCalDate	**Applicable functions** PIL_WriteSub PIL_ViewSub
1: RES(19)	Precision resistor 1	PR1 switched resistance elements
2: RES(19)	Precision resistor 2	PR2 switched resistance elements
3: RES(19)	Precision resistor 3	PR3 switched resistance elements
4: RES(19)	Precision resistor 4	PR4 switched resistance elements
5: RES(19)	Precision resistor 5	PR5 switched resistance elements
6: RES(19)	Precision resistor 6	PR6 switched resistance elements
7: RES(19)	Precision resistor 7	PR7 switched resistance elements
8: RES(19)	Precision resistor 8	PR8 switched resistance elements
9: RES(19)	Precision resistor 9	PR9 switched resistance elements

Model 40-297-003 (6 channels): functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(28)	Precision resistor 1
2: RES(28)	Precision resistor 2
3: RES(28)	Precision resistor 3
4: RES(28)	Precision resistor 4
5: RES(28)	Precision resistor 5
6: RES(28)	Precision resistor 6

Model 40-297-003 (6 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_SetCalPoint PIL_ReadCalFP	**Applicable functions** PIL_WriteSub PIL_ViewSub

	PIL_WriteCalFP PIL_WriteCalDate	
1: RES(28)	Precision resistor 1	PR1 switched resistance elements
2: RES(28)	Precision resistor 2	PR2 switched resistance elements
3: RES(28)	Precision resistor 3	PR3 switched resistance elements
4: RES(28)	Precision resistor 4	PR4 switched resistance elements
5: RES(28)	Precision resistor 5	PR5 switched resistance elements
6: RES(28)	Precision resistor 6	PR6 switched resistance elements

Refer to the 40-297 User Manual for more detail.

40-412-001 Digital Input-Output

The 40-412-001 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Units	**Applicable functions** PIL_OpBit PIL_ViewBit PIL_WriteSub PIL_ViewSub PIL_ClearSub PIL_MaskBit PIL_ViewMaskBit PIL_WriteMask PIL_ViewMask PIL_ClearMask
1: DIGITAL(32)	Controls output SINK driver states, each bit: 0 = INACTIVE 1 = ACTIVE
2: DIGITAL(32)	Controls output SOURCE driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub PIL_ClearSub
3: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
4: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewBit PIL_ViewSub PIL_ClearSub
5: MUX(32)	Input channel selector

Input Sub-Units	**Applicable function** PIL_ReadSub
1: INPUT(2)	Gets level of selected input channel (2 bits): 00 = below threshold 2, below threshold 1 01 = below threshold 2, above threshold 1 10 = above threshold 2, below threshold 1 11 = above threshold 2, above threshold 1
2: INPUT(64)	Gets levels of all 32 input channels (2 bits each, as above).

	NOTE: each input channel from 1 to 32 is sampled sequentially. The precise rate of sampling is undefined.

Refer to the 40-412 User Manual for more detail.

40-412-101 Digital Input-Output

The 40-412-101 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Units	Applicable functions PIL_OpBit PIL_ViewBit PIL_WriteSub PIL_ViewSub PIL_ClearSub PIL_MaskBit PIL_ViewMaskBit PIL_WriteMask PIL_ViewMask PIL_ClearMask
1: DIGITAL(32)	Controls output SINK driver states, each bit: 0 = INACTIVE 1 = ACTIVE
2: DIGITAL(32)	Controls output SOURCE driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	Applicable functions PIL_WriteSub PIL_ViewSub PIL_ClearSub
3: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
4: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Input Sub-Units	Applicable function PIL_ReadSub
1: INPUT(64)	Gets levels of all 32 input channels, relative to the set thresholds. All input channels are sampled synchronously.

Refer to the 40-412 User Manual for more detail.

40-413-001 Digital Input-Output

The 40-413-001 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewBit PIL_WriteSub PIL_ViewSub PIL_ClearSub PIL_MaskBit PIL_ViewMaskBit PIL_WriteMask PIL_ViewMask PIL_ClearMask
1: DIGITAL(32)	Controls output (SOURCE) driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub PIL_ClearSub
2: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
3: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewBit PIL_ViewSub PIL_ClearSub
4: MUX(32)	Input channel selector

Input Sub-Units	**Applicable function** PIL_ReadSub
1: INPUT(2)	Gets level of selected input channel (2 bits): 00 = below threshold 2, below threshold 1 01 = below threshold 2, above threshold 1 10 = above threshold 2, below threshold 1 11 = above threshold 2, above threshold 1
2: INPUT(64)	Gets levels of all 32 input channels (2 bits each, as above). **NOTE: each input channel from 1 to 32 is sampled sequentially. The precise rate of sampling is**

	undefined.

Refer to the 40-413 User Manual for more detail.

40-413-002 Digital Input-Output

The 40-413-002 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Unit	Applicable functions PIL_OpBit PIL_ViewBit PIL_WriteSub PIL_ViewSub PIL_ClearSub PIL_MaskBit PIL_ViewMaskBit PIL_WriteMask PIL_ViewMask PIL_ClearMask
1: DIGITAL(32)	Controls output (SINK) driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	Applicable functions PIL_WriteSub PIL_ViewSub PIL_ClearSub
2: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
3: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Output Sub-Unit	Applicable functions PIL_OpBit PIL_ViewBit PIL_ViewSub PIL_ClearSub
4: MUX(32)	Input channel selector

Input Sub-Units	Applicable function PIL_ReadSub
1: INPUT(2)	Gets level of selected input channel (2 bits): 00 = below threshold 2, below threshold 1 01 = below threshold 2, above threshold 1 10 = above threshold 2, below threshold 1 11 = above threshold 2, above threshold 1
2: INPUT(64)	Gets levels of all 32 input channels (2 bits each, as above). **NOTE: each input channel from 1 to 32 is sampled sequentially. The precise rate of sampling is undefined.**

Refer to the 40-413 User Manual for more detail.

40-413-003 Digital Input-Output

The 40-413-003 Digital Input-Output card contains an array of sub-units for its operation:

Output Sub-Units	**Applicable functions** PIL_OpBit PIL_ViewBit PIL_WriteSub PIL_ViewSub PIL_ClearSub PIL_MaskBit PIL_ViewMaskBit PIL_WriteMask PIL_ViewMask PIL_ClearMask
1: DIGITAL(32)	Controls output SINK driver states, each bit: 0 = INACTIVE 1 = ACTIVE
2: DIGITAL(32)	Controls output SOURCE driver states, each bit: 0 = INACTIVE 1 = ACTIVE

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub PIL_ClearSub
3: DIGITAL(12)	Set input threshold 1 (12-bit binary value)
4: DIGITAL(12)	Set input threshold 2 (12-bit binary value)

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewBit PIL_ViewSub PIL_ClearSub
5: MUX(32)	Input channel selector

Input Sub-Units	**Applicable function** PIL_ReadSub
1: INPUT(2)	Gets level of selected input channel (2 bits): 00 = below threshold 2, below threshold 1 01 = below threshold 2, above threshold 1 10 = above threshold 2, below threshold 1 11 = above threshold 2, above threshold 1
2: INPUT(64)	Gets levels of all 32 input channels (2 bits each, as above).

	NOTE: each input channel from 1 to 32 is sampled sequentially. The precise rate of sampling is undefined.

Refer to the 40-413 User Manual for more detail.

41-750-001 Battery Simulator

The 41-750-001 Battery Simulator card contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Unit	Applicable functions PIL_OpBit PIL_ViewBit PIL_ViewSub PIL_ClearSub
1: MUX(4)	PIMS multiplexer

Output Sub-Units	Applicable functions PIL_WriteSub PIL_ViewSub PIL_ClearSub
2: DIGITAL(96)	Current-sink setting
3: DIGITAL(16)	Voltage output DAC setting

Output Sub-Unit	Applicable functions PIL_OpBit PIL_ViewBit PIL_WriteSub PIL_ViewSub PIL_ClearSub
8: DIGITAL(1)	Output on/off control

Input Sub-Unit	Applicable functions PIL_ReadBit PIL_ReadSub
1: INPUT(1)	Read the Reg Limit Shutdown PXI Monitor signal

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub
4: DIGITAL(8)	Write RDAC1 register (pot #1 volatile setting)
5: DIGITAL(8)	Write RDAC3 register (pot #3 volatile setting)
6: DIGITAL(8)	Write EEMEM1 register (pot #1 non-volatile setting)
7: DIGITAL(8)	Write EEMEM3 register (pot #3 non-volatile setting)

Input Sub-Units	**Applicable function** PIL_ReadSub
2: INPUT(8)	Read RDAC1 register (pot #1 volatile setting)
3: INPUT(8)	Read RDAC3 register (pot #3 volatile setting)

Refer to the 41-750-001 User Manual for more detail.

41-751-001 Battery Simulator

The 41-751-001 Battery Simulator card contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewBit PIL_ViewSub PIL_ClearSub
1: MUX(4)	PIMS multiplexer

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub PIL_ClearSub
2: DIGITAL(48)	Current-sink setting
3: DIGITAL(16)	Voltage output DAC setting

Output Sub-Unit	**Applicable functions** PIL_OpBit PIL_ViewBit PIL_WriteSub PIL_ViewSub PIL_ClearSub
8: DIGITAL(1)	Output on/off control

Input Sub-Unit	**Applicable functions** PIL_ReadBit PIL_ReadSub
1: INPUT(2)	Read status signals RLSPM, CDPM

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub
4: DIGITAL(8)	Write RDAC2 register (pot #2 volatile setting)
5: DIGITAL(8)	Write RDAC3 register (pot #3 volatile setting)
6: DIGITAL(8)	Write EEMEM2 register (pot #2 non-volatile setting)
7: DIGITAL(8)	Write EEMEM3 register (pot #3 non-volatile setting)
9: DIGITAL(8)	Write RDAC1 register (pot #1 volatile setting)
10: DIGITAL(8)	Write EEMEM1 register (pot #1 non-volatile setting)

Input Sub-Units	**Applicable function** PIL_ReadSub
2: INPUT(8)	Read RDAC2 register (pot #2 volatile setting)
3: INPUT(8)	Read RDAC3 register (pot #3 volatile setting)
4: INPUT(8)	Read RDAC1 register (pot #1 volatile setting)

Refer to the 41-751-001 User Manual for more detail.

41-752-001 Battery Simulator

The 41-752-001 Battery Simulator card contains an array of sub-units for control and calibration.

Functions for normal operation

Output Sub-Units	Applicable functions PIL_BattSetVoltage PIL_BattGetVoltage PIL_BattSetCurrent PIL_BattGetCurrent PIL_BattSetEnable PIL_BattGetEnable PIL_BattReadInterlockState
1: BATT(14) 2: BATT(14) 3: BATT(14) 4: BATT(14) 5: BATT(14) 6: BATT(14)	Battery simulator channels 1 thru 6

Calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	Applicable functions PIL_WriteSub PIL_ViewSub PIL_ClearSub
1: BATT(14) 2: BATT(14) 3: BATT(14) 4: BATT(14) 5: BATT(14) 6: BATT(14)	Simulator channels 1 thru 6 voltage-setting DACs (direct binary access)

Output Sub-Units	Applicable functions PIL_WriteCal PIL_ReadCal
1: BATT(14)	Simulator channels 1

2: BATT(14) 3: BATT(14) 4: BATT(14) 5: BATT(14) 6: BATT(14)	thru 6 calibration data (14 x 16-bit values per channel)

Output Sub-Units	**Applicable functions** PIL_WriteSub PIL_ViewSub PIL_ClearSub
7: DIGITAL(16) 8: DIGITAL(16) 9: DIGITAL(16) 10: DIGITAL(16) 11: DIGITAL(16) 12: DIGITAL(16)	Simulator channels 1 thru 6 current-setting DACs (direct binary access)

Output Sub-Unit	**Applicable functions** PIL_WriteSub PIL_ViewSub PIL_OpBit PIL_ViewBit PIL_ClearSub
13: DIGITAL(6)	Simulator channels 1 thru 6 enable

Input Sub-Unit	**Applicable functions** PIL_ReadSub PIL_ReadBit
1: INPUT(1)	Global interlock state

Refer to the 41-752-001 User Manual for more detail.

50-297 Precision Resistor

50-297 Precision Resistor cards contain an array of sub-units for control and calibration.

Model 50-297-001 (18 channels): functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(10)	Precision resistor 1
2: RES(10)	Precision resistor 2
3: RES(10)	Precision resistor 3
4: RES(10)	Precision resistor 4
5: RES(10)	Precision resistor 5
6: RES(10)	Precision resistor 6
7: RES(10)	Precision resistor 7
8: RES(10)	Precision resistor 8
9: RES(10)	Precision resistor 9
10: RES(10)	Precision resistor 10
11: RES(10)	Precision resistor 11
12: RES(10)	Precision resistor 12
13: RES(10)	Precision resistor 13
14: RES(10)	Precision resistor 14
15: RES(10)	Precision resistor 15
16: RES(10)	Precision resistor 16
17: RES(10)	Precision resistor 17
18: RES(10)	Precision resistor 18

Model 50-297-001 (18 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_SetCalPoint PIL_ReadCalFP PIL_WriteCalFP PIL_WriteCalDate	**Applicable functions** PIL_WriteSub PIL_ViewSub
1: RES(10)	Precision resistor 1	PR1 switched resistance elements
2: RES(10)	Precision resistor 2	PR2 switched resistance elements

3: RES(10)	Precision resistor 3	PR3 switched resistance elements
4: RES(10)	Precision resistor 4	PR4 switched resistance elements
5: RES(10)	Precision resistor 5	PR5 switched resistance elements
6: RES(10)	Precision resistor 6	PR6 switched resistance elements
7: RES(10)	Precision resistor 7	PR7 switched resistance elements
8: RES(10)	Precision resistor 8	PR8 switched resistance elements
9: RES(10)	Precision resistor 9	PR9 switched resistance elements
10: RES(10)	Precision resistor 10	PR10 switched resistance elements
11: RES(10)	Precision resistor 11	PR11 switched resistance elements
12: RES(10)	Precision resistor 12	PR12 switched resistance elements
13: RES(10)	Precision resistor 13	PR13 switched resistance elements
14: RES(10)	Precision resistor 14	PR14 switched resistance elements
15: RES(10)	Precision resistor 15	PR15 switched resistance elements
16: RES(10)	Precision resistor 16	PR16 switched resistance elements
17: RES(10)	Precision resistor 17	PR17 switched resistance elements
18: RES(10)	Precision resistor 18	PR18 switched resistance elements

Model 50-297-002 (9 channels): functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(19)	Precision resistor 1
2: RES(19)	Precision resistor 2
3: RES(19)	Precision resistor 3
4: RES(19)	Precision resistor 4
5: RES(19)	Precision resistor 5
6: RES(19)	Precision resistor 6
7: RES(19)	Precision resistor 7
8: RES(19)	Precision resistor 8
9: RES(19)	Precision resistor 9

Model 50-297-002 (9 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_SetCalPoint PIL_ReadCalFP PIL_WriteCalFP PIL_WriteCalDate	**Applicable functions** PIL_WriteSub PIL_ViewSub
1: RES(19)	Precision resistor 1	PR1 switched resistance elements
2: RES(19)	Precision resistor 2	PR2 switched resistance elements
3: RES(19)	Precision resistor 3	PR3 switched resistance elements
4: RES(19)	Precision resistor 4	PR4 switched resistance elements
5: RES(19)	Precision resistor 5	PR5 switched resistance elements
6: RES(19)	Precision resistor 6	PR6 switched resistance elements
7: RES(19)	Precision resistor 7	PR7 switched resistance elements
8: RES(19)	Precision resistor 8	PR8 switched resistance elements
9: RES(19)	Precision resistor 9	PR9 switched resistance elements

Model 50-297-003 (6 channels): functions for normal operation

Output Sub-Units	**Applicable functions** PIL_ResInfo PIL_ResGetResistance PIL_ResSetResistance PIL_ClearSub PIL_ReadCalDate
1: RES(28)	Precision resistor 1
2: RES(28)	Precision resistor 2
3: RES(28)	Precision resistor 3
4: RES(28)	Precision resistor 4
5: RES(28)	Precision resistor 5
6: RES(28)	Precision resistor 6

Model 50-297-003 (6 channels): calibration functions

Only a calibration utility is expected to use these sub-units and functions.

Output Sub-Units	**Applicable functions** PIL_SetCalPoint PIL_ReadCalFP	**Applicable functions** PIL_WriteSub PIL_ViewSub

	PIL_WriteCalFP PIL_WriteCalDate	
1: RES(28)	Precision resistor 1	PR1 switched resistance elements
2: RES(28)	Precision resistor 2	PR2 switched resistance elements
3: RES(28)	Precision resistor 3	PR3 switched resistance elements
4: RES(28)	Precision resistor 4	PR4 switched resistance elements
5: RES(28)	Precision resistor 5	PR5 switched resistance elements
6: RES(28)	Precision resistor 6	PR6 switched resistance elements

Refer to the 50-297 User Manual for more detail.

Language Support

Language Support

The Pilpxi driver is provided with support for the following languages and programming environments:

- Microsoft Visual Basic
- Microsoft Visual C++
- Borland C++
- LabWindows/CVI
- LabVIEW

Visual Basic

Visual Basic

The following files are required for traditional Visual Basic:

- Pilpxi.bas
- Pilpxi.lib
- Pilpxi.dll

Pilpxi.bas and Pilpxi.lib must be accessible by Visual Basic at compile-time. Typically, copies of these files can be placed in the folder containing your application's source files. You should include Pilpxi.bas in your Visual Basic project.

Pilpxi.dll must be accessible by your application at run-time. Windows searches a number of standard locations for DLLs in the following order:

1. The directory containing the executable module.
2. The current directory.
3. The Windows system directory.
4. The Windows directory.
5. The directories listed in the PATH environment variable.

Placing Pilpxi.dll in one of the Windows directories has the advantage that a single copy serves any number of applications that use it, but does add to the clutter of system DLLs stored there. The Pickering Setup program places a copy of Pilpxi.dll in the Windows system directory.

Visual Basic .NET

Include file "Pilpxi.vb" is provided for Visual Basic .NET.

Visual Basic Function Tree

Initialise	
Initialise all cards	PIL_OpenCards
Initialise single card	PIL_OpenSpecifiedCard
Close	
Close all cards	PIL_CloseCards
Close single card	PIL_CloseSpecifiedCard
Card Information and Status	
Get card identification	PIL_CardId
Get card location	PIL_CardLoc
Get sub-unit closure limit	PIL_ClosureLimit
Get count of unopened cards	PIL_CountFreeCards
Get diagnostic information	PIL_Diagnostic
Get sub-unit counts	PIL_EnumerateSubs
Get description of an error	PIL_ErrorMessage
Get locations of unopened cards	PIL_FindFreeCards
Get sub-unit settling time	PIL_SettleTime
Get card status	PIL_Status
Get sub-unit information	PIL_SubInfo
Get sub-unit status	PIL_SubStatus
Get sub-unit description	PIL_SubType
Get driver version	PIL_Version
Switching and General Purpose Output	
Clear outputs of all open cards	PIL_ClearAll
Clear a single card's outputs	PIL_ClearCard
Clear a sub-unit's outputs	PIL_ClearSub
Set or clear a single output	PIL_OpBit
Get a single output's state	PIL_ViewBit
Get a sub-unit's output pattern	PIL_ViewSub
	PIL_ViewSubArray
Set a sub-unit's output pattern	PIL_WriteSub
	PIL_WriteSubArray
Specialised Switching	
Set or clear a matrix crosspoint	PIL_OpCrosspoint
Obtain/set the state of a switch	PIL_OpSwitch
Get sub-unit attribute	PIL_SubAttribute
Get a matrix crosspoint's state	PIL_ViewCrosspoint
Switch Masking	

Clear a sub-unit's mask	PIL_ClearMask
Set or clear a single output's mask	PIL_MaskBit
Set or clear a matrix crosspoint's mask	PIL_MaskCrosspoint
Get a sub-unit's mask pattern	PIL_ViewMask
	PIL_ViewMaskArray
Get a single output's mask state	PIL_ViewMaskBit
Get a matrix crosspoint's mask state	PIL_ViewMaskCrosspoint
Set a sub-unit's mask pattern	PIL_WriteMask
	PIL_WriteMaskArray
Input	
Read single input	PIL_ReadBit
Read input sub-unit pattern	PIL_ReadSub
Calibration	
Read an integer calibration value	PIL_ReadCal
Read a sub-unit's calibration date	PIL_ReadCalDate
Read floating-point calibration value(s)	PIL_ReadCalFP
Set Calibration Point	PIL_SetCalPoint
Write an integer calibration value	PIL_WriteCal
Write a sub-unit's calibration date	PIL_WriteCalDate
Write floating-point calibration value(s)	PIL_WriteCalFP
Programmable Resistor	
Get resistance value	PIL_ResGetResistance
Get resistor information	PIL_ResInfo
Set resistance value	PIL_ResSetResistance
Programmable RF Attenuator	
Get attenuation setting	PIL_AttenGetAttenuation
Get attenuator information	PIL_AttenInfo
Get the attenuation of a pad	PIL_AttenPadValue
Set attenuation level	PIL_AttenSetAttenuation
Get attenuator description	PIL_AttenType
Power Supplies	
Enable/disable output	PIL_PsuEnable
Get output voltage setting	PIL_PsuGetVoltage
Get PSU information	PIL_PsuInfo
Set output voltage	PIL_PsuSetVoltage
Get PSU description	PIL_PsuType
Battery Simulator	
Set voltage	PIL_BattSetVoltage
Get voltage	PIL_BattGetVoltage
Set current	PIL_BattSetCurrent

Get current	PIL_BattGetCurrent
Set enable	PIL_BattSetEnable
Get enable	PIL_BattGetEnable
Read interlock state	PIL_BattReadInterlockState
Mode Control	
Set driver mode	PIL_SetMode

Visual Basic Code Sample

A small demonstration project illustrates usage of many of the driver's functions. It consists of the following files in addition to the necessary driver files:

- VBDEMO.VBP
- VBDEMO.FRM

WARNING

WHEN RUN, THIS PROGRAM ACTIVATES OUTPUTS BOTH INDIVIDUALLY AND IN COMBINATIONS. IT SHOULD NOT BE RUN UNDER ANY CONDITIONS WHERE DAMAGE COULD RESULT FROM SUCH EVENTS. FOR GREATEST SAFETY IT SHOULD BE RUN ONLY WHEN NO EXTERNAL POWER IS APPLIED TO ANY CARD.

Initialise and Close

Initialise and Close

This section details the use in Visual Basic of functions for initialising and closing cards.

The Pilpxi driver supports two mechanisms for taking control of Pickering cards. The two mechanisms are mutually exclusive - the first use of one method after loading the driver DLL disables the other.

Controlling all cards

This method allows a single application program to open and access all installed Pickering cards. Using this method the cards are first opened by calling function PIL_OpenCards. Cards can then be accessed by other driver functions as necessary.

When the application has finished using the cards it should close them by calling function PIL_CloseCards.

Controlling cards individually

This method allows application programs to open and access Pickering cards on an individual basis. Using this method a card is first opened by calling function PIL_OpenSpecifiedCard. The card can then be accessed by other driver functions as necessary.

When the application has finished using the card it should be closed by calling function PIL_CloseSpecifiedCard.

Functions PIL_CountFreeCards and PIL_FindFreeCards assist in locating cards for opening by this mechanism.

Close All Cards (Visual Basic)

Description

Closes all open Pickering cards, which must have been opened using PIL_OpenCards. This function should be called when the application program has finished using them.

Declaration

Declare Sub PIL_CloseCards Lib "Pilpxi.dll" ()

Parameters:

None.

Returns:

Nothing.

Close Specified Card (Visual Basic)

Description

Closes the specified Pickering card, which must have been opened using PIL_OpenSpecifiedCard. This function should be called when the application program has finished using the card.

Declaration

Declare Function PIL_CloseSpecifiedCard Lib "Pilpxi.dll" (ByVal CardNum As Long) As Long

Parameters:

CardNum - card number

Returns:

Zero for success, or non-zero error code.

Open All Cards (Visual Basic)

Description

Locates and opens all installed Pickering cards. Once cards have been opened, other functions may then be used to access cards numbered 1 thru the value returned.

If cards have already been opened by the calling program, they are first closed - as though by PIL_CloseCards - and then re-opened.

If cards are currently opened by some other program they cannot be accessed and the function returns zero.

Declaration

Declare Function PIL_OpenCards Lib "Pilpxi.dll" () As Long

Parameters:

None.

Returns:

The number of Pickering cards located and opened.

Note

When multiple Pickering cards are installed, the assignment of card numbers depends upon their relative physical locations in the system (or more accurately, on the order in which they are detected by the computer's operating system at boot time).

Open Specified Card (Visual Basic)

Description

Opens the specified Pickering card, clearing all of its outputs. Once a card has been opened, other driver functions may then be used to access it.

If the card is currently opened by some other program it cannot be accessed and the function returns an error.

Declaration

Declare Function PIL_OpenSpecifiedCard Lib "Pilpxi.dll" (ByVal Bus As Long, ByVal Slot As Long, ByRef CardNum As Long) As Long

Parameters:

Bus - the card's logical bus location

Slot - the card's logical slot location

CardNum - variable to receive the card's logical card number

Returns:

Zero for success, or non-zero error code.

Note

The logical Bus and Slot values corresponding to a particular card are determined by system topology; values for cards that are operable by the Pilpxi driver can be discovered using PIL_FindFreeCards.

Information and Status

Information and Status

This section details the use in Visual Basic of functions for obtaining card and sub-unit information. Most of these functions are applicable to all card or sub-unit types.

Functions are provided for obtaining:

- The software driver version number: PIL_Version
- The number of unopened cards: PIL_CountFreeCards
- The bus and slot locations of unopened cards: PIL_FindFreeCards
- A card's identification string: PIL_CardId
- A card's logical bus and slot location: PIL_CardLoc
- A card's status flags: PIL_Status
- A string describing an error from the numeric code returned by a function: PIL_ErrorMessage
- A card's diagnostic information string: PIL_Diagnostic
- The numbers of input and output sub-units on a card: PIL_EumerateSubs
- Sub-unit information (numeric format): PIL_SubInfo
- Sub-unit information (string format): PIL_SubType
- An output sub-unit's closure limit value: PIL_ClosureLimit
- An output sub-unit's settling time value: PIL_SettleTime
- A sub-unit's status flags: PIL_SubStatus

Card ID (Visual Basic)

Description

Obtains the identification string of the specified card. The string contains these elements:

<type code>,<serial number>,<revision code>.

The <revision code> value represents the hardware version of the unit - cards have no firmware on-board.

Declaration

Declare Function PIL_CardId Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal Str As String) As Long

Parameters:

CardNum - card number

Str - reference to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Note

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(Str, character_count).

Card Location (Visual Basic)

Description

Obtains the location of the specified card in terms of the logical PCI bus and slot number in which it is located.
These values can be cross-referenced to physical slot locations in a particular system.

Declaration

Declare Function PIL_CardLoc Lib "Pilpxi.dll" (ByVal CardNum As Long, ByRef Bus As Long, ByRef Slot As Long) As Long

Parameters:

CardNum - card number

Bus - reference to variable to receive bus location

Slot - reference to variable to receive slot location

Returns:

Zero for success, or non-zero error code.

Closure Limit (Visual Basic)

Description

Obtains the maximum number of switches that may be activated simultaneously in the specified sub-unit. A single-channel multiplexer (MUX type) allows only one channel to be closed at any time. In some other models such as high-density matrix types a limit is imposed to prevent overheating; although it is possible to disable the limit for these types (see PIL_SetMode), doing so is not recommended.

Declaration

```
Declare Function PIL_ClosureLimit Lib "Pilpxi.dll" (ByVal CardNum As Long,
ByVal OutSub As Long, ByRef Limit As Long) As Long
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Limit - the variable to receive the result

Returns:

Zero for success, or non-zero error code.

Count Free Cards (Visual Basic)

Description

Obtains the number of installed cards that are operable by the Pilpxi driver but are not currently opened by it.

Declaration

Declare Function PIL_CountFreeCards Lib "Pilpxi.dll" (ByRef NumCards As Long) As Long

Parameters:

NumCards - reference to variable to receive the result

Returns:

Zero for success, or non-zero error code.

Diagnostic (Visual Basic)

Description

Obtains the diagnostic string of the specified card, giving expanded information on any fault conditons indicated by the PIL_Status value.

Declaration

Declare Function PIL_Diagnostic Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal Str As String) As Long

Parameters:

CardNum - card number

Str - reference to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Notes

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(Str, character_count).

The result string may include embedded newline characters, coded as the ASCII <linefeed> character (&H0A). If the string is to be displayed they must be processed to convert them to vbCrLf.

The length of the result string will not exceed the value of the driver constant MAX_DIAG_LENGTH.

Warning

Formatting and content of the diagnostic string may change as enhanced diagnostic features are made available. It should therefore not be interpreted programatically.

Enumerate Sub-units (Visual Basic)

Description

Obtains the numbers of input and output sub-units implemented on the specified card.

Declaration

Declare Function PIL_EnumerateSubs Lib "Pilpxi.dll" (ByVal CardNum As Long, ByRef InSubs As Long, ByRef OutSubs As Long) As Long

Parameters:

CardNum - card number

InSubs - reference to variable to receive the number of INPUT sub-units

OutSubs - reference to variable to receive the number of OUTPUT sub-units

Returns:

Zero for success, or non-zero error code.

Error Message (Visual Basic)

Description

Obtains a string description of the error codes returned by other driver functions.

Declaration

Declare Function PIL_ErrorMessage Lib "Pilpxi.dll" (ByVal ErrorCode As Long, ByVal Str As String) As Long

Parameters:

ErrorCode - the error code to be described

Str - reference to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Notes

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(Str, character_count).

Find Free Cards (Visual Basic)

Description

Obtains the logical bus and slot locations of installed cards that are operable by the Pilpxi driver and are currently unopened. These values are used with PIL_OpenSpecifiedCard.

Declaration

```
Declare Function PIL_FindFreeCards Lib "Pilpxi.dll" (ByVal NumCards As Long,
ByRef BusList As Long, ByRef SlotList As Long) As Long
```

Parameters:

NumCards - the number of cards (maximum) for which information is to be obtained

BusList - reference to the one-dimensional array (vector) to receive cards' bus location values

SlotList - reference to the one-dimensional array (vector) to receive cards' slot location values

Returns:

Zero for success, or non-zero error code.

Notes

The bus and slot locations of the first card found are placed respectively in the least significant elements of the BusList and SlotList arrays. Successive elements contain the values for further cards.

If the value given for NumCards is less than the number of cards currently accessible, information is obtained only for the number of cards specified.

To use this function in Visual Basic, it must be passed references to the first elements of the data arrays. For example, assuming zero-based arrays:

```
PIL_FindFreeCards(NumCards, BusList(0), SlotList(0))
```

Warning

The arrays referenced must have been assigned at least as many elements as the number of cards for which information is being requested or adjacent memory will be overwritten, causing data corruption and/or a program crash. The number of accessible cards can be discovered using PIL_CountFreeCards.

Settle Time (Visual Basic)

Description

Obtains a sub-unit's settling time (or debounce period - the time taken for its switches to stabilise). By default, Pilpxi driver functions retain control during this period so that switches are guaranteed to have stabilised on completion. This mode of operation can be overridden if required - see PIL_SetMode.

Declaration

Declare Function PIL_SettleTime Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Ti As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Ti - the variable to receive the result (in microseconds)

Returns:

Zero for success, or non-zero error code.

Card Status (Visual Basic)

Description

Obtains the current status flags for the specified card.

Declaration

Declare Function PIL_Status Lib "Pilpxi.dll" (ByVal CardNum As Long) As Long

Parameters:

CardNum - card number

Returns:

A value representing the card's status flags.

Status Bit Definitions

Status bits are as follows:

&H80000000 - STAT_NO_CARD (no card with specified number)

&H40000000 - STAT_WRONG_DRIVER (card requires newer driver)

&H20000000 - STAT_EEPROM_ERR (card EEPROM fault)

&H10000000 - STAT_DISABLED (card disabled)

&H04000000 - STAT_BUSY (card operations not completed)

&H02000000 - STAT_HW_FAULT (card hardware defect)

&H01000000 - STAT_PARITY_ERROR (PCIbus parity error)

&H00080000 - STAT_CARD_INACCESSIBLE (Card cannot be accessed - failed/removed/unpowered)

&H00040000 - STAT_UNCALIBRATED (one or more sub-units is uncalibrated)

&H00020000 - STAT_CALIBRATION_DUE (one or more sub-units is due for calibration)

&H00000000 - STAT_OK (card functional and stable)

Corresponding global constants are provided in Pilpxi.bas.

Notes

Certain status bits are relevant only for specific classes of sub-unit, or for those having particular characteristics.

At card level, STAT_BUSY indicates if any of a card's sub-units have not yet stabilised.

Diagnostic information on fault conditions indicated in the status value can be obtained using PIL_Diagnostic.

Related functions

PIL_SubStatus

Sub-unit Information (Visual Basic)

Description

Obtains a description of a sub-unit, as numeric values.

Declaration

Declare Function PIL_SubInfo Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Out As Boolean, ByRef TypeNum As Long, ByRef Rows As Long, ByRef Cols As Long) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Out - sub-unit function: 0 for INPUT, 1 for OUTPUT

TypeNum - reference to variable to receive type code

Rows - reference to variable to receive row count

Cols - reference to variable to receive column count

Returns:

Zero for success, or non-zero error code.

Results

Output sub-unit type codes are:

1 - TYPE_SW (uncommitted switch)

2 - TYPE_MUX (multiplexer single-channel)

3 - TYPE_MUXM (multiplexer, multi-channel)

4 - TYPE_MAT (matrix - LF)

5 - TYPE_MATR (matrix - RF)

6 - TYPE_DIG (digital outputs)

7 - TYPE_RES (programmable resistor)

8 - TYPE_ATTEN (programmable RF attenuator)

9 - TYPE_PSUDC (DC power supply)

10 - TYPE_BATT (Battery Simulator)

Corresponding global constants are provided in Pilpxi.bas.

Input sub-unit type codes are:

1 - INPUT

Row and column values give the dimensions of the sub-unit. For all types other than matrices the column value contains the significant dimension: their row value is always '1'.

Note

Some sub-unit types are supported by functions providing alternate and/or more detailed information. These include:

TYPE_ATTEN - PIL_AttenInfo

TYPE_PSUDC - PIL_PsuInfo

Sub-unit Status (Visual Basic)

Description

Obtains the current status flags for the specified output sub-unit. Status bits associated with significant card-level conditions are also returned.

Declaration

Declare Function PIL_SubStatus Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Returns:

A value representing the sub-unit's status flags.

Status Bit Definitions

Status bits are as follows:

&H80000000 - STAT_NO_CARD (no card with specified number)

&H40000000 - STAT_WRONG_DRIVER (card requires newer driver)

&H20000000 - STAT_EEPROM_ERR (card EEPROM fault)

&H10000000 - STAT_DISABLED (card disabled)

&H08000000 - STAT_NO_SUB (no sub-unit with specified number)

&H04000000 - STAT_BUSY (sub-unit operations not completed)

&H02000000 - STAT_HW_FAULT (card hardware defect)

&H01000000 - STAT_PARITY_ERROR (PCIbus parity error)

&H00800000 - STAT_PSU_INHIBITED (power supply output is disabled - by software)

&H00400000 - STAT_PSU_SHUTDOWN (power supply output is shutdown - due to overload)

&H00200000 - STAT_PSU_CURRENT_LIMIT (power supply is operating in current-limited mode)

&H00100000 - STAT_CORRUPTED (sub-unit logical state is corrupted)

&H00080000 - STAT_CARD_INACCESSIBLE (Card cannot be accessed - failed/removed/unpowered)

&H00040000 - STAT_UNCALIBRATED (sub-unit is uncalibrated)

&H00020000 - STAT_CALIBRATION_DUE (sub-unit is due for calibration)

&H00000000 - STAT_OK (sub-unit functional and stable)

Corresponding global constants are provided in Pilpxi.bas.

Notes

Certain status bits are relevant only for specific classes of sub-unit, or for those having particular characteristics.

Diagnostic information on fault conditions indicated in the status value can be obtained using PIL_Diagnostic.

Related functions

PIL_Status

Sub-unit Type (Visual Basic)

Description

Obtains a description of a sub-unit, as a text string.

Declaration

Declare Function PIL_SubType Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Out As Boolean, ByVal Str As String) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Out - sub-unit function: 0 for INPUT, 1 for OUTPUT

Str - reference to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Type string	**Description**
INPUT(<size>)	Digital inputs
SWITCH(<size>)	Uncommitted switches
MUX(<size>)	Multiplexer, single-channel only
MUXM(<size>)	Multiplexer, multi-channel
MATRIX(<columns>X<rows>)	Matrix, LF
MATRIXR(<columns>X<rows>)	Matrix, RF
DIGITAL(<size>)	Digital Outputs
RES(<number of resistors in chain>)	Programmable resistor
ATTEN(<number of pads>)	Programmable RF attenuator
PSUDC(0)	DC Power Supply
BATT(<Voltage DAC resolution, bits>)	Battery Simulator

Notes

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(Str, character_count).

Some sub-unit types are supported by functions providing more detailed information. These include:

ATTEN - PIL_AttenType

PSUDC - PIL_PsuType

Version (Visual Basic)

Description

Obtains the driver version code.

Declaration

```
Declare Function PIL_Version Lib "Pilpxi.dll" () As Long
```

Parameters:

None.

Returns:

The driver version code, multiplied by 100 (i.e. a value of 100 represents version 1.00)

Switching and General Purpose Output

Switching and General Purpose Output

This section details the use in Visual Basic of functions that are applicable to most output sub-unit types.

Note that although these functions may be used with them, some sub-unit types - for example matrix and programmable RF attenuator - are also served by specific functions offering more straightforward control.

Functions are provided to:

- Clear all output channels of all open Pickering cards: PIL_ClearAll
- Clear all output channels of a single Pickering card: PIL_ClearCard
- Clear all output channels of a sub-unit: PIL_ClearSub
- Open or close a single output channel: PIL_OpBit
- Set a sub-unit's output pattern: (PIL_WriteSub), PIL_WriteSubArray
- Obtain the state of a single output channel: PIL_ViewBit
- Obtain a sub-unit's output pattern: (PIL_ViewSub), PIL_ViewSubArray

Clear All (Visual Basic)

Description

Clears (de-energises or sets to logic '0') all outputs of all sub-units of every open Pickering card.

Declaration

Declare Function PIL_ClearAll Lib "Pilpxi.dll" () As Long

Parameters:

None.

Returns:

Zero for success, or non-zero error code.

Clear Card (Visual Basic)

Description

Clears (de-energises or sets to logic '0') all outputs of all sub-units of the specified Pickering card.

Declaration

```
Declare Function PIL_ClearCard Lib "Pilpxi.dll" (ByVal CardNum As Long) As Long
```

Parameters:

CardNum - card number

Returns:

Zero for success, or non-zero error code.

Clear Sub-unit (Visual Basic)

Description

Clears (de-energises or sets to logic '0') all outputs of a sub-unit.

Declaration

Declare Function PIL_ClearSub Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Returns:

Zero for success, or non-zero error code.

Operate Bit (Visual Basic)

Description

Operate a single output channel or bit.

Note that in the case of a single-channel multiplexer (MUX type) any existing channel closure will be cleared automatically prior to selecting the new channel.

Note that PIL_OpCrosspoint allows more straightforward use of row/column co-ordinates with matrix sub-units.

Declaration

Declare Function PIL_OpBit Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal BitNum As Long, ByVal Action As Boolean) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

BitNum - output bit number

Action - 1 to energise, 0 to de-energise

Returns:

Zero for success, or non-zero error code.

View Bit (Visual Basic)

Description

Obtains the state of an individual output.

Declaration

Declare Function PIL_ViewBit Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal BitNum As Long, ByRef State As Boolean) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

BitNum - output bit number

State - the variable to receive the result (0 = OFF or logic '0', 1 = ON or logic '1')

Returns:

Zero for success, or non-zero error code.

View Sub-unit (Visual Basic)

Description

Obtains the state of all outputs of a sub-unit. The result fills the number of least significant bits corresponding to the size of the sub-unit.

Declaration

```
Declare Function PIL_ViewSub Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Data As Long) As Long
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - reference to the one-dimensional array (vector) to receive the result

Returns:

Zero for success, or non-zero error code.

Notes

Although this function is usable in Visual Basic, PIL_ViewSubArray should be preferred because it incorporates bounds-checking and other safety features.

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable for the result:

```
PIL_ViewSub(CardNum, OutSub, Data)
```

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

```
PIL_ViewSub(CardNum, OutSub, Data(0))
```

For a Matrix sub-unit, the result is folded into the vector on its row-axis: see Data Formats.

Warning

The data array referenced must contain sufficient bits to hold the bit-pattern for the specified sub-unit, or adjacent memory will be overwritten, causing data corruption and/or a program crash.

Example Code

See the description of PIL_WriteSub for example code using an array-based function.

View Sub-unit - Array (Visual Basic)

Description

Obtains the state of all outputs of a sub-unit. The result fills the number of least significant bits corresponding to the size of the sub-unit.

Declaration

Declare Function PIL_ViewSubArray Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Data() As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - reference to the one-dimensional array (vector) to receive the result

Returns:

Zero for success, or non-zero error code.

Notes

This function must be passed a reference to the data array, for example:

PIL_ViewSubArray(CardNum, OutSub, Data())

For a Matrix sub-unit, the result is folded into the vector on its row-axis: see Data Formats.

Example Code

See the description of PIL_WriteSubArray for example code using a safe array-based function.

Write Sub-unit (Visual Basic)

Description

Sets all outputs of a sub-unit to the supplied bit-pattern. The number of least significant bits corresponding to the size of the sub-unit are written.

Declaration

```
Declare Function PIL_WriteSub Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Data As Long) As Long
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - reference to the one-dimensional array (vector) containing the bit-pattern to be written

Returns:

Zero for success, or non-zero error code.

Notes

Although this function is usable in Visual Basic, PIL_WriteSubArray should be preferred because it incorporates bounds-checking and other safety features.

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable containing the bit-pattern:

```
PIL_WriteSub(CardNum, OutSub, Data)
```

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

```
PIL_WriteSub(CardNum, OutSub, Data(0))
```

For a Matrix sub-unit, the data is folded into the vector on its row-axis: see Data Formats.

Warning

The data array referenced must contain sufficient bits to represent the bit-pattern for the specified sub-unit, or undefined data will be written to the more significant bits.

Example Code

For clarity, this example omits initialising the variables CardNum, OutSub etc. and does no error-checking.

```
' Dimension a longword data array (index base zero) to contain the

' number of bits necessary to represent the sub-unit (e.g. 2
longwords

' supports sub-units having upto 64 switches)

Dim Data(1) As Long ' Value specifies the highest allowed index


' Data(0) bit 0 represents switch #1

' Data(0) bit 1 represents switch #2

' ... etc.

' Data(0) bit 31 represents switch #32

' Data(1) bit 0 represents switch #33

' ... etc.


' Setup array data to turn on switches 3, 33 and output to the card

Data(0) = &H4 ' set longword 0 bit 2 (switch 3)

Data(1) = &H1 ' set longword 1 bit 0 (switch 33)

Result = PIL_WriteSub(CardNum, OutSub, Data(0))


' Add switch 4 to the array and output to the card

Data(0) = (Data(0) Or &H8) ' set longword 0 bit 3 (switch 4)

Result = PIL_WriteSub(CardNum, OutSub, Data(0))

' ... now have switches 3, 4, 33 energised


' Delete switch 33 from the array and output to the card
```

```
Data(1) = (Data(1) And &HFFFFFFFE) ' clear longword 1 bit 0 (switch
33)

Result = PIL_WriteSub(CardNum, OutSub, Data(0))

' ... leaving switches 3 and 4 energised
```

Write Sub-unit - Array (Visual Basic)

Description

Sets all outputs of a sub-unit to the supplied bit-pattern. The number of least significant bits corresponding to the size of the sub-unit are written.

Declaration

Declare Function PIL_WriteSubArray Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Data() As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - reference to the one-dimensional array (vector) containing the bit-pattern to be written

Returns:

Zero for success, or non-zero error code.

Notes

This function must be passed a reference to the data array, for example:

PIL_WriteSubArray(CardNum, OutSub, Data())

For a Matrix sub-unit, the data is folded into the vector on its row-axis: see Data Formats.

Example Code

For clarity, this example omits initialising the variables CardNum, OutSub etc. and does no error-checking.

```
' Dimension a longword data array (index base zero) to contain the

' number of bits necessary to represent the sub-unit (e.g. 2
longwords

' supports sub-units having upto 64 switches)

Dim Data(1) As Long ' Value specifies the highest allowed index
```

```
' Data(0) bit 0 represents switch #1

' Data(0) bit 1 represents switch #2

' ... etc.

' Data(0) bit 31 represents switch #32

' Data(1) bit 0 represents switch #33

' ... etc.


' Setup array data to turn on switches 3, 33 and output to the card

Data(0) = &H4 ' set longword 0 bit 2 (switch 3)

Data(1) = &H1 ' set longword 1 bit 0 (switch 33)

Result = PIL_WriteSubArray(CardNum, OutSub, Data())


' Add switch 4 to the array and output to the card

Data(0) = (Data(0) Or &H8) ' set longword 0 bit 3 (switch 4)

Result = PIL_WriteSubArray(CardNum, OutSub, Data())

' ... now have switches 3, 4, 33 energised


' Delete switch 33 from the array and output to the card

Data(1) = (Data(1) And &HFFFFFFFE) ' clear longword 1 bit 0 (switch
33)

Result = PIL_WriteSubArray(CardNum, OutSub, Data())

' ... leaving switches 3 and 4 energised
```

Specialised Switching

Specialised Switching

This section details the use in Visual Basic of functions specific to particular types of switching sub-unit (uncommitted switches, multiplexer, matrix and digital output types).

Matrix operations

- Open or close a single matrix crosspoint: PIL_OpCrosspoint
- Obtain the state of a single matrix crosspoint: PIL_ViewCrosspoint

Individual switch operations, complex matrix sub-units

- Obtain/set the state of an individual switch: PIL_OpSwitch

Attribute information, complex matrix sub-units

- Obtain sub-unit attribute values: PIL_SubAttribute

Operate Crosspoint (Visual Basic)

Description

Operate a single matrix crosspoint.

Declaration

Declare Function PIL_OpCrosspoint Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Row As Long, ByVal Column As Long, ByVal Action As Boolean) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Row - crosspoint row (Y) location

Column - crosspoint column (X) location

Action - 1 to energise, 0 to de-energise

Returns:

Zero for success, or non-zero error code.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized bit-number method employed by PIL_OpBit. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

Related Matrix Functions

PIL_ViewCrosspoint

PIL_MaskCrosspoint

PIL_ViewMaskCrosspoint

Operate switch (Visual Basic)

Description

This function obtains, and optionally sets, the state of a switch. It allows explicit access to the individual switches making up a sub-unit, in types where their operation is normally handled automatically by the driver. The main purpose of this is in implementing fault diagnostic programs for such types; it can also be used where normal automated behaviour does not suit an application.

Declaration

Declare Function PIL_OpSwitch Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal SwitchFunc As Long, ByVal SegNum As Long, ByVal SwitchNum As Long, ByVal SubSwitch As Long, ByVal SwitchAction As Long, ByRef State As Boolean) As Long

Parameters:

CardNum - card number

OutSub - sub-unit number

SwitchFunc - code indicating the functional group of the switch, see below

SegNum - the segment location of the switch

SwitchNum - the number of the switch in its functional group (unity-based)

SubSwitch - the number of the subswitch to operate (unity-based)

SwitchAction - code indicating the action to be performed, see below

State - reference to variable to receive the state of the switch (after performing any action)

Returns:

Zero for success, or non-zero error code.

Applicable sub-unit types

This function is only usable with matrix sub-units having auto-isolation and/or auto-loopthru features. For further information see: segmented matrix, unsegmented matrix.

SwitchFunc value

A value indicating the functional group of the switch to be accessed.

Value	Ident	Function

0	SW_FUNC_CHANNEL	A channel (matrix crosspoint) switch
1	SW_FUNC_X_ISO	A matrix X-isolation switch
2	SW_FUNC_Y_ISO	A matrix Y-isolation switch
3	SW_FUNC_X_LOOPTHRU	A matrix X-loopthru switch
4	SW_FUNC_Y_LOOPTHRU	A matrix Y-loopthru switch

SegNum value

The segment location of the switch. The numbers and sizes of segments on each matrix axis can be obtained using PIL_SubAttribute.

In an unsegmented matrix, use SegNum = 1.

In a segmented matrix, segment numbers for crosspoint and isolation switches are determined logically.

SwitchNum value

The number of the switch in its functional group (unity-based).

For channel (crosspoint) switches, the switch number can be either:

- if SegNum is zero, the global channel number of the switch (see output bit number)
- if SegNum is non-zero, the segment-local number of the switch, calculated in a similar way to the above

SubSwitch value

The number of the subswitch to operate (unity-based). This parameter caters for a situation in which a logical channel, isolation or loopthru switch is served by more than one physical relay (as for example when 2-pole operation is implemented using independently-driven single-pole relays).

The numbers of subswitches for each functional group can be obtained using PIL_SubAttribute.

SwitchAction value

A code indicating the action to be performed.

Value	**Ident**	**Function**

0	SW_ACT_NONE	No switch change - just set State result
1	SW_ACT_OPEN	Open switch
2	SW_ACT_CLOSE	Close switch

Loopthru switches

Loopthru switches are initialised by the driver to a **closed** state, which may mean that they are either energised or de-energised depending upon their type. In normal automated operation loopthru switches open when any crosspoint on their associated line is closed. Actions SW_ACT_CLOSE and SW_ACT_OPEN close or open loopthru switch contacts as their names imply.

Operational considerations

This function can be used to alter a pre-existing switch state in a sub-unit, set up by fuctions such as PIL_OpBit or PIL_WriteSubArray. However once the state of any switch is changed by PIL_OpSwitch the logical state of the sub-unit is considered to have been destroyed. This condition is flagged in the result of PIL_SubStatus (bit STAT_CORRUPTED). Subsequent attempts to operate it using 'ordinary' switch functions such as PIL_OpBit, PIL_ViewBit etc. will fail (result ER_STATE_CORRUPT). Normal operation can be restored by clearing the sub-unit using PIL_ClearSub, PIL_ClearCard or PIL_ClearAll.

View Crosspoint (Visual Basic)

Description

Obtains the state of an individual matrix crosspoint.

Declaration

Declare Function PIL_ViewCrosspoint Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Row As Long, ByVal Column As Long, ByRef State As Boolean) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Row - crosspoint row (Y) location

Column - crosspoint column (X) location

State - the variable to receive the result (0 = OFF, 1 = ON)

Returns:

Zero for success, or non-zero error code.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized bit-number method employed by PIL_ViewBit. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

Sub-unit Attribute (Visual Basic)

Description

Obtains the value of a sub-unit attribute. These values facilitate operation using PIL_OpSwitch.

Declaration

Declare Function PIL_SubAttribute Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Out As Boolean, ByVal AttrCode As Long, ByRef AttrValue As Long) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Out - sub-unit function: 0 for INPUT, 1 for OUTPUT

AttrCode - a value indicating the sub-unit attribute to be queried, see below

AttrValue - reference to variable to receive the attribute's value

Returns:

Zero for success, or non-zero error code.

AttrCode values

Value	Ident	Function
1	SUB_ATTR_CHANNEL_SUBSWITCHES	Gets number of subswitches per logical channel (matrix crosspoint)
2	SUB_ATTR_X_ISO_SUBSWITCHES	Gets number of subswitches per logical X-isolator
3	SUB_ATTR_Y_ISO_SUBSWITCHES	Gets number of subswitches per logical Y-isolator
4	SUB_ATTR_X_LOOPTHRU_SUBSWITCHES	Gets number of subswitches per logical X-loopthru
5	SUB_ATTR_Y_LOOPTHRU_SUBSWITCHES	Gets number of subswitches per logical X-loopthru
&H100	SUB_ATTR_NUM_X_SEGMENTS	Gets number of X-axis segments
&H101	SUB_ATTR_X_SEGMENT01_SIZE	Gets size of X-axis segment 1
&H102	SUB_ATTR_X_SEGMENT02_SIZE	Gets size of X-axis segment 2
&H103	SUB_ATTR_X_SEGMENT03_SIZE	Gets size of X-axis segment 3
&H104	SUB_ATTR_X_SEGMENT04_SIZE	Gets size of X-axis segment 4
&H105	SUB_ATTR_X_SEGMENT05_SIZE	Gets size of X-axis segment 5

&H106	SUB_ATTR_X_SEGMENT06_SIZE	Gets size of X-axis segment 6
&H107	SUB_ATTR_X_SEGMENT07_SIZE	Gets size of X-axis segment 7
&H108	SUB_ATTR_X_SEGMENT08_SIZE	Gets size of X-axis segment 8
&H109	SUB_ATTR_X_SEGMENT09_SIZE	Gets size of X-axis segment 9
&H10A	SUB_ATTR_X_SEGMENT10_SIZE	Gets size of X-axis segment 10
&H10B	SUB_ATTR_X_SEGMENT11_SIZE	Gets size of X-axis segment 11
&H10C	SUB_ATTR_X_SEGMENT12_SIZE	Gets size of X-axis segment 12
&H200	SUB_ATTR_NUM_Y_SEGMENTS	Gets number of Y-axis segments
&H201	SUB_ATTR_Y_SEGMENT01_SIZE	Gets size of y-axis segment 1
&H202	SUB_ATTR_Y_SEGMENT02_SIZE	Gets size of y-axis segment 2

Switch Masking

Switch Masking

This section details the use in Visual Basic of switch masking functions.

Masking permits disabling operation of chosen switch channels by the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions. These functions report error ER_OUTPUT_MASKED if an attempt is made to activate a masked channel.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Masking functions, all switching sub-unit types

- Clear a sub-unit's mask: PIL_ClearMask
- Mask or unmask a single output channel: PIL_MaskBit
- Set a sub-unit's mask pattern: (PIL_WriteMask), PIL_WriteMaskArray
- Obtain the mask state of a single output channel: PIL_ViewMaskBit
- Obtain a sub-unit's mask pattern: (PIL_ViewMask), PIL_ViewMaskArray

Masking functions, matrix sub-units

- Mask or unmask a single matrix crosspoint: PIL_MaskCrosspoint
- Obtain the mask state of a single matrix crosspoint: PIL_ViewMaskCrosspoint

Note

Masking only allows output channels to be disabled in the OFF state; applying a mask to a channel that is already turned ON forces it OFF.

Clear Mask (Visual Basic)

Description

Clears a sub-unit's switch mask, enabling operation of all outputs by the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

Declaration

Declare Function PIL_ClearMask Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Returns:

Zero for success, or non-zero error code.

Mask Bit (Visual Basic)

Description

Mask or unmask a single output bit.

Masking disables the corresponding switch for the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits. Note that PIL_MaskCrosspoint allows more straightforward use of row/column co-ordinates with matrices.

Declaration

Declare Function PIL_MaskBit Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal BitNum As Long, ByVal Action As Boolean) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

BitNum - output bit number

Action - 1 to mask, 0 to unmask

Returns:

Zero for success, or non-zero error code.

Note

Certain single-channel multiplexer (MUX type) sub-units have a default channel (that is, a channel that is connected when the sub-unit is in a 'cleared' state). This channel cannot be masked, and error ER_ILLEGAL_MASK is given if an attempt is made to mask it.

Mask Crosspoint (Visual Basic)

Description

Mask or unmask a single matrix crosspoint.

Masking disables the corresponding switch for the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

This facility is particularly useful to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Declaration

Declare Function PIL_MaskCrosspoint Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Row As Long, ByVal Column As Long, ByVal Action As Boolean) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Row - crosspoint row (Y) location

Column - crosspoint column (X) location

Action - 1 to mask, 0 to unmask

Returns:

Zero for success, or non-zero error code.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized bit-number method employed by PIL_MaskBit. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

View Mask (Visual Basic)

Description

Obtains the switch mask of a sub-unit. The result fills the number of least significant bits corresponding to the size of the sub-unit.

Declaration

Declare Function PIL_ViewMask Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Data As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - reference to the one-dimensional array (vector) to receive the result

Returns:

Zero for success, or non-zero error code.

Notes

Although this function is usable in Visual Basic, PIL_ViewMaskArray should be preferred because it incorporates bounds-checking and other safety features.

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable for the result:

PIL_ViewMask(CardNum, OutSub, Data)

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

PIL_ViewMask(CardNum, OutSub, Data(0))

For a Matrix sub-unit, the result is folded into the vector on its row-axis: see Data Formats.

Warning

The data array referenced must contain sufficient bits to hold the bit-pattern for the specified sub-unit, or adjacent memory will be overwritten, causing data corruption and/or a program crash.

Example Code

See the description of PIL_WriteSub for example code using an array-based function.

View Mask - Array (Visual Basic)

Description

Obtains the switch mask of a sub-unit. The result fills the number of least significant bits corresponding to the size of the sub-unit.

Declaration

Declare Function PIL_ViewMaskArray Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Data() As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - reference to the one-dimensional array (vector) to receive the result

Returns:

Zero for success, or non-zero error code.

Notes

This function must be passed a reference to the data array, for example:

PIL_ViewMaskArray(CardNum, OutSub, Data())

For a Matrix sub-unit, the result is folded into the vector on its row-axis: see Data Formats.

Example Code

See the description of PIL_WriteSubArray for example code using a safe array-based function.

View Mask Bit (Visual Basic)

Description

Obtains the state of an individual output's mask.

Declaration

Declare Function PIL_ViewMaskBit Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal BitNum As Long, ByRef State As Boolean) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

BitNum - output bit number

State - the variable to receive the result (0 = unmasked, 1 = masked)

Returns:

Zero for success, or non-zero error code.

View Mask Crosspoint (Visual Basic)

Description

Obtains the state of an individual matrix crosspoint's mask.

Declaration

Declare Function PIL_ViewMaskCrosspoint Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Row As Long, ByVal Column As Long, ByRef State As Boolean) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Row - crosspoint row (Y) location

Column - crosspoint column (X) location

State - the variable to receive the result (0 = unmasked, 1 = masked)

Returns:

Zero for success, or non-zero error code.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized bit-number method employed by PIL_ViewMaskBit. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

Write Mask (Visual Basic)

Description

Sets a sub-unit's switch mask to the supplied bit-pattern. The number of least significant bits corresponding to the size of the sub-unit are written into the mask. A '1' bit in the mask disables the corresponding switch for the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Declaration

Declare Function PIL_WriteMask Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Data As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - reference to the one-dimensional array (vector) containing the mask pattern to be set

Returns:

Zero for success, or non-zero error code.

Notes

Although this function is usable in Visual Basic, PIL_WriteMaskArray should be preferred because it incorporates bounds-checking and other safety features.

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable containing the bit-pattern:

PIL_WriteMask(CardNum, OutSub, Data)

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

PIL_WriteMask(CardNum, OutSub, Data(0))

For a Matrix sub-unit, the mask data is folded into the vector on its row-axis: see Data Formats.

Certain single-channel multiplexer (MUX type) sub-units have a default channel (that is, a channel that is connected when the sub-unit is in a 'cleared' state). This channel cannot be masked, and error ER_ILLEGAL_MASK is given if an attempt is made to mask it.

Warning

The data array referenced must contain sufficient bits to represent the mask pattern for the specified sub-unit, or undefined data will be written to the more significant bits.

Example Code

See the description of PIL_WriteSub for example code using an array-based function.

Write Mask - Array (Visual Basic)

Description

Sets a sub-unit's switch mask to the supplied bit-pattern. The number of least significant bits corresponding to the size of the sub-unit are written into the mask. A '1' bit in the mask disables the corresponding switch for the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Declaration

Declare Function PIL_WriteMaskArray Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Data() As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - reference to the one-dimensional array (vector) containing the mask pattern to be set

Returns:

Zero for success, or non-zero error code.

Notes

This function must be passed a reference to the data array, for example:

PIL_WriteMaskArray(CardNum, OutSub, Data())

For a Matrix sub-unit, the mask data is folded into the vector on its row-axis: see Data Formats.

Certain single-channel multiplexer (MUX type) sub-units have a default channel (that is, a channel that is connected when the sub-unit is in a 'cleared' state). This channel cannot be masked, and error ER_ILLEGAL_MASK is given if an attempt is made to mask it.

Example Code

See the description of PIL_WriteSubArray for example code using a safe array-based function.

Input

Input

This section details the use in Visual Basic of functions specific to input sub-units.

Specific functions are provided to:

- Obtain the state of a single input: PIL_ReadBit
- Obtain a sub-unit's input pattern: PIL_ReadSub

Read Bit (Visual Basic)

Description

Obtains the state of an individual input.

Declaration

Declare Function PIL_ReadBit Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal InSub As Long, ByVal BitNum As Long, ByRef State As Boolean) As Long

Parameters:

CardNum - card number

InSub - input sub-unit number

BitNum - input bit number

State - the variable to receive the result (0 = logic '0', 1 = logic '1')

Returns:

Zero for success, or non-zero error code.

Read Sub-unit (Visual Basic)

Description

Obtains the current state of all inputs of a sub-unit.

Declaration

Declare Function PIL_ReadSub Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal InSub As Long, ByRef Data As Long) As Long

Parameters:

CardNum - card number

InSub - input sub-unit number

Data - reference to the one-dimensional array (vector) to receive the result

Returns:

Zero for success, or non-zero error code.

Note

For sub-units of 32 bits or less it is acceptable to pass a reference to a simple variable for the result:

PIL_ReadSub(CardNum, OutSub, Data)

For sub-units of more than 32 bits a reference must be passed to **the first element of a data array**. For example, assuming a zero-based array:

PIL_ReadSub(CardNum, OutSub, Data(0))

Warning

The Data object referenced must contain sufficient bits to hold the bit-pattern for the specified sub-unit, or adjacent memory will be overwritten, causing data corruption and/or a program crash.

Example Code

See the description of PIL_WriteSub for example code using an array-based function.

Calibration

Calibration

This section details the use in Visual Basic of functions associated with storing calibration values in a card's non-volatile memory. This facility is only available for certain sub-unit types, such as programmable resistors.

Specific functions are provided to:

- Retrieve an integer calibration value from non-volatile memory: PIL_ReadCal
- Store an integer calibration value in non-volatile memory: PIL_WriteCal
- Retrieve floating-point calibration value(s) from non-volatile memory: PIL_ReadCalFP
- Store floating-point calibration value(s) in non-volatile memory: PIL_WriteCalFP
- Retrieve a sub-unit's calibration date from non-volatile memory: PIL_ReadCalDate
- Store a sub-unit's calibration date in non-volatile memory: PIL_WriteCalDate
- Set a calibration point: PIL_SetCalPoint

Read Integer Calibration Value (Visual Basic)

Description

Reads an integer calibration value from on-card EEPROM.

Declaration

Declare Function PIL_ReadCal Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Idx As Long, ByRef Data As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Idx - calibration value index number (see below)

Data - reference to variable to receive result

Returns:

Zero for success, or non-zero error code.

Notes

This function is usable only with sub-units that support integer calibration data.

In simple programmable resistor models such as:

40-280

40-281

40-282

40-290

40-291

40-295

40-296

50-295

the Pilpxi driver places no interpretation on the stored value - an application program can utilise it in any way it wishes.

In some other models, including:

41-735-001

41-752-001

stored values are utilised by specific Pilpxi driver functions, and they should only be overwritten by an appropriate calibration utility.

For programmable resistors supporting this function the valid range of Idx values corresponds to the number of bits, i.e. to the range of output bit number values. A 16-bit resistor sub-unit typically provides 16 x 16-bit values.

The storage capacity of other types supporting this feature is determined by their functionality.

Related functions

PIL_WriteCal

Read Calibration Date (Visual Basic)

Description

Reads a sub-unit's calibration date and interval from on-card EEPROM.

Declaration

Declare Function PIL_ReadCalDate Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Store As Long, ByRef Year As Long, ByRef Day As Long, ByRef Interval As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Store - value indicating which store to access (see below)

Year - reference to variable to receive the year of calibration

Day - reference to variable to receive the day in the year of calibration

Interval - reference to variable to receive the calibration interval (in days)

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data; it can be used to discover when the sub-unit was last calibrated, and when recalibration will become due. Bit STAT_CALIBRATION_DUE in the result of PIL_Status or PIL_SubStatus indicates the need for recalibration.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "Store" Parameter	**Ident**	**Function**
0	CAL_STORE_USER	Access user calibration store
1	CAL_STORE_FACTORY	Access factory calibration store

Related functions

PIL_WriteCalDate

Read Floating-point Calibration Value (Visual Basic)

Description

Reads one or more floating-point calibration values from on-card EEPROM.

Declaration

Declare Function PIL_ReadCalFP Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Store As Long, ByVal Offset As Long, ByVal NumValues As Long, ByRef Data As Double) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Store - value indicating which store to access (see below)

Offset - the offset in the sub-unit's calibration store at which to start

NumValues - the number of values to be read

Data - reference to array to receive result

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as PIL_ResSetResistance. The number of values stored and their purpose is specific to the target sub-unit.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "Store" Parameter	**Ident**	**Function**
0	CAL_STORE_USER	Access user calibration store
1	CAL_STORE_FACTORY	Access factory calibration store

Related functions

PIL_WriteCalFP

Set Calibration Point (Visual Basic)

Description

Sets a sub-unit to a state corresponding to one of its defined calibration points.

Declaration

Declare Function PIL_SetCalPoint Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Idx As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Idx - the index number of the calibration point (see below)

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as PIL_ResSetResistance. The number of calibration points supported is specific to the target sub-unit.

The Idx value used by this function corresponds directly to the offset in the sub-unit's calibration store at which the value is to be stored and retrieved, using PIL_WriteCalFP and PIL_ReadCalFP.

WARNING

Selection of a calibration point causes the sub-unit to change state; the resulting state may be outside its normally desired range of operation. On completion of a calibration sequence, PIL_ResSetResistance can be used to normalise the setting.

Write Integer Calibration Value (Visual Basic)

Description

Writes an integer calibration value to on-card EEPROM.

Declaration

Declare Function PIL_WriteCal Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Idx As Long, ByVal Data As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Idx - calibration value index number (see below)

Data - the value to be written

Returns:

Zero for success, or non-zero error code.

Notes

This function is usable only with sub-units that support integer calibration data.

In simple programmable resistor models such as:

40-280

40-281

40-282

40-290

40-291

40-295

40-296

50-295

the Pilpxi driver places no interpretation on the stored value - an application program can utilise it in any way it wishes.

In some other models, including:

41-735-001

41-752-001

stored values are utilised by specific Pilpxi driver functions, and they should only be overwritten by an appropriate calibration utility.

The number of bits actually stored is specific to the target sub-unit - any redundant high-order bits of the supplied Data value are ignored.

For programmable resistors supporting this function the valid range of Idx values corresponds to the number of bits, i.e. to the range of output bit number values. A 16-bit resistor sub-unit typically provides 16 x 16-bit values.

The storage capacity of other types supporting this feature is determined by their functionality.

Related functions

PIL_ReadCal

Write Calibration Date (Visual Basic)

Description

Writes a sub-unit's calibration date and interval into on-card EEPROM. Date information is obtained from the current system date.

Declaration

Declare Function PIL_WriteCalDate Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Store As Long, ByVal Interval As Long) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Store - value indicating which store to access (see below)

Interval - the desired calibration interval (in days)

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as PIL_ResSetResistance. The number of values stored and their purpose is specific to the target sub-unit.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "Store" Parameter	**Ident**	**Function**
0	CAL_STORE_USER	Access user calibration store
1	CAL_STORE_FACTORY	Access factory calibration store

Related functions

PIL_ReadCalDate

Write Floating-point Calibration Value (Visual Basic)

Description

Writes one or more floating-point calibration values into on-card EEPROM.

Declaration

Declare Function PIL_WriteCalFP Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Store As Long, ByVal Offset As Long, ByVal NumValues As Long, ByRef Data As Double) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Store - value indicating which store to access (see below)

Offset - the offset in the sub-unit's calibration store at which to start

NumValues - the number of values to be written

Data - reference to array containing values to write

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as PIL_ResSetResistance. The number of values stored and their purpose is specific to the target sub-unit.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "Store" Parameter	Ident	Function
0	CAL_STORE_USER	Access user calibration store
1	CAL_STORE_FACTORY	Access factory calibration store

WARNING

Writing new values will affect the sub-unit's calibration.

Related functions

PIL_ReadCalFP

Programmable Resistor

Programmable Resistor

This section details the use in Visual Basic of functions specific to programmable resistor sub-units.

Detailed information about a programmable resistor sub-unit, if available, can be obtained using function PIL_ResInfo.

Precision models

Precision programmable resistor models such as 40-260-001 are supported by functions:

- PIL_ResGetResistance
- PIL_ResSetResistance

which allow chosen resistance values to be set.

Simple models

In models not supported by the above functions general purpose output functions such as PIL_WriteSubArray must be used to program resistance values by setting bit-patterns explicitly.

Models 40-280, 40-281 and 40-282 are configured as simple resistor/switch arrays and programming should be straightforward.

In models employing a series resistor chain - such as 40-290, 40-291 and 40-295 - each of a card's programmable resistors is implemented as a separate logical sub-unit constructed from a series chain of individual fixed resistor elements, each element having an associated shorting switch. In the cleared state all switches are open, giving the programmable resistor its maximum value. A nominal value of zero ohms is obtained by turning all switches ON; other values by turning on an appropriate pattern of switches.

In standard models the individual fixed resistors are arranged in a binary sequence, the least significant bit of the least significant element in the array passed to PIL_WriteSubArray corresponding to the lowest value resistor element. For example, in a standard model 40-290 16-bit resistor of 32768 ohms:

Data(0) bit 0 (value &H1) corresponds to the 0R5 resistor element

Data(0) bit 1 (value &H2) corresponds to the 1R0 resistor element

thru...

Data(0) bit 15 (value &H8000) corresponds to the 16384R resistor element

Setting a nominal value of 68 ohms (= 64 + 4 ohms) therefore requires Data(0) set to &HFF77 (the inverse of the binary pattern 0000 0000 1000 1000).

Special models may have some other arrangement, and may also include a fixed offset resistor that is permanently in circuit.

Non-volatile storage of calibration values is supported through the functions PIL_ReadCal and PIL_WriteCal.

See the application note on Simple Programmable Resistor Cards.

Summary of functions for normal operation of "Programmable Resistor" cards

Model(s)	Class	Functions
40-260-001	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
40-260-999	Precision	PIL_WriteSubArray
		PIL_ViewSubArray
40-261	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
40-262	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
40-265	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
40-280, 40-281, 40-282	Simple	PIL_OpBit
		PIL_ViewBit
		PIL_WriteSubArray
		PIL_ViewSubArray
		PIL_ReadCal
		PIL_WriteCal
40-290, 40-291	Simple	PIL_WriteSubArray
		PIL_ViewSubArray
		PIL_ReadCal
		PIL_WriteCal
40-295	Simple	PIL_WriteSubArray
		PIL_ViewSubArray

		PIL_ReadCal
		PIL_WriteCal
40-296	Simple	PIL_WriteSubArray
		PIL_ViewSubArray
		PIL_ReadCal
		PIL_WriteCal
40-297	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
50-295	Simple	PIL_WriteSubArray
		PIL_ViewSubArray
		PIL_ReadCal
		PIL_WriteCal
50-297	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
...		

Get Resistance Value (Visual Basic)

Description

Obtains the current resistance setting of the specified programmable resistor. This function is only usable with programmable resistor models that support it: such capability is indicated in the result of PIL_ResInfo.

The value obtained for a resistance setting of infinity, if the sub-unit permits this, is **HUGE_VAL**.

Declaration

Declare Function PIL_ResGetResistance Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByRef Resistance As Double) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Resistance - reference to variable to receive the result

Returns:

Zero for success, or non-zero error code.

Related functions

PIL_ResInfo

PIL_ResSetResistance

Resistor information (Visual Basic)

Description

Obtains detailed information on a programmable resistor sub-unit.

Declaration

Declare Function PIL_ResInfo Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByRef MinRes As Double, ByRef MaxRes As Double, ByRef RefRes As Double, ByRef PrecPC As Double, ByRef PrecDelta As Double, ByRef Int1 As Double, ByRef IntDelta As Double, ByRef Capabilities As Long) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

MinRes - reference to variable to receive minimum resistance setting

MaxRes - reference to variable to receive maximum resistance setting

RefRes - reference to variable to receive reference resistance value

PrecPC - reference to variable to receive percentage precision value

PrecDelta - reference to variable to receive offset precision, in ohms

Int1 - reference to (currently unused) variable

IntDelta - reference to variable to receive internal precision, in ohms

Capabilities - reference to variable to receive capability flags (see below)

Returns:

Zero for success, or non-zero error code.

Capabilities Bit Flag Definitions

Capability bits are as follows:

&H00000008 - RES_CAP_REF (supports reference calibration value)

&H00000004 - RES_CAP_INF (supports setting "open-circuit")

&H00000002 - RES_CAP_ZERO (supports setting "zero ohms")

&H00000001 - RES_CAP_PREC (precision resistor - supporting function PIL_ResSetResistance etc.)

&H00000000 - RES_CAP_NONE (no special capabilities)

Corresponding global constants are provided in Pilpxi.bas.

Notes

MinRes and MaxRes are the minimum and maximum values that can be set in the sub-unit's continuous range of adjustment. If capability RES_CAP_ZERO is flagged a setting of "zero ohms" is also possible. If RES_CAP_INF is flagged an open-circuit setting is also possible.

If capability RES_CAP_REF is flagged, RefRes is the reference resistance value - such as in model 40-265, where it gives the balanced state resistance.

PrecPC and PrecDelta represent the sub-unit's precision specification, such as (±0.2%, ±0.1 ohms).

IntDelta is the notional precision to which the sub-unit works internally; this value will be less than or equal to the figure indicated by PrecPC and PrecDelta, indicating greater internal precision.

Where information is not available for the sub-unit concerned, null values are returned.

Set Resistance Value (Visual Basic)

Description

Sets a programmable resistor to the closest available setting to the value specified. This function is only usable with programmable resistor models that support it: such capability is indicated in the result of PIL_ResInfo.

If the sub-unit permits, the resistance value can be set to:

- zero ohms (nominally), by passing the resistance value 0.0
- infinity, using function PIL_ClearSub

The resistance value actually set can be found using PIL_ResGetResistance.

Declaration

Declare Function PIL_ResSetResistance Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal OutSub As Long, ByVal Mode As Long, ByVal Resistance As Double) As Long

Parameters:

CardNum - card number

OutSub - output sub-unit number

Mode - the resistance setting mode (see below)

Resistance - the resistance value

Returns:

Zero for success, or non-zero error code.

Mode value

A value indicating how the given resistance value is to be applied. Only one mode is currently supported:

Value	Ident	Function
0	RES_MODE_SET	Set resistance to the specified value

Note

In programmable resistor models having gapped ranges, resistance values falling within such gaps are not coerced. For example, in a unit supporting settings:

- zero ohms
- 100 - 200 ohms continuously variable
- infinity

attempting to set values above zero but below 100 ohms, or above 200 ohms but less than infinity, gives error ER_BAD_RESISTANCE.

Related functions

PIL_ResInfo

PIL_ResGetResistance

Programmable Potentiometer

Programmable Potentiometer

This section details the use in Visual Basic of functions specific to programmable potentiometer sub-units.

No potentiometer-specific functions are currently provided.

A potentiometer such as model 40-296 is represented logically as a programmable resistor (RES type) having twice the number of switched bits as its nominal resolution, i.e. a 24-bit potentiometer returns the type description RES(48). To make the unit behave correctly appropriate bit-patterns must be set in the upper and lower halves using general purpose output function PIL_WriteSubArray (or PIL_WriteSub). Transient effects must be expected when changing the wiper position; provided MODE_NO_WAIT is not in force resistance values can only be transiently high.

Note that a potentiometer's state at power-up and when cleared is as a device of twice the nominal resistance with its wiper centred.

WARNING

Mis-programming can result in the potentiometer presenting a lower than normal resistance between its end terminals - in the worst case zero ohms.

Non-volatile (EEPROM) storage of calibration values is supported through the functions PIL_ReadCal and PIL_WriteCal.

Programmable RF Attenuator

Programmable RF Attenuator

This section details the use in Visual Basic of functions specific to programmable RF attenuator sub-units.

Specific functions are provided to:

- Obtain attenuator information, in numeric format: PIL_AttenInfo
- Obtain attenuator description, in string format: PIL_AttenType
- Set an attenuation level, in dB: PIL_AttenSetAttenuation
- Obtain the current attenuation setting, in dB: PIL_AttenGetAttenuation
- Obtain the value of each individual attenuator pad, in dB: PIL_AttenPadValue

RF attenuator sub-units can also be controlled using general purpose output functions such as PIL_WriteSubArray. This allows the explicit selection of particular pad patterns that may in some circumstances yield improved RF performance.

Get attenuation (Visual Basic)

Description

Obtains the current attenuation setting.

Declaration

Declare Function PIL_AttenGetAttenuation Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByRef Atten As Single) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Atten - reference to variable to receive the attenuation value, in dB

Returns:

Zero for success, or non-zero error code.

Attenuator information (Visual Basic)

Description

Obtains a description of an attenuator sub-unit, as numeric values.

Declaration

```
Declare Function PIL_AttenInfo Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByRef TypeNum As Long, ByRef NumSteps As Long, ByRef StepSize As Single) As Long
```

Parameters:

CardNum - card number

SubNum - sub-unit number

TypeNum - reference to variable to receive type code

NumSteps - reference to variable to receive step count

StepSize - reference to variable to receive step size, in dB

Returns:

Zero for success, or non-zero error code.

Results

RF attenuator sub-unit type code is:

8 - TYPE_ATTEN (programmable RF attenuator)

A corresponding global constant is provided in Pilpxi.bas.

Note

The description obtained by this function is a *logical* one; a *physical* description indicating the number of discrete pads in the attenuator can be obtained using PIL_SubInfo.

Attenuator pad value (Visual Basic)

Description

Obtains the attenuation value of a numbered pad.

Declaration

Declare Function PIL_AttenPadValue Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal PadNum As Long, ByRef Atten As Single) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

PadNum - pad number

Atten - reference to variable to receive the pad's attenuation value, in dB

Returns:

Zero for success, or non-zero error code.

Note

This function facilitates explicit pad selection using PIL_OpBit or PIL_WriteSubArray, if the selections made by PIL_attenSetAttenuation are not optimal for the application.

The number of pads in the sub-unit can be found using PIL_SubInfo.

Set attenuation (Visual Basic)

Description

Sets the attenuation to the specified value.

Declaration

Declare Function PIL_AttenSetAttenuation Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Atten As Single) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Atten - the attenuation value to set, in dB

Returns:

Zero for success, or non-zero error code.

Note

The combination of pads inserted to achieve the desired attenuation level is determined by the driver for best all-round performance. In some models it may be possible to optimise particular aspects of attenuator performance by setting other pad combinations explicitly using PIL_OpBit or PIL_WriteSubArray. The pad value associated with each output channel can be discovered with PIL_AttenPadValue.

Attenuator type (Visual Basic)

Description

Obtains a description of an attenuator sub-unit, as a text string.

Declaration

Declare Function PIL_AttenType Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Str As String) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Str - reference to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Result

The format of the result is "ATTEN(<number of steps>,<step size in dB>)".

Note

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(Str, character_count).

Note

The description obtained by this function is a *logical* one; a *physical* description indicating the number of discrete pads employed in the attenuator can be obtained using PIL_SubType.

Power Supplies

Power Supply functions

This section details the use in Visual Basic of functions specific to power supply sub-units.

Specific functions are provided to:

- Obtain power supply description, in string format: PIL_PsuType
- Obtain power supply information, in numeric format: PIL_PsuInfo
- Set power supply output voltage: PIL_PsuSetVoltage
- Obtain a power supply's current voltage setting: PIL_PsuGetVoltage
- Enable/disable a power supply's output: PIL_PsuEnable

Other functions that are relevant to operation of power supply sub-units include:

- Clear a power supply (restore start-up state): PIL_ClearSub
- Obtain power supply status information: PIL_SubStatus
- Retrieve a calibration value from non-volatile memory (some models): PIL_ReadCal
- Store a calibration value in non-volatile memory (some models): PIL_WriteCal

Power Supply - enable/disable output (Visual Basic)

Description

Enables or disables a power supply's output.

Declaration

Declare Function PIL_PsuEnable Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal State As Boolean) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

State - 1 to enable, 0 to disable output

Returns:

Zero for success, or non-zero error code.

Note

This function is usable only with sub-units having the capability PSU_CAP_OUTPUT_CONTROL - see PIL_PsuInfo.

Power Supply - Get Voltage (Visual Basic)

Description

Obtains the voltage setting of a power supply sub-unit.

Declaration

Declare Function PIL_PsuGetVoltage Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByRef Voltage As Double) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Voltage - reference to variable to receive the output setting, in Volts

Returns:

Zero for success, or non-zero error code.

Notes

The result is the nominal value to which the output has been set, not necessarily the actual voltage being output (which may be affected by device tolerances, current-limit conditions etc.).

This function is also usable with fixed-voltage supplies, returning the nominal output voltage.

Power Supply - Information (Visual Basic)

Description

Obtains a description of a power supply sub-unit, as numeric values.

Declaration

Declare Function PIL_PsuInfo Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByRef TypeNum As Long, ByRef Voltage As Double, ByRef Current As Double, ByRef Precis As Long, ByRef Capabilities As Long) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

TypeNum - reference to variable to receive type code

Voltage - reference to variable to receive rated voltage (in Volts)

Current - reference to variable to receive rated current (in Amps)

Precis - reference to variable to receive precision (in bits, meaningful only for programmable supplies)

Capabilities - reference to variable to receive capability flags (see below)

Returns:

Zero for success, or non-zero error code.

Results

Power supply sub-unit type code is:

9 - TYPE_PSUDC (DC power supply)

A corresponding global constant is provided in Pilpxi.bas.

Capability flag bit definitions:

&H00000010 - PSU_CAP_CURRENT_MODE_SENSE (can sense if operating in current-limited mode)

&H00000008 - PSU_CAP_PROG_CURRENT (output current is programmable)

&H00000004 - PSU_CAP_PROG_VOLTAGE (output voltage is programmable)

&H00000002 - PSU_CAP_OUTPUT_SENSE (has logic-level sensing of output active state)

&H00000001 - PSU_CAP_OUTPUT_CONTROL (has output on/off control)

Certain driver functions are only usable with sub-units having appropriate capabilities - examples being:

PIL_PsuEnable

PIL_PsuSetVoltage

Power Supply - Set Voltage (Visual Basic)

Description

Sets the output voltage of a power supply sub-unit to the specified value.

Declaration

Declare Function PIL_PsuSetVoltage Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Voltage As Double) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Voltage - the output voltage to set, in Volts

Returns:

Zero for success, or non-zero error code.

Notes

The voltage value specified is rounded to the precision of the supply's DAC. The actual voltage setting can be obtained using PIL_PsuGetVoltage.

This function is usable only with sub-units having the capability PSU_CAP_PROG_VOLTAGE - see PIL_PsuInfo.

Power Supply - Type (Visual Basic)

Description

Obtains a description of a power supply sub-unit, as a text string.

Declaration

Declare Function PIL_PsuType Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Str As String) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Str - reference to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Result

For a DC power supply the format of the result is "PSUDC(<rated voltage>,<rated current>)".

Note

The result is a C-style string, terminated by an ASCII null character. It can be converted to a Visual Basic string by counting the number of characters upto but excluding the terminating null, then performing:

VBstring = LEFT$(Str, character_count).

Note

More detailed information on power supply characteristics is obtainable in numeric format, using PIL_PsuInfo.

Battery Simulator

Battery Simulator

This section details the use in Visual Basic of functions specific to battery simulator models.

Models 41-750-001 and 41-751-001

No special-purpose functions are implemented for these models - they are operable using general-purpose input-output functions. See:

40-750-001

40-751-001

Model 41-752-001

Model 41-752-001 is implemented as an array of BATT sub-units, employing the following special-purpose functions for normal operation:

- Set output voltage: PIL_BattSetVoltage
- Obtain the present output voltage setting: PIL_BattGetVoltage
- Set sink current: PIL_BattSetCurrent
- Obtain the present sink current setting: PIL_BattGetCurrent
- Set output enable states: PIL_BattSetEnable
- Obtain present output enable states: PIL_BattGetEnable
- Obtain the present state of the hardware interlock: PIL_BattReadInterlockState

Battery Simulator - set voltage (Visual Basic)

Description

Sets the output voltage of battery simulator (BATT type) sub-units.

Declaration

Declare Function PIL_BattSetVoltage Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Voltage As Double) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Voltage - the output voltage to set, in Volts

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function sets the voltage of that sub-unit alone.

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), all of the card's BATT sub-units are set to the given voltage.

The voltage value specified is rounded to the precision of the sub-unit's DAC. The actual voltage setting can be obtained using PIL_BattGetVoltage.

Battery Simulator - get voltage (Visual Basic)

Description

Obtains the voltage setting of a battery simulator (BATT type) sub-unit.

Declaration

Declare Function PIL_BattGetVoltage Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByRef Voltage As Double) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Voltage - reference to variable to receive the output setting, in Volts

Returns:

Zero for success, or non-zero error code.

Notes

The result is the nominal value to which the output has been set, not necessarily the actual voltage being output (which could be affected by conditions such as current-limiting).

Battery Simulator - set current (Visual Basic)

Description

Sets the output sink current of battery simulator (BATT type) sub-units.

Declaration

Declare Function PIL_BattSetCurrent Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Current As Double) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Current - the output sink current to set, in Amps

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function sets the sink current of that sub-unit alone.

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), all of the card's BATT sub-units are set to the given current.

For non-zero values, output sink current is set to the nearest available value **greater** than that specified, typically using a low-precision DAC (e.g. 4-bit). The actual sink current setting can be obtained using PIL_BattGetCurrent.

Battery Simulator - get current (Visual Basic)

Description

Obtains the current sink setting of a battery simulator (BATT type) sub-unit.

Declaration

Declare Function PIL_BattGetCurrent Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByRef Current As Double) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Current - reference to variable to receive the output setting, in Amps

Returns:

Zero for success, or non-zero error code.

Battery Simulator - set enable (Visual Basic)

Description

Sets the output enable pattern of battery simulator (BATT type) sub-units.

Declaration

Declare Function PIL_BattSetEnable Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByVal Pattern As Long) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Pattern - the pattern of output enables to set

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function sets the output enable state of that sub-unit alone according to the least significant bit of Pattern (0 = OFF, 1 = ON).

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), enable states of all the card's BATT sub-units are set; bits in the supplied Pattern are utilised in ascending order of BATT sub-unit, i.e.

Pattern bit 0 = enable state of lowest numbered BATT sub-unit (0 = OFF, 1 = ON)

Pattern bit 1 = enable state of next numbered BATT sub-unit (0 = OFF, 1 = ON)

etc.

Note that the operation can fail (returning ER_EXECUTION_FAIL) if a necessary hardware interlock is disconnected.

The present enable pattern can be obtained using PIL_BattGetEnable.

Battery Simulator - get enable (Visual Basic)

Description

Obtains the output enable pattern of battery simulator (BATT type) sub-units.

Declaration

Declare Function PIL_BattGetEnable Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByRef Pattern As Long) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Pattern - reference to variable to receive the output enable pattern

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function gets the output enable state of that sub-unit alone in the least significant bit of Pattern (0 = OFF, 1 = ON).

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), enable states of all the card's BATT sub-units are obtained; bits in Pattern are assigned in ascending order of BATT sub-unit, i.e.

Pattern bit 0 = enable state of lowest numbered BATT sub-unit (0 = OFF, 1 = ON)

Pattern bit 1 = enable state of next numbered BATT sub-unit (0 = OFF, 1 = ON)

etc.

Battery Simulator - read interlock state (Visual Basic)

Description

Obtains the present state of a hardware interlock associated with battery simulator (BATT type) sub-units.

Declaration

Declare Function PIL_BattReadInterlockState Lib "Pilpxi.dll" (ByVal CardNum As Long, ByVal SubNum As Long, ByRef Interlock As Boolean) As Long

Parameters:

CardNum - card number

SubNum - sub-unit number

Interlock - reference to variable to receive the interlock state

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function gets the state of the hardware interlock associated with that sub-unit:

0 = interlock is "down"

1 = interlock is "up"

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), the function gets the summary state of all BATT sub-unit interlocks :

0 = one or more interlocks is "down"

1 = all interlocks are "up"

Model 41-752-001 has a single global interlock affecting all channels, and both modes above yield the same result.

Interlock "up" state is hardware-latched from the physical wired interlock by the action of PIL_BattSetEnable, when that function succeeds. Hence:

- If the "up" state is indicated, the physical interlock has remained intact and outputs are enabled as previously set by PIL_BattSetEnable.
- If the "down" state is indicated, the physical interlock has been broken and all outputs will have been disabled automatically through hardware.

Mode Control

Mode Control

This section details the use in Visual Basic of functions controlling the driver's operation.

This feature is implemented through a single function: PIL_SetMode.

Set Mode (Visual Basic)

Description

Allows control flags affecting the driver's global behaviour to be set and read. This function gives access to low-level control features of the Pilpxi driver and is intended for 'expert' use only - the default driver behaviour should be satisfactory for the great majority of applications.

Declaration

Declare Function PIL_SetMode Lib "Pilpxi.dll" (ByVal ModeFlags As Long) As Long

Parameters:

ModeFlags - new value for driver mode flags

Returns:

The driver's mode flags prior to executing this function.

Flag Bit Definitions

Flag bits are as follows:

&H00000000 - MODE_DEFAULT (standard operating mode)

&H00000001 - MODE_NO_WAIT (sequencing and settling time delays disabled)

&H00000002 - MODE_UNLIMITED (closure limits disabled - see **Warning** below)

&H00000004 - MODE_REOPEN (allow re-opening without clearing cards)

&H00000008 - MODE_IGNORE_TEST (enable card operation even if selftest fails - see **Warning** below)

Corresponding global constants are provided in Pilpxi.bas.

Warning - MODE_UNLIMITED

Use of MODE_UNLIMITED to disable the limit on the maximum number of switch closures permitted on high-density cards is **not** recommended, because it carries the danger of overheating and consequent damage to both the card itself and the system in which it is installed. See Closure Limits.

Warning - MODE_IGNORE_TEST

The MODE_IGNORE_TEST feature should be used with **extreme caution**. If a defective card is forcibly enabled, under some fault conditions a large number of outputs could be energised spuriously, resulting in overheating and consequent damage to both the card itself and the system in which it is installed. The intended purpose of this feature is to allow continued operation of a BRIC unit from which a daughtercard has been removed for maintenance. See BRIC Operation.

Visual C++

Visual C++

The following files are provided for Visual C++:

- Pilpxi.h
- Pilpxi.lib
- Pilpxi.dll

For implicit linking (the simplest method), Pilpxi.h and Pilpxi.lib must be accessible by Visual C++ at compile-time. Typically, copies of these files can be placed in the folder containing your application's source files; alternatively your Visual C++ project may be configured to access them in their installed location (or some other centralized location).

For explicit linking Pilpxi.lib is not required. Information on techniques for explicit linking can be found in MSDN reference. Another technique is "delay loading", again referenced in MSDN. These methods permit better error handling (within the application, instead of generating a system error dialog) for example if Pilpxi.dll cannot be accessed, or is an out-of-date version missing some vital function.

Pilpxi.dll must be accessible by your application at run-time. Windows searches a number of standard locations for DLLs in the following order:

1. The directory containing the executable module.
2. The current directory.
3. The Windows system directory.
4. The Windows directory.
5. The directories listed in the PATH environment variable.

Placing Pilpxi.dll in one of the Windows directories has the advantage that a single copy serves any number of applications that use it, but does add to the clutter of system DLLs stored there. The Pickering Setup program places a copy of Pilpxi.dll in the Windows system directory.

Visual C++ Function Tree

Initialise	
Initialise all cards	PIL_OpenCards
Initialise single card	PIL_OpenSpecifiedCard
Close	
Close all cards	PIL_CloseCards
Close single card	PIL_CloseSpecifiedCard
Card Information and Status	
Get card identification	PIL_CardId
Get card location	PIL_CardLoc
Get sub-unit closure limit	PIL_ClosureLimit
Get count of unopened cards	PIL_CountFreeCards
Get diagnostic information	PIL_Diagnostic
Get sub-unit counts	PIL_EnumerateSubs
Get description of an error	PIL_ErrorMessage
Get locations of unopened cards	PIL_FindFreeCards
Get sub-unit settling time	PIL_SettleTime
Get card status	PIL_Status
Get sub-unit information	PIL_SubInfo
Get sub-unit status	PIL_SubStatus
Get sub-unit description	PIL_SubType
Get driver version	PIL_Version
Switching and General Purpose Output	
Clear outputs of all open cards	PIL_ClearAll
Clear a single card's outputs	PIL_ClearCard
Clear a sub-unit's outputs	PIL_ClearSub
Set or clear a single output	PIL_OpBit
Get a single output's state	PIL_ViewBit
Get a sub-unit's output pattern	PIL_ViewSub
	PIL_ViewSubArray
Set a sub-unit's output pattern	PIL_WriteSub
	PIL_WriteSubArray
Specialised Switching	
Set or clear a matrix crosspoint	PIL_OpCrosspoint
Obtain/set the state of a switch	PIL_OpSwitch
Get sub-unit attribute	PIL_SubAttribute
Get a matrix crosspoint's state	PIL_ViewCrosspoint
Switch Masking	

Clear a sub-unit's mask	PIL_ClearMask
Set or clear a single output's mask	PIL_MaskBit
Set or clear a matrix crosspoint's mask	PIL_MaskCrosspoint
Get a sub-unit's mask pattern	PIL_ViewMask
	PIL_ViewMaskArray
Get a single output's mask state	PIL_ViewMaskBit
Get a matrix crosspoint's mask state	PIL_ViewMaskCrosspoint
Set a sub-unit's mask pattern	PIL_WriteMask
	PIL_WriteMaskArray
Input	
Read single input	PIL_ReadBit
Read input sub-unit pattern	PIL_ReadSub
Calibration	
Read an integer calibration value	PIL_ReadCal
Read a sub-unit's calibration data	PIL_ReadCalDate
Read floating-point calibration value(s)	PIL_ReadCalFP
Set Calibration Point	PIL_SetCalPoint
Write an integer calibration value	PIL_WriteCal
Write a sub-unit's calibration date	PIL_WriteCalDate
Write floating-point calibration value(s)	PIL_WriteCalFP
Programmable Resistor	
Get resistance value	PIL_ResGetResistance
Get resistor information	PIL_ResInfo
Set resistance value	PIL_ResSetResistance
Programmable RF Attenuator	
Get attenuation setting	PIL_AttenGetAttenuation
Get attenuator information	PIL_AttenInfo
Get the attenuation of a pad	PIL_AttenPadValue
Set attenuation level	PIL_AttenSetAttenuation
Get attenuator description	PIL_AttenType
Power Supplies	
Enable/disable output	PIL_PsuEnable
Get output voltage setting	PIL_PsuGetVoltage
Get PSU information	PIL_PsuInfo
Set output voltage	PIL_PsuSetVoltage
Get PSU description	PIL_PsuType
Battery Simulator	
Set voltage	PIL_BattSetVoltage
Get voltage	PIL_BattGetVoltage

Set current	PIL_BattSetCurrent
Get current	PIL_BattGetCurrent
Set enable	PIL_BattSetEnable
Get enable	PIL_BattGetEnable
Read interlock state	PIL_BattReadInterlockState
Mode Control	
Set driver mode	PIL_SetMode

Visual C++ Code Sample

File PILDEMO.C contains the source code for the PILDemo demonstration program, and illustrates usage of many of the driver's functions.

WARNING

WHEN RUN, THIS PROGRAM ACTIVATES OUTPUTS BOTH INDIVIDUALLY AND IN COMBINATIONS. IT SHOULD NOT BE RUN UNDER ANY CONDITIONS WHERE DAMAGE COULD RESULT FROM SUCH EVENTS. FOR GREATEST SAFETY IT SHOULD BE RUN ONLY WHEN NO EXTERNAL POWER IS APPLIED TO ANY CARD.

Initialise and Close

Initialise and Close

This section details the use in Visual C++ of functions for initialising and closing cards.

The Pilpxi driver supports two mechanisms for taking control of Pickering cards. The two mechanisms are mutually exclusive - the first use of one method after loading the driver DLL disables the other.

Controlling all cards

This method allows a single application program to open and access all installed Pickering cards. Using this method the cards are first opened by calling function PIL_OpenCards. Cards can then be accessed by other driver functions as necessary.

When the application has finished using the cards it should close them by calling function PIL_CloseCards.

Controlling cards individually

This method allows application programs to open and access Pickering cards on an individual basis. Using this method a card is first opened by calling function PIL_OpenSpecifiedCard. The card can then be accessed by other driver functions as necessary.

When the application has finished using the card it should be closed by calling function PIL_CloseSpecifiedCard.

Functions PIL_CountFreeCards and PIL_FindFreeCards assist in locating cards for opening by this mechanism.

Close All Cards (Visual C++)

Description

Closes all open Pickering cards, which must have been opened using PIL_OpenCards. This function should be called when the application program has finished using them.

Prototype

```
void _stdcall PIL_CloseCards(void);
```

Parameters:

None.

Returns:

Nothing.

Close Specified Card (Visual C++)

Description

Closes the specified Pickering card, which must have been opened using PIL_OpenSpecifiedCard. This function should be called when the application program has finished using the card.

Prototype

```
DWORD _stdcall PIL_CloseSpecifiedCard(DWORD CardNum);
```

Parameters:

CardNum - card number

Returns:

Zero for success, or non-zero error code.

Open All Cards (Visual C++)

Description

Locates and opens all installed Pickering cards. Once cards have been opened, other functions may then be used to access cards numbered 1 thru the value returned.

If cards have already been opened by the calling program, they are first closed - as though by PIL_CloseCards - and then re-opened.

If cards are currently opened by some other program they cannot be accessed and the function returns zero.

Prototype

```
DWORD _stdcall PIL_OpenCards(void);
```

Parameters:

None.

Returns:

The number of Pickering cards located and opened.

Note

When multiple Pickering cards are installed, the assignment of card numbers depends upon their relative physical locations in the system (or more accurately, on the order in which they are detected by the computer's operating system at boot time).

Open Specified Card (Visual C++)

Description

Opens the specified Pickering card, clearing all of its outputs. Once a card has been opened, other driver functions may then be used to access it.

If the card is currently opened by some other program it cannot be accessed and the function returns an error.

Prototype

DWORD _stdcall PIL_OpenSpecifiedCard(DWORD Bus, DWORD Slot, DWORD *CardNum);

Parameters:

Bus - the card's logical bus location

Slot - the card's logical slot location

CardNum - pointer to variable to receive the card's logical card_number

Returns:

Zero for success, or non-zero error code.

Note

The logical Bus and Slot values corresponding to a particular card are determined by system topology; values for cards that are operable by the Pilpxi driver can be discovered using PIL_FindFreeCards.

Information and Status

Information and Status

This section details the use in Visual C++ of functions for obtaining card and sub-unit information. Most of these functions are applicable to all card or sub-unit types.

Functions are provided for obtaining:

- The software driver version number: PIL_Version
- The number of unopened cards: PIL_CountFreeCards
- The bus and slot locations of unopened cards: PIL_FindFreeCards
- A card's identification string: PIL_CardId
- A card's logical bus and slot location: PIL_CardLoc
- A card's status flags: PIL_Status
- A string describing an error from the numeric code returned by a function: PIL_ErrorMessage
- A card's diagnostic information string: PIL_Diagnostic
- The numbers of input and output sub-units on a card: PIL_EumerateSubs
- Sub-unit information (numeric format): PIL_SubInfo
- Sub-unit information (string format): PIL_SubType
- An output sub-unit's closure limit value: PIL_ClosureLimit
- An output sub-unit's settling time value: PIL_SettleTime
- A sub-unit's status flags: PIL_SubStatus

Card ID (Visual C++)

Description

Obtains the identification string of the specified card. The string contains these elements:

<type code>,<serial number>,<revision code>.

The <revision code> value represents the hardware version of the unit - cards have no firmware on-board.

Prototype

DWORD _stdcall PIL_CardId(DWORD CardNum, CHAR *Str);

Parameters:

CardNum - card number

Str - pointer to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Card Location (Visual C++)

Description

Obtains the location of the specified card in terms of the logical PCI bus and slot number in which it is located.

These values can be cross-referenced to physical slot locations in a particular system.

Prototype

```
DWORD _stdcall PIL_CardLoc(DWORD CardNum, DWORD *Bus, DWORD *Slot);
```

Parameters:

CardNum - card number

Bus - pointer to variable to receive bus location

Slot - pointer to variable to receive slot location

Returns:

Zero for success, or non-zero error code.

Closure Limit (Visual C++)

Description

Obtains the maximum number of switches that may be activated simultaneously in the specified sub-unit. A single-channel multiplexer (MUX type) allows only one channel to be closed at any time. In some other models such as high-density matrix types a limit is imposed to prevent overheating; although it is possible to disable the limit for these types (see PIL_SetMode), doing so is not recommended.

Prototype

```
DWORD _stdcall PIL_ClosureLimit(DWORD CardNum, DWORD OutSub,
DWORD *Limit);
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Limit - pointer to the variable to receive the result

Returns:

Zero for success, or non-zero error code.

Count Free Cards (Visual C++)

Description

Obtains the number of installed cards that are operable by the Pilpxi driver but are not currently opened by it.

Prototype

```
DWORD _stdcall PIL_CountFreeCards(DWORD *NumCards);
```

Parameters:

NumCards - pointer to the variable to receive the result

Returns:

Zero for success, or non-zero error code.

Diagnostic (Visual C++)

Description

Obtains the diagnostic string of the specified card, giving expanded information on any fault conditons indicated by the PIL_Status value.

Prototype

DWORD _stdcall PIL_Diagnostic(DWORD CardNum, CHAR *Str);

Parameters:

CardNum - card number

Str - pointer to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Notes

The result string may include embedded newline characters, coded as the ASCII <linefeed> character ('\x0A').

The length of the result string will not exceed the value of the driver constant MAX_DIAG_LENGTH.

Warning

Formatting and content of the diagnostic string may change as enhanced diagnostic features are made available. It should therefore not be interpreted programatically.

Enumerate Sub-units (Visual C++)

Description

Obtains the numbers of input and output sub-units implemented on the specified card.

Prototype

DWORD _stdcall PIL_EnumerateSubs(DWORD CardNum, DWORD *InSubs, DWORD *OutSubs);

Parameters:

CardNum - card number

InSubs - pointer to variable to receive the number of INPUT sub-units

OutSubs - pointer to variable to receive the number of OUTPUT sub-units

Returns:

Zero for success, or non-zero error code.

Error Message (Visual C++)

Description

Obtains a string description of the error codes returned by other driver functions.

Prototype

```
DWORD _stdcall PIL_ErrorMessage(DWORD ErrorCode, CHAR *Str);
```

Parameters:

ErrorCode - the error code to be described

Str - pointer to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Find Free Cards (Visual C++)

Description

Obtains the logical bus and slot locations of installed cards that are operable by the Pilpxi driver and are currently unopened. These values are used with PIL_OpenSpecifiedCard.

Prototype

DWORD _stdcall PIL_FindFreeCards(DWORD NumCards, DWORD *BusList, DWORD *SlotList);

Parameters:

NumCards - the number of cards (maximum) for which information is to be obtained

BusList - pointer to the one-dimensional array (vector) to receive cards' bus location values

SlotList - pointer to the one-dimensional array (vector) to receive cards' slot location values

Returns:

Zero for success, or non-zero error code.

Notes

The bus and slot locations of the first card found are placed respectively in the least significant elements of the BusList and SlotList arrays. Successive elements contain the values for further cards.

If the value given for NumCards is less than the number of cards currently accessible, information is obtained only for the number of cards specified.

Warning

The arrays pointed to must have been assigned at least as many elements as the number of cards for which information is being requested or adjacent memory will be overwritten, causing data corruption and/or a program crash. The number of accessible cards can be discovered using PIL_CountFreeCards.

Settle Time (Visual C++)

Description

Obtains a sub-unit's settling time (or debounce period - the time taken for its switches to stabilise). By default, Pilpxi driver functions retain control during this period so that switches are guaranteed to have stabilised on completion. This mode of operation can be overridden if required - see PIL_SetMode.

Prototype

```
DWORD _stdcall PIL_SettleTime(DWORD CardNum, DWORD OutSub, DWORD *Time);
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Time - pointer to variable to receive the result (in microseconds)

Returns:

Zero for success, or non-zero error code.

Card Status (Visual C++)

Description

Obtains the current status flags for the specified card.

Prototype

DWORD _stdcall PIL_Status(DWORD CardNum);

Parameters:

CardNum - card number

Returns:

A value representing the card's status flags.

Status Bit Definitions

Status bits are as follows:

0x80000000 - STAT_NO_CARD (no card with specified number)

0x40000000 - STAT_WRONG_DRIVER (card requires newer driver)

0x20000000 - STAT_EEPROM_ERR (card EEPROM fault)

0x10000000 - STAT_DISABLED (card disabled)

0x04000000 - STAT_BUSY (card operations not completed)

0x02000000 - STAT_HW_FAULT (card hardware defect)

0x01000000 - STAT_PARITY_ERROR (PCIbus parity error)

0x00080000 - STAT_CARD_INACCESSIBLE (Card cannot be accessed - failed/removed/unpowered)

0x00040000 - STAT_UNCALIBRATED (one or more sub-units is uncalibrated)

0x00020000 - STAT_CALIBRATION_DUE (one or more sub-units is due for calibration)

0x00000000 - STAT_OK (card functional and stable)

Corresponding enumerated constants are provided in Pilpxi.h.

Notes

Certain status bits are relevant only for specific classes of sub-unit, or for those having particular characteristics.

At card level, STAT_BUSY indicates if any of a card's sub-units have not yet stabilised.

Diagnostic information on fault conditions indicated in the status value can be obtained using PIL_Diagnostic.

Related functions

PIL_SubStatus

Sub-unit Information (Visual C++)

Description

Obtains a description of a sub-unit, as numeric values.

Prototype

DWORD _stdcall PIL_SubInfo(DWORD CardNum, DWORD SubNum, BOOL Out, DWORD *TypeNum, DWORD *Rows, DWORD *Cols);

Parameters:

CardNum - card number

SubNum - sub-unit number

Out - sub-unit function: 0 for INPUT, 1 for OUTPUT

TypeNum - pointer to variable to receive type code

Rows - pointer to variable to receive row count

Cols - pointer to variable to receive column count

Returns:

Zero for success, or non-zero error code.

Results

Output sub-unit type codes are:

1 - TYPE_SW (uncommitted switch)

2 - TYPE_MUX (multiplexer single-channel)

3 - TYPE_MUXM (multiplexer, multi-channel)

4 - TYPE_MAT (matrix - LF)

5 - TYPE_MATR (matrix - RF)

6 - TYPE_DIG (digital outputs)

7 - TYPE_RES (programmable resistor)

8 - TYPE_ATTEN (programmable RF attenuator)

9 - TYPE_PSUDC (DC power supply)

10 - TYPE_BATT (Battery Simulator)

Corresponding enumerated constants are provided in Pilpxi.h.

Input sub-unit type codes are:

1 - INPUT

Row and column values give the dimensions of the sub-unit. For all types other than matrices the column value contains the significant dimension: their row value is always '1'.

Note

Some sub-unit types are supported by functions providing alternate and/or more detailed information. These include:

TYPE_ATTEN - PIL_AttenInfo

TYPE_PSUDC - PIL_PsuInfo

Sub-unit Status (Visual C++)

Description

Obtains the current status flags for the specified output sub-unit. Status bits associated with significant card-level conditions are also returned.

Prototype

DWORD _stdcall PIL_SubStatus(DWORD CardNum, DWORD SubNum);

Parameters:

CardNum - card number

SubNum - sub-unit number

Returns:

A value representing the sub-unit's status flags.

Status Bit Definitions

Status bits are as follows:

0x80000000 - STAT_NO_CARD (no card with specified number)

0x40000000 - STAT_WRONG_DRIVER (card requires newer driver)

0x20000000 - STAT_EEPROM_ERR (card EEPROM fault)

0x10000000 - STAT_DISABLED (card disabled)

0x08000000 - STAT_NO_SUB (no sub-unit with specified number)

0x04000000 - STAT_BUSY (sub-unit operations not completed)

0x02000000 - STAT_HW_FAULT (card hardware defect)

0x01000000 - STAT_PARITY_ERROR (PCIbus parity error)

0x00800000 - STAT_PSU_INHIBITED (power supply output is disabled - by software)

0x00400000 - STAT_PSU_SHUTDOWN (power supply output is shutdown - due to overload)

0x00200000 - STAT_PSU_CURRENT_LIMIT (power supply is operating in current-limited mode)

0x00100000 - STAT_CORRUPTED (sub-unit logical state is corrupted)

0x00080000 - STAT_CARD_INACCESSIBLE (Card cannot be accessed - failed/removed/unpowered)

0x00040000 - STAT_UNCALIBRATED (sub-unit is uncalibrated)

0x00020000 - STAT_CALIBRATION_DUE (sub-unit is due for calibration)

0x00000000 - STAT_OK (sub-unit functional and stable)

Corresponding enumerated constants are provided in Pilpxi.h.

Notes

Certain status bits are relevant only for specific classes of sub-unit, or for those having particular characteristics.

Diagnostic information on fault conditions indicated in the status value can be obtained using PIL_Diagnostic.

Related functions

PIL_Status

Sub-unit Type (Visual C++)

Description

Obtains a description of a sub-unit, as a text string.

Prototype

DWORD _stdcall PIL_SubType(DWORD CardNum, DWORD SubNum, BOOL Out, CHAR *Str);

Parameters:

CardNum - card number

SubNum - sub-unit number

Out - sub-unit function: 0 for INPUT, 1 for OUTPUT

Str - pointer to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Type string	**Description**
INPUT(<size>)	Digital inputs
SWITCH(<size>)	Uncommitted switches
MUX(<size>)	Multiplexer, single-channel only
MUXM(<size>)	Multiplexer, multi-channel
MATRIX(<columns>X<rows>)	Matrix, LF
MATRIXR(<columns>X<rows>)	Matrix, RF
DIGITAL(<size>)	Digital Outputs
RES(<number of resistors in chain>)	Programmable resistor
ATTEN(<number of pads>)	Programmable RF attenuator
PSUDC(0)	DC Power Supply
BATT(<Voltage DAC resolution, bits>)	Battery Simulator

Note

Some sub-unit types are supported by functions providing more detailed information. These include:

ATTEN - PIL_AttenType

PSUDC - PIL_PsuType

Version (Visual C++)

Description

Obtains the driver version code.

Prototype

DWORD _stdcall PIL_Version(void);

Parameters:

None.

Returns:

The driver version code, multiplied by 100 (i.e. a value of 100 represents version 1.00)

Switching and General Purpose Output

Switching and General Purpose Output

This section details the use in Visual C++ of functions that are applicable to most output sub-unit types.

Note that although these functions may be used with them, some sub-unit types - for example matrix and programmable RF attenuator - are also served by specific functions offering more straightforward control.

Functions are provided to:

- Clear all output channels of all open Pickering cards: PIL_ClearAll
- Clear all output channels of a single Pickering card: PIL_ClearCard
- Clear all output channels of a sub-unit: PIL_ClearSub
- Open or close a single output channel: PIL_OpBit
- Set a sub-unit's output pattern: PIL_WriteSub, (PIL_WriteSubArray)
- Obtain the state of a single output channel: PIL_ViewBit
- Obtain a sub-unit's output pattern: PIL_ViewSub, (PIL_ViewSubArray)

Clear All (Visual C++)

Description

Clears (de-energises or sets to logic '0') all outputs of all sub-units of every open Pickering card.

Prototype

```
DWORD _stdcall PIL_ClearAll(void);
```

Parameters:

None.

Returns:

Zero for success, or non-zero error code.

Clear Card (Visual C++)

Description

Clears (de-energises or sets to logic '0') all outputs of all sub-units of the specified Pickering card.

Prototype

```
DWORD _stdcall PIL_ClearCard(DWORD CardNum);
```

Parameters:

CardNum - card number

Returns:

Zero for success, or non-zero error code.

Clear Sub-unit (Visual C++)

Description

Clears (de-energises or sets to logic '0') all outputs of a sub-unit.

Prototype

DWORD _stdcall PIL_ClearSub(DWORD CardNum, DWORD OutSub);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Returns:

Zero for success, or non-zero error code.

Operate Bit (Visual C++)

Description

Operate a single output channel or bit.

Note that in the case of a single-channel multiplexer (MUX type) any existing channel closure will be cleared automatically prior to selecting the new channel.

Note that PIL_OpCrosspoint allows more straightforward use of row/column co-ordinates with matrix sub-units.

Prototype

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

Parameters:

CardNum - card number

OutSub - output sub-unit number

BitNum - output bit number

Action - 1 to energise, 0 to de-energise

Returns:

Zero for success, or non-zero error code.

View Bit (Visual C++)

Description

Obtains the state of an individual output.

Prototype

DWORD _stdcall PIL_ViewBit(DWORD CardNum, DWORD OutSub, DWORD BitNum, BOOL *State);

Parameters:

CardNum - card number

OutSub - output sub-unit number

BitNum - output bit number

State - pointer to variable to receive the result (0 = OFF or logic '0', 1 = ON or logic '1')

Returns:

Zero for success, or non-zero error code.

View Sub-unit (Visual C++)

Description

Obtains the state of all outputs of a sub-unit. The result fills the number of least significant bits corresponding to the size of the sub-unit.

Prototype

```
DWORD _stdcall PIL_ViewSub(DWORD CardNum, DWORD OutSub, DWORD *Data);
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - pointer to the one-dimensional array (vector) to receive the result

Returns:

Zero for success, or non-zero error code.

Note

For a Matrix sub-unit, the result is folded into the vector on its row-axis: see Data Formats.

Warning

The data array pointed to must contain sufficient bits to hold the bit-pattern for the specified sub-unit, or adjacent memory will be overwritten, causing data corruption and/or a program crash.

Example Code

See the description of PIL_WriteSub for example code using an array-based function.

View Sub-unit - SafeArray (Visual C++)

Description

Obtains the state of all outputs of a sub-unit. The result fills the number of least significant bits corresponding to the size of the sub-unit.

Prototype

```
DWORD _stdcall PIL_ViewSubArray(DWORD CardNum, DWORD OutSub,
LPSAFEARRAY FAR* Data);
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - pointer to the one-dimensional SAFEARRAY structure to receive the result

Returns:

Zero for success, or non-zero error code.

Notes

Although mainly intended to provide robust array handling in Visual Basic, this function is also usable in Visual C++.

Function PIL_ViewSub is an equivalent function employing a 'standard' C data array.

For a Matrix sub-unit, the result is folded into the SAFEARRAY on its row-axis: see Data Formats.

Write Sub-unit (Visual C++)

Description

Sets all outputs of a sub-unit to the supplied bit-pattern. The number of least significant bits corresponding to the size of the sub-unit are written.

Prototype

DWORD _stdcall PIL_WriteSub(DWORD CardNum, DWORD OutSub, DWORD *Data);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - pointer to the one-dimensional array (vector) containing the bit-pattern to be written

Returns:

Zero for success, or non-zero error code.

Note

For a Matrix sub-unit, the data is folded into the vector on its row-axis: see Data Formats.

Warning

The data array pointed to must contain sufficient bits to represent the bit-pattern for the specified sub-unit, or undefined data will be written to the more significant bits.

Example Code

For clarity, this example omits initialising the variables CardNum, OutSub etc. and does no error-checking.

```
/* Dimension a DWORD data array to contain the number of bits

  necessary to represent the sub-unit (e.g. 2 longwords

  supports sub-units having upto 64 switches) */

DWORD Data[2]; /* Value specifies the number of array elements */



/* Data[0] bit 0 represents switch #1
```

```
  Data[0] bit 1 represents switch #2

  ... etc.

  Data[0] bit 31 represents switch #32

  Data[1] bit 0 represents switch #33

  ... etc. */


/* Setup array data to turn on switches 3, 33 and output to the card
*/

Data[0] = 0x00000004UL; /* set DWORD 0 bit 2 (switch 3) */

Data[1] = 0x00000001UL; /* set DWORD 1 bit 0 (switch 33) */

Result = PIL_WriteSub(CardNum, OutSub, Data);


/* Add switch 4 to the array and output to the card */

Data[0] |= 0x00000008UL; /* set DWORD 0 bit 3 (switch 4) */

Result = PIL_WriteSub(CardNum, OutSub, Data);

/* ... now have switches 3, 4, 33 energised */


/* Delete switch 33 from the array and output to the card */

Data[1] &= 0xFFFFFFFEUL; /* clear DWORD 1 bit 0 (switch 33) */

Result = PIL_WriteSub(CardNum, OutSub, Data);

/* ... leaving switches 3 and 4 energised */
```

Write Sub-unit - SafeArray (Visual C++)

Description

Sets all outputs of a sub-unit to the supplied bit-pattern. The number of least significant bits corresponding to the size of the sub-unit are written.

Prototype

DWORD _stdcall PIL_WriteSubArray(DWORD CardNum, DWORD OutSub, LPSAFEARRAY FAR* Data);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - pointer to the one-dimensional SAFEARRAY structure containing the bit-pattern to be written

Returns:

Zero for success, or non-zero error code.

Notes

Although mainly intended to provide robust array handling in Visual Basic, this function is also usable in Visual C++.

Function PIL_WriteSub is an equivalent function employing a 'standard' C data array.

For a Matrix sub-unit, the data is folded into the SAFEARRAY on its row-axis: see Data Formats.

Specialised Switching

Specialised Switching

This section details the use in Visual C++ of functions specific to particular types of switching sub-unit (uncommitted switches, multiplexer, matrix and digital output types).

Matrix operations

- Open or close a single matrix crosspoint: PIL_OpCrosspoint
- Obtain the state of a single matrix crosspoint: PIL_ViewCrosspoint

Individual switch operations, complex matrix sub-units

- Obtain/set the state of an individual switch: PIL_OpSwitch

Attribute information, complex matrix sub-units

- Obtain sub-unit attribute values: PIL_SubAttribute

Operate Crosspoint (Visual C++)

Description

Operate a single matrix crosspoint.

Prototype

DWORD _stdcall PIL_OpCrosspoint(DWORD CardNum, DWORD OutSub, DWORD Row, DWORD Column, BOOL Action);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Row - crosspoint row (Y) location

Column - crosspoint column (X) location

Action - 1 to energise, 0 to de-energise

Returns:

Zero for success, or non-zero error code.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized bit-number method employed by PIL_OpBit. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

Related Matrix Functions

PIL_ViewCrosspoint

PIL_MaskCrosspoint

PIL_ViewMaskCrosspoint

Operate switch (Visual C++)

Description

This function obtains, and optionally sets, the state of a switch. It allows explicit access to the individual switches making up a sub-unit, in types where their operation is normally handled automatically by the driver. The main purpose of this is in implementing fault diagnostic programs for such types; it can also be used where normal automated behaviour does not suit an application.

Prototype

DWORD _stdcall PIL_OpSwitch(DWORD CardNum, DWORD OutSub, DWORD SwitchFunc, DWORD SegNum, DWORD SwitchNum, DWORD SubSwitch, DWORD SwitchAction, BOOL *State);

Parameters:

CardNum - card number

OutSub - sub-unit number

SwitchFunc - code indicating the functional group of the switch, see below

SegNum - the segment location of the switch

SwitchNum - the number of the switch in its functional group (unity-based)

SubSwitch - the number of the subswitch to operate (unity-based)

SwitchAction - code indicating the action to be performed, see below

State - pointer to variable to receive the state of the switch (after performing any action)

Returns:

Zero for success, or non-zero error code.

Applicable sub-unit types

This function is only usable with matrix sub-units having auto-isolation and/or auto-loopthru features. For further information see: segmented matrix, unsegmented matrix.

SwitchFunc value

A value indicating the functional group of the switch to be accessed.

Value	Ident	Function
0	SW_FUNC_CHANNEL	A channel (matrix crosspoint) switch
1	SW_FUNC_X_ISO	A matrix X-isolation switch

2	SW_FUNC_Y_ISO	A matrix Y-isolation switch
3	SW_FUNC_X_LOOPTHRU	A matrix X-loopthru switch
4	SW_FUNC_Y_LOOPTHRU	A matrix Y-loopthru switch

SegNum value

The segment location of the switch. The numbers and sizes of segments on each matrix axis can be obtained using PIL_SubAttribute.

In an unsegmented matrix, use SegNum = 1.

In a segmented matrix, segment numbers for crosspoint and isolation switches are determined logically.

SwitchNum value

The number of the switch in its functional group (unity-based).

For channel (crosspoint) switches, the switch number can be either:

- if SegNum is zero, the global channel number of the switch (see output bit number)
- if SegNum is non-zero, the segment-local number of the switch, calculated in a similar way to the above

SubSwitch value

The number of the subswitch to operate (unity-based). This parameter caters for a situation in which a logical channel, isolation or loopthru switch is served by more than one physical relay (as for example when 2-pole operation is implemented using independently-driven single-pole relays).

The numbers of subswitches for each functional group can be obtained using PIL_SubAttribute.

SwitchAction value

A code indicating the action to be performed.

Value	Ident	Function
0	SW_ACT_NONE	No switch change - just set State result
1	SW_ACT_OPEN	Open switch

2	SW_ACT_CLOSE	Close switch

Loopthru switches

Loopthru switches are initialised by the driver to a **closed** state, which may mean that they are either energised or de-energised depending upon their type. In normal automated operation loopthru switches open when any crosspoint on their associated line is closed. Actions SW_ACT_CLOSE and SW_ACT_OPEN close or open loopthru switch contacts as their names imply.

Operational considerations

This function can be used to alter a pre-existing switch state in a sub-unit, set up by fuctions such as PIL_OpBit or PIL_WriteSub. However once the state of any switch is changed by PIL_OpSwitch the logical state of the sub-unit is considered to have been destroyed. This condition is flagged in the result of PIL_SubStatus (bit STAT_CORRUPTED). Subsequent attempts to operate it using 'ordinary' switch functions such as PIL_OpBit, PIL_ViewBit etc. will fail (result ER_STATE_CORRUPT). Normal operation can be restored by clearing the sub-unit using PIL_ClearSub, PIL_ClearCard or PIL_ClearAll.

View Crosspoint (Visual C++)

Description

Obtains the state of an individual matrix crosspoint.

Prototype

DWORD _stdcall PIL_ViewCrosspoint(DWORD CardNum, DWORD OutSub, DWORD Row, DWORD Column, BOOL *State);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Row - crosspoint row (Y) location

Column - crosspoint column (X) location

State - pointer to variable to receive the result (0 = OFF, 1 = ON)

Returns:

Zero for success, or non-zero error code.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized bit-number method employed by PIL_ViewBit. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

Sub-unit Attribute (Visual C++)

Description

Obtains the value of a sub-unit attribute. These values facilitate operation using PIL_OpSwitch.

Prototype

DWORD _stdcall PIL_SubAttribute(DWORD CardNum, DWORD SubNum, BOOL Out, DWORD AttrCode, DWORD *AttrValue);

Parameters:

CardNum - card number

SubNum - sub-unit number

Out - sub-unit function: 0 for INPUT, 1 for OUTPUT

AttrCode - a value indicating the sub-unit attribute to be queried, see below

AttrValue - pointer to variable to receive the attribute's value

Returns:

Zero for success, or non-zero error code.

AttrCode values

Value	Ident	Function
1	SUB_ATTR_CHANNEL_SUBSWITCHES	Gets number of subswitches per logical channel (matrix crosspoint)
2	SUB_ATTR_X_ISO_SUBSWITCHES	Gets number of subswitches per logical X-isolator
3	SUB_ATTR_Y_ISO_SUBSWITCHES	Gets number of subswitches per logical Y-isolator
4	SUB_ATTR_X_LOOPTHRU_SUBSWITCHES	Gets number of subswitches per logical X-loopthru
5	SUB_ATTR_Y_LOOPTHRU_SUBSWITCHES	Gets number of subswitches per logical Y-loopthru
0x100	SUB_ATTR_NUM_X_SEGMENTS	Gets number of X-axis segments
0x101	SUB_ATTR_X_SEGMENT01_SIZE	Gets size of X-axis segment 1
0x102	SUB_ATTR_X_SEGMENT02_SIZE	Gets size of X-axis segment 2
0x103	SUB_ATTR_X_SEGMENT03_SIZE	Gets size of X-axis segment 3
0x104	SUB_ATTR_X_SEGMENT04_SIZE	Gets size of X-axis segment 4
0x105	SUB_ATTR_X_SEGMENT05_SIZE	Gets size of X-axis segment 5
0x106	SUB_ATTR_X_SEGMENT06_SIZE	Gets size of X-axis segment 6

0x107	SUB_ATTR_X_SEGMENT07_SIZE	Gets size of X-axis segment 7
0x108	SUB_ATTR_X_SEGMENT08_SIZE	Gets size of X-axis segment 8
0x109	SUB_ATTR_X_SEGMENT09_SIZE	Gets size of X-axis segment 9
0x10A	SUB_ATTR_X_SEGMENT10_SIZE	Gets size of X-axis segment 10
0x10B	SUB_ATTR_X_SEGMENT11_SIZE	Gets size of X-axis segment 11
0x10C	SUB_ATTR_X_SEGMENT12_SIZE	Gets size of X-axis segment 12
0x200	SUB_ATTR_NUM_Y_SEGMENTS	Gets number of Y-axis segments
0x201	SUB_ATTR_Y_SEGMENT01_SIZE	Gets size of y-axis segment 1
0x202	SUB_ATTR_Y_SEGMENT02_SIZE	Gets size of y-axis segment 2

Switch Masking

Switch Masking

This section details the use in Visual C++ of switch masking functions.

Masking permits disabling operation of chosen switch channels by the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions. These functions report error ER_OUTPUT_MASKED if an attempt is made to activate a masked channel.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Masking functions, all switching sub-unit types

- Clear a sub-unit's mask: PIL_ClearMask
- Mask or unmask a single output channel: PIL_MaskBit
- Set a sub-unit's mask pattern: PIL_WriteMask, (PIL_WriteMaskArray)
- Obtain the mask state of a single output channel: PIL_ViewMaskBit
- Obtain a sub-unit's mask pattern: PIL_ViewMask, (PIL_ViewMaskArray)

Masking functions, matrix sub-units

- Mask or unmask a single matrix crosspoint: PIL_MaskCrosspoint
- Obtain the mask state of a single matrix crosspoint: PIL_ViewMaskCrosspoint

Note

Masking only allows channels to be disabled in the OFF state; applying a mask to a channel that is already turned ON forces it OFF.

Clear Mask (Visual C++)

Description

Clears a sub-unit's switch mask, enabling operation of all outputs by the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

Prototype

```
DWORD _stdcall PIL_ClearMask(DWORD CardNum, DWORD OutSub);
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Returns:

Zero for success, or non-zero error code.

Mask Bit (Visual C++)

Description

Mask or unmask a single output bit.

Masking disables the corresponding switch for the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits. Note that PIL_MaskCrosspoint allows more straightforward use of row/column co-ordinates with matrices.

Prototype

```
DWORD _stdcall PIL_MaskBit(DWORD CardNum, DWORD OutSub, DWORD BitNum, BOOL Action);
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

BitNum - output bit number

Action - 1 to mask, 0 to unmask

Returns:

Zero for success, or non-zero error code.

Note

Certain single-channel multiplexer (MUX type) sub-units have a default channel (that is, a channel that is connected when the sub-unit is in a 'cleared' state). This channel cannot be masked, and error ER_ILLEGAL_MASK is given if an attempt is made to mask it.

Mask Crosspoint (Visual C++)

Description

Mask or unmask a single matrix crosspoint.

Masking disables the corresponding switch for the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

This facility can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Prototype

```
DWORD _stdcall PIL_MaskCrosspoint(DWORD CardNum, DWORD OutSub,
DWORD Row, DWORD Column, BOOL Action);
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Row - crosspoint row (Y) location

Column - crosspoint column (X) location

Action - 1 to mask, 0 to unmask

Returns:

Zero for success, or non-zero error code.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized bit-number method employed by PIL_MaskBit. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

View Mask (Visual C++)

Description

Obtains the switch mask of a sub-unit. The result fills the number of least significant bits corresponding to the size of the sub-unit.

Prototype

DWORD _stdcall PIL_ViewMask(DWORD CardNum, DWORD OutSub, DWORD *Data);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - pointer to the one-dimensional array (vector) to receive the result

Returns:

Zero for success, or non-zero error code.

Note

For a Matrix sub-unit, the result is folded into the vector on its row-axis: see Data Formats.

Warning

The data array pointed to must contain sufficient bits to hold the bit-pattern for the specified sub-unit, or adjacent memory will be overwritten, causing data corruption and/or a program crash.

Example Code

See the description of PIL_WriteSub for example code using an array-based function.

View Mask - SafeArray (Visual C++)

Description

Obtains the switch mask of a sub-unit. The result fills the number of least significant bits corresponding to the size of the sub-unit.

Prototype

DWORD _stdcall PIL_ViewMaskArray(DWORD CardNum, DWORD OutSub, LPSAFEARRAY FAR* Data);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - pointer to the one-dimensional SAFEARRAY structure to receive the result

Returns:

Zero for success, or non-zero error code.

Note

Although mainly intended to provide robust array handling in Visual Basic, this function is also usable in Visual C++.

Function PIL_ViewMask is an equivalent function employing a 'standard' C data array.

For a Matrix sub-unit, the result is folded into the SAFEARRAY on its row-axis: see Data Formats.

View Mask Bit (Visual C++)

Description

Obtains the state of an individual output's mask.

Prototype

```
DWORD _stdcall PIL_ViewMaskBit(DWORD CardNum, DWORD OutSub,
DWORD BitNum, BOOL *State);
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

BitNum - output bit number

State - pointer to variable to receive the result (0 = unmasked, 1 = masked)

Returns:

Zero for success, or non-zero error code.

View Mask Crosspoint (Visual C++)

Description

Obtains the state of an individual matrix crosspoint's mask.

Prototype

DWORD _stdcall PIL_ViewMaskCrosspoint(DWORD CardNum, DWORD OutSub, DWORD Row, DWORD Column, BOOL *State);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Row - crosspoint row (Y) location

Column - crosspoint column (X) location

State - pointer to variable to receive the result (0 = unmasked, 1 = masked)

Returns:

Zero for success, or non-zero error code.

Note

This function supports matrix operation using row/column co-ordinates in place of the linearized bit-number method employed by PIL_ViewMaskBit. It offers more straightforward matrix operation, and avoids the need for re-coding if a matrix card is replaced by one having different dimensions.

Write Mask (Visual C++)

Description

Sets a sub-unit's switch mask to the supplied bit-pattern. The number of least significant bits corresponding to the size of the sub-unit are written into the mask. A '1' bit in the mask disables the corresponding switch for the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Prototype

```
DWORD _stdcall PIL_WriteMask(DWORD CardNum, DWORD OutSub, DWORD *Data);
```

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - pointer to the one-dimensional array (vector) containing the mask pattern to be set

Returns:

Zero for success, or non-zero error code.

Notes

For a Matrix sub-unit, the mask data is folded into the vector on its row-axis: see Data Formats.

Certain single-channel multiplexer (MUX type) sub-units have a default channel (that is, a channel that is connected when the sub-unit is in a 'cleared' state). This channel cannot be masked, and error ER_ILLEGAL_MASK is given if an attempt is made to mask it.

Warning

The data array pointed to must contain sufficient bits to represent the mask pattern for the specified sub-unit, or undefined data will be written to the more significant bits.

Example Code

See the description of PIL_WriteSub for example code using an array-based function.

Write Mask - SafeArray (Visual C++)

Description

Sets a sub-unit's switch mask to the supplied bit-pattern. The number of least significant bits corresponding to the size of the sub-unit are written into the mask. A '1' bit in the mask disables the corresponding switch for the PIL_OpBit, PIL_OpCrosspoint, PIL_WriteSub and PIL_WriteSubArray functions.

This facility is particularly useful for matrix sub-units, where it can be used to guard against programming errors that could otherwise result in damage to matrix switches or external circuits.

Prototype

DWORD _stdcall PIL_WriteMaskArray(DWORD CardNum, DWORD OutSub, LPSAFEARRAY FAR* Data);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Data - pointer to the one-dimensional SAFEARRAY structure containing the mask pattern to be set

Returns:

Zero for success, or non-zero error code.

Notes

Although mainly intended to provide robust array handling in Visual Basic, this function is also usable in Visual C++.

Function PIL_WriteMask is an equivalent function employing a 'standard' C data array.

For a Matrix sub-unit, the mask data is folded into the SAFEARRAY on its row-axis: see Data Formats.

Certain single-channel multiplexer (MUX type) sub-units have a default channel (that is, a channel that is connected when the sub-unit is in a 'cleared' state). This channel cannot be masked, and error ER_ILLEGAL_MASK is given if an attempt is made to mask it.

Input

Input

This section details the use in Visual C++ of functions specific to input sub-units.

Specific functions are provided to:

- Obtain the state of a single input: PIL_ReadBit
- Obtain a sub-unit's input pattern: PIL_ReadSub

Read Bit (Visual C++)

Description

Obtains the state of an individual input.

Prototype

DWORD _stdcall PIL_ReadBit(DWORD CardNum, DWORD InSub, DWORD BitNum, BOOL *State);

Parameters:

CardNum - card number

InSub - input sub-unit number

BitNum - input bit number

State - pointer to variable to receive the result (0 = logic '0', 1 = logic '1')

Returns:

Zero for success, or non-zero error code.

Read Sub-unit (Visual C++)

Description

Obtains the current state of all inputs of a sub-unit.

Prototype

DWORD _stdcall PIL_ReadSub(DWORD CardNum, DWORD InSub, DWORD *Data);

Parameters:

CardNum - card number

InSub - input sub-unit number

Data - pointer to variable to receive result

Returns:

Zero for success, or non-zero error code.

Warning

The data array pointed to must contain sufficient bits to hold the bit-pattern for the specified sub-unit, or adjacent memory will be overwritten, causing data corruption and/or a program crash.

Example Code

See the description of PIL_WriteSub for example code using an array-based function.

Calibration

Calibration

This section details the use in Visual C++ of functions associated with storing calibration values in a card's non-volatile memory. This facility is only available for certain sub-unit types, such as programmable resistors.

Specific functions are provided to:

- Retrieve an integer calibration value from non-volatile memory: PIL_ReadCal
- Store an integer calibration value in non-volatile memory: PIL_WriteCal
- Retrieve floating-point calibration value(s) from non-volatile memory: PIL_ReadCalFP
- Store floating-point calibration value(s) in non-volatile memory: PIL_WriteCalFP
- Retrieve a sub-unit's calibration date from non-volatile memory: PIL_ReadCalDate
- Store a sub-unit's calibration date in non-volatile memory: PIL_WriteCalDate
- Set a calibration point: PIL_SetCalPoint

Read Integer Calibration Value (Visual C++)

Description

Reads an integer calibration value from on-card EEPROM.

Prototype

DWORD _stdcall PIL_ReadCal(DWORD CardNum, DWORD OutSub, DWORD Idx, DWORD *Data);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Idx - calibration value index number (see below)

Data - pointer to variable to receive result

Returns:

Zero for success, or non-zero error code.

Notes

This function is usable only with sub-units that support integer calibration data.

In simple programmable resistor models such as:

40-280

40-281

40-282

40-290

40-291

40-295

40-296

50-295

the Pilpxi driver places no interpretation on the stored value - an application program can utilise it in any way it wishes.

In some other models, including:

41-735-001

41-752-001

stored values are utilised by specific Pilpxi driver functions, and they should only be overwritten by an appropriate calibration utility.

For programmable resistors supporting this function the valid range of Idx values corresponds to the number of bits, i.e. to the range of output bit number values. A 16-bit resistor sub-unit typically providess 16 x 16-bit values.

The storage capacity of other types supporting this feature is determined by their functionality.

Related functions

PIL_WriteCal

Read Calibration Date (Visual C++)

Description

Reads a sub-unit's calibration date and interval from on-card EEPROM.

Prototype

DWORD _stdcall PIL_ReadCalDate(DWORD CardNum, DWORD OutSub, DWORD Store, DWORD *Year, DWORD *Day, DWORD *Interval);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Store - value indicating which store to access (see below)

Year - pointer to variable to receive the year of calibration

Day - pointer to variable to receive the day in the year of calibration

Interval - pointer to variable to receive the calibration interval (in days)

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data; it can be used to discover when the sub-unit was last calibrated, and when recalibration will become due. Bit STAT_CALIBRATION_DUE in the result of PIL_Status or PIL_SubStatus indicates the need for recalibration.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "Store" Parameter	Ident	Function
0	CAL_STORE_USER	Access user calibration store
1	CAL_STORE_FACTORY	Access factory calibration store

Related functions

PIL_WriteCalDate

Read Floating-point Calibration Value (Visual C++)

Description

Reads one or more floating-point calibration values from on-card EEPROM.

Prototype

DWORD _stdcall PIL_ReadCalFP(DWORD CardNum, DWORD OutSub, DWORD Store, DWORD Offset, DWORD NumValues, double *Data);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Store - value indicating which store to access (see below)

Offset - the offset in the sub-unit's calibration store at which to start

NumValues - the number of values to be read

Data - pointer to array to receive result

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as PIL_ResSetResistance. The number of values stored and their purpose is specific to the target sub-unit.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "Store" Parameter	**Ident**	**Function**
0	CAL_STORE_USER	Access user calibration store
1	CAL_STORE_FACTORY	Access factory calibration store

Related functions

PIL_WriteCalFP

Set Calibration Point (Visual C++)

Description

Sets a sub-unit to a state corresponding to one of its defined calibration points.

Prototype

DWORD _stdcall PIL_SetCalPoint(DWORD CardNum, DWORD OutSub, DWORD Idx);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Idx - the index number of the calibration point (see below)

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as PIL_ResSetResistance. The number of calibration points supported is specific to the target sub-unit.

The Idx value used by this function corresponds directly to the offset in the sub-unit's calibration store at which the value is to be stored and retrieved, using PIL_WriteCalFP and PIL_ReadCalFP.

WARNING

Selection of a calibration point causes the sub-unit to change state; the resulting state may be outside its normally desired range of operation. On completion of a calibration sequence, PIL_ResSetResistance can be used to normalise the setting.

Write Integer Calibration Value (Visual C++)

Description

Writes an integer calibration value to on-card EEPROM.

Prototype

DWORD _stdcall PIL_WriteCal(DWORD CardNum, DWORD OutSub, DWORD Idx, DWORD Data);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Idx - calibration value index number (see below)

Data - the value to be written

Returns:

Zero for success, or non-zero error code.

Notes

This function is usable only with sub-units that support integer calibration data.

In simple programmable resistor models such as:

40-280

40-281

40-282

40-290

40-291

40-295

40-296

50-295

the Pilpxi driver places no interpretation on the stored value - an application program can utilise it in any way it wishes.

In some other models, including:

41-735-001

41-752-001

stored values are utilised by specific Pilpxi driver functions, and they should only be overwritten by an appropriate calibration utility.

The number of bits actually stored is specific to the target sub-unit - any redundant high-order bits of the supplied Data value are ignored.

For programmable resistors supporting this function the valid range of Idx values corresponds to the number of bits, i.e. to the range of output bit number values. A 16-bit resistor sub-unit typically provides 16 x 16-bit values.

The storage capacity of other types supporting this feature is determined by their functionality.

Related functions

PIL_ReadCal

Write Calibration Date (Visual C++)

Description

Writes a sub-unit's calibration date and interval into on-card EEPROM. Date information is obtained from the current system date.

Prototype

DWORD _stdcall PIL_WriteCalDate(DWORD CardNum, DWORD OutSub, DWORD Store, DWORD Interval);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Store - value indicating which store to access (see below)

Interval - the desired calibration interval (in days)

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as PIL_ResSetResistance. The number of values stored and their purpose is specific to the target sub-unit.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "Store" Parameter	Ident	Function
0	CAL_STORE_USER	Access user calibration store
1	CAL_STORE_FACTORY	Access factory calibration store

Related functions

PIL_ReadCalDate

Write Floating-point Calibration Value (Visual C++)

Description

Writes one or more floating-point calibration values into on-card EEPROM.

Prototype

DWORD _stdcall PIL_WriteCalFP(DWORD CardNum, DWORD OutSub, DWORD Store, DWORD Offset, DWORD NumValues, double *Data);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Store - value indicating which store to access (see below)

Offset - the offset in the sub-unit's calibration store at which to start

NumValues - the number of values to be written

Data - pointer to array containing values to write

Returns:

Zero for success, or non-zero error code.

Notes

This function is only applicable to sub-units that support floating-point calibration data, and would normally be used by a calibration tool for the model concerned. Floating-point calibration data is utilised by functions such as PIL_ResSetResistance. The number of values stored and their purpose is specific to the target sub-unit.

Some sub-units support dual calibration stores, known as "user" and "factory" stores. The user store holds the active calibration data, while the factory store holds a backup calibration that can be reverted to in the event of the user store contents becoming invalid.

Value of "Store" Parameter	**Ident**	**Function**
0	CAL_STORE_USER	Access user calibration store
1	CAL_STORE_FACTORY	Access factory calibration store

WARNING

Writing new values will affect the sub-unit's calibration.

Related functions

PIL_ReadCalFP

Programmable Resistor

Programmable Resistor

This section details the use in Visual C++ of functions specific to programmable resistor sub-units.

Detailed information about a programmable resistor sub-unit, if available, can be obtained using function PIL_ResInfo.

Precision models

Precision programmable resistor models such as 40-260-001 are supported by functions:

- PIL_ResGetResistance
- PIL_ResSetResistance

which allow chosen resistance values to be set.

Simple models

In models not supported by the above functions general purpose output functions such as PIL_WriteSub must be used to program resistance values by setting bit-patterns explicitly.

Models 40-280, 40-281 and 40-282 are configured as simple resistor/switch arrays and programming should be straightforward.

In models employing a series resistor chain - such as 40-290, 40-291 and 40-295 - each of a card's programmable resistors is implemented as a separate logical sub-unit constructed from a series chain of individual fixed resistor elements, each element having an associated shorting switch. In the cleared state all switches are open, giving the programmable resistor its maximum value. A nominal value of zero ohms is obtained by turning all switches ON; other values by turning on an appropriate pattern of switches.

In standard models the individual fixed resistors are arranged in a binary sequence, the least significant bit of the least significant element in the array passed to PIL_WriteSub corresponding to the lowest value resistor element. For example, in a standard model 40-290 16-bit resistor of 32768 ohms:

Data[0] bit 0 (value 0x0001) corresponds to the 0R5 resistor element

Data[0] bit 1 (value 0x0002) corresponds to the 1R0 resistor element

thru...

Data[0] bit 15 (value 0x8000) corresponds to the 16384R resistor element

Setting a nominal value of 68 ohms (= 64 + 4 ohms) therefore requires Data[0] set to 0xFF77 (the inverse of the binary pattern 0000 0000 1000 1000).

Special models may have some other arrangement, and may also include a fixed offset resistor that is permanently in circuit.

Non-volatile storage of calibration values is supported through the functions PIL_ReadCal and PIL_WriteCal.

See the application note on Simple Programmable Resistor Cards.

Summary of functions for normal operation of "Programmable Resistor" cards

Model(s)	Class	Functions
40-260-001	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
40-260-999	Precision	PIL_WriteSub
		PIL_ViewSub
40-261	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
40-262	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
40-265	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
40-280, 40-281, 40-282	Simple	PIL_OpBit
		PIL_ViewBit
		PIL_WriteSub
		PIL_ViewSub
		PIL_ReadCal
		PIL_WriteCal
40-290, 40-291	Simple	PIL_WriteSub
		PIL_ViewSub
		PIL_ReadCal
		PIL_WriteCal
40-295	Simple	PIL_WriteSub
		PIL_ViewSub

		PIL_ReadCal
		PIL_WriteCal
40-296	Simple	PIL_WriteSub
		PIL_ViewSub
		PIL_ReadCal
		PIL_WriteCal
40-297	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
50-295	Simple	PIL_WriteSub
		PIL_ViewSub
		PIL_ReadCal
		PIL_WriteCal
50-297	Precision	PIL_ResSetResistance
		PIL_ResGetResistance
		PIL_ReadCalDate
...		

Get Resistance Value (Visual C++)

Description

Obtains the current resistance setting of the specified programmable resistor. This function is only usable with programmable resistor models that support it: such capability is indicated in the result of PIL_ResInfo.

The value obtained for a resistance setting of infinity, if the sub-unit permits this, is HUGE_VAL (#include <math.h>).

Prototype

DWORD _stdcall PIL_ResGetResistance(DWORD CardNum, DWORD OutSub, double *Resistance);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Resistance - pointer to variable to receive the result

Returns:

Zero for success, or non-zero error code.

Related functions

PIL_ResInfo

PIL_ResSetResistance

Resistor information (Visual C++)

Description

Obtains detailed information on a programmable resistor sub-unit.

Prototype

DWORD _stdcall PIL_ResInfo(DWORD CardNum, DWORD OutSub, double *MinRes, double *MaxRes, double *RefRes, double *PrecPC, double *PrecDelta, double *Int1, double *IntPrec, DWORD *Capabilities);

Parameters:

CardNum - card number

SubNum - sub-unit number

MinRes - pointer to variable to receive minimum resistance setting

MaxRes - pointer to variable to receive maximum resistance setting

RefRes - pointer to variable to receive reference resistance value

PrecPC - pointer to variable to receive percentage precision value

PrecDelta - pointer to variable to receive offset precision, in ohms

Int1 - pointer to (currently unused) variable

IntDelta - pointer to variable to receive internal precision, in ohms

Capabilities - pointer to variable to receive capability flags (see below)

Returns:

Zero for success, or non-zero error code.

Capabilities Bit Flag Definitions

Capability bits are as follows:

0x00000008 - RES_CAP_REF (supports reference calibration value)

0x00000004 - RES_CAP_INF (supports setting "open-circuit")

0x00000002 - RES_CAP_ZERO (supports setting "zero ohms")

0x00000001 - RES_CAP_PREC (precision resistor - supporting function PIL_ResSetResistance etc.)

0x00000000 - RES_CAP_NONE (no special capabilities)

Corresponding enumerated constants are provided in Pilpxi.h.

Notes

MinRes and MaxRes are the minimum and maximum values that can be set in the sub-unit's continuous range of adjustment. If capability RES_CAP_ZERO is flagged a setting of "zero ohms" is also possible. If RES_CAP_INF is flagged an open-circuit setting is also possible.

If capability RES_CAP_REF is flagged, RefRes is the reference resistance value - such as in model 40-265, where it gives the balanced state resistance.

PrecPC and PrecDelta represent the sub-unit's precision specification, such as (±0.2%, ±0.1 ohms).

IntDelta is the notional precision to which the sub-unit works internally; this value will be less than or equal to the figure indicated by PrecPC and PrecDelta, indicating greater internal precision.

Where information is not available for the sub-unit concerned, null values are returned.

Set Resistance Value (Visual C++)

Description

Sets a programmable resistor to the closest available setting to the value specified. This function is only usable with programmable resistor models that support it: such capability is indicated in the result of PIL_ResInfo.

If the sub-unit permits, the resistance value can be set to:

- zero ohms (nominally), by passing the resistance value 0.0
- infinity, by passing the resistance value HUGE_VAL (#include <math.h>); or alternatively by using function PIL_ClearSub

The resistance value actually set can be found using PIL_ResGetResistance.

Prototype

DWORD _stdcall PIL_ResSetResistance(DWORD CardNum, DWORD OutSub, DWORD Mode, double Resistance);

Parameters:

CardNum - card number

OutSub - output sub-unit number

Mode - the resistance setting mode (see below)

Resistance - the resistance value to set

Returns:

Zero for success, or non-zero error code.

Mode value

A value indicating how the given resistance value is to be applied. Only one mode is currently supported:

Value	Ident	Function
0	RES_MODE_SET	Set resistance to the specified value

Note

In programmable resistor models having gapped ranges, resistance values falling within such gaps are not coerced. For example, in a unit supporting settings:

- zero ohms
- 100 - 200 ohms continuously variable
- infinity

attempting to set values above zero but below 100 ohms, or above 200 ohms but less than infinity, gives error ER_BAD_RESISTANCE.

Related functions

PIL_ResInfo

PIL_ResGetResistance

Programmable Potentiometer

Programmable Potentiometer

This section details the use in Visual C++ of functions specific to programmable potentiometer sub-units.

No potentiometer-specific functions are currently provided.

A potentiometer such as model 40-296 is represented logically as a programmable resistor (RES type) having twice the number of switched bits as its nominal resolution, i.e. a 24-bit potentiometer returns the type description RES(48). To make the unit behave correctly appropriate bit-patterns must be set in the upper and lower halves using general purpose output function PIL_WriteSub (or PIL_WriteSubArray). Transient effects must be expected when changing the wiper position; provided MODE_NO_WAIT is not in force resistance values can only be transiently high.

Note that a potentiometer's state at power-up and when cleared is as a device of twice the nominal resistance with its wiper centred.

WARNING

Mis-programming can result in the potentiometer presenting a lower than normal resistance between its end terminals - in the worst case zero ohms.

Non-volatile (EEPROM) storage of calibration values is supported through the functions PIL_ReadCal and PIL_WriteCal.

Programmable RF Attenuator

Programmable RF Attenuator

This section details the use in Visual C++ of functions specific to programmable RF attenuator sub-units.

Specific functions are provided to:

- Obtain attenuator information, in numeric format: PIL_AttenInfo
- Obtain attenuator description, in string format: PIL_AttenType
- Set an attenuation level, in dB: PIL_AttenSetAttenuation
- Obtain the current attenuation setting, in dB: PIL_AttenGetAttenuation
- Obtain the value of each individual attenuator pad, in dB: PIL_AttenPadValue

RF attenuator sub-units can also be controlled using general purpose output functions such as PIL_WriteSub. This allows the explicit selection of particular pad patterns that may in some circumstances yield improved RF performance.

Get attenuation (Visual C++)

Description

Obtains the current attenuation setting.

Prototype

DWORD _stdcall PIL_AttenGetAttenuation(DWORD CardNum, DWORD SubNum, float *Atten);

Parameters:

CardNum - card number

SubNum - sub-unit number

Atten - pointer to variable to receive the attenuation value, in dB

Returns:

Zero for success, or non-zero error code.

Attenuator information (Visual C++)

Description

Obtains a description of an RF attenuator sub-unit, as numeric values.

Prototype

DWORD _stdcall PIL_AttenInfo(DWORD CardNum, DWORD SubNum, DWORD *TypeNum, DWORD *NumSteps, float *StepSize);

Parameters:

CardNum - card number

SubNum - sub-unit number

TypeNum - pointer to variable to receive type code

NumSteps - pointer to variable to receive step count

StepSize - pointer to variable to receive step size, in dB

Returns:

Zero for success, or non-zero error code.

Results

RF attenuator sub-unit type code is:

8 - TYPE_ATTEN (programmable RF attenuator)

A corresponding enumerated constant is provided in Pilpxi.h.

Note

The description obtained by this function is a *logical* one; a *physical* description indicating the number of discrete pads in the attenuator can be obtained using PIL_SubInfo.

Attenuator pad value (Visual C++)

Description

Obtains the attenuation value of a numbered pad.

Prototype

DWORD _stdcall PIL_AttenPadValue(DWORD CardNum, DWORD SubNum, DWORD PadNum, float *Atten);

Parameters:

CardNum - card number

SubNum - sub-unit number

PadNum - pad number

Atten - pointer to variable to receive the pad's attenuation value, in dB

Returns:

Zero for success, or non-zero error code.

Note

This function facilitates explicit pad selection using PIL_OpBit or PIL_WriteSub, if the selections made by PIL_attenSetAttenuation are not optimal for the application.

The number of pads in the sub-unit can be found using PIL_SubInfo.

Set attenuation (Visual C++)

Description

Sets the attenuation to the specified value.

Prototype

DWORD _stdcall PIL_AttenSetAttenuation(DWORD CardNum, DWORD SubNum, float Atten);

Parameters:

CardNum - card number

SubNum - sub-unit number

Atten - the attenuation value to set, in dB

Returns:

Zero for success, or non-zero error code.

Note

The combination of pads inserted to achieve the desired attenuation level is determined by the driver for best all-round performance. In some models it may be possible to optimise particular aspects of attenuator performance by setting other pad combinations explicitly using PIL_OpBit or PIL_WriteSub. The pad value associated with each output channel can be discovered with PIL_AttenPadValue.

Attenuator type (Visual C++)

Description

Obtains a description of an attenuator sub-unit, as a text string.

Prototype

```
DWORD _stdcall PIL_AttenType(DWORD CardNum, DWORD SubNum, CHAR *Str);
```

Parameters:

CardNum - card number

SubNum - sub-unit number

Str - pointer to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Result

The format of the result is "ATTEN(<number of steps>,<step size in dB>)".

Note

The description obtained by this function is a *logical* one; a *physical* description indicating the number of discrete pads employed in the attenuator can be obtained using PIL_SubType.

Power Supplies

Power Supply functions

This section details the use in Visual C++ of functions specific to power supply sub-units.

Specific functions are provided to:

- Obtain power supply description, in string format: PIL_PsuType
- Obtain power supply information, in numeric format: PIL_PsuInfo
- Set power supply output voltage: PIL_PsuSetVoltage
- Obtain a power supply's current voltage setting: PIL_PsuGetVoltage
- Enable/disable a power supply's output: PIL_PsuEnable

Other functions that are relevant to operation of power supply sub-units include:

- Clear a power supply (restore start-up state): PIL_ClearSub
- Obtain power supply status information: PIL_SubStatus
- Retrieve a calibration value from non-volatile memory (some models): PIL_ReadCal
- Store a calibration value in non-volatile memory (some models): PIL_WriteCal

Power Supply - enable/disable output (Visual C++)

Description

Enables or disables a power supply's output.

Prototype

DWORD _stdcall PIL_PsuEnable(DWORD CardNum, DWORD SubNum, BOOL State);

Parameters:

CardNum - card number

SubNum - sub-unit number

State - 1 to enable, 0 to disable output

Returns:

Zero for success, or non-zero error code.

Note

This function is usable only with sub-units having the capability PSU_CAP_OUTPUT_CONTROL - see PIL_PsuInfo.

Power Supply - Get Voltage (Visual C++)

Description

Obtains the voltage setting of a power supply sub-unit.

Prototype

DWORD _stdcall PIL_PsuGetVoltage(DWORD CardNum, DWORD SubNum, double *Voltage);

Parameters:

CardNum - card number

SubNum - sub-unit number

Voltage - pointer to variable to receive the output setting, in Volts

Returns:

Zero for success, or non-zero error code.

Notes

The result is the nominal value to which the output has been set, not necessarily the actual voltage being output (which may be affected by device tolerances, current-limit conditions etc.).

This function is also usable with fixed-voltage supplies, returning the nominal output voltage.

Power Supply - Information (Visual C++)

Description

Obtains a description of a power supply sub-unit, as numeric values.

Prototype

DWORD _stdcall PIL_PsuInfo(DWORD CardNum, DWORD SubNum, DWORD *TypeNum, double *Voltage, double *Current, DWORD *Precis, DWORD *Capabilities);

Parameters:

CardNum - card number

SubNum - sub-unit number

TypeNum - pointer to variable to receive type code

Voltage - pointer to variable to receive rated voltage (in Volts)

Current - pointer to variable to receive rated current (in Amps)

Precis - pointer to variable to receive precision (in bits, meaningful only for programmable supplies)

Capabilities - pointer to variable to receive capability flags (see below)

Returns:

Zero for success, or non-zero error code.

Results

Power supply sub-unit type code is:

9 - TYPE_PSUDC (DC power supply)

A corresponding enumerated constant is provided in Pilpxi.h.

Capability flag bit definitions:

0x00000010 - PSU_CAP_CURRENT_MODE_SENSE (can sense if operating in current-limited mode)

0x00000008 - PSU_CAP_PROG_CURRENT (output current is programmable)

0x00000004 - PSU_CAP_PROG_VOLTAGE (output voltage is programmable)

0x00000002 - PSU_CAP_OUTPUT_SENSE (has logic-level sensing of output active state)

0x00000001 - PSU_CAP_OUTPUT_CONTROL (has output on/off control)

Certain driver functions are only usable with sub-units having appropriate capabilities - examples being:

PIL_PsuEnable

PIL_PsuSetVoltage

Power Supply - Set Voltage (Visual C++)

Description

Sets the output voltage of a power supply sub-unit to the specified value.

Prototype

DWORD _stdcall PIL_PsuSetVoltage(DWORD CardNum, DWORD SubNum, double Voltage);

Parameters:

CardNum - card number

SubNum - sub-unit number

Voltage - the output voltage to set, in Volts

Returns:

Zero for success, or non-zero error code.

Notes

The voltage value specified is rounded to the precision of the supply's DAC. The actual voltage setting can be obtained using PIL_PsuGetVoltage.

This function is usable only with sub-units having the capability PSU_CAP_PROG_VOLTAGE - see PIL_PsuInfo.

Power Supply - Type (Visual C++)

Description

Obtains a description of a power supply sub-unit, as a text string.

Prototype

DWORD _stdcall PIL_PsuType(DWORD CardNum, DWORD SubNum, CHAR *Str);

Parameters:

CardNum - card number

SubNum - sub-unit number

Str - pointer to character string to receive the result

Returns:

Zero for success, or non-zero error code.

Result

For a DC power supply the format of the result is "PSUDC(<rated voltage>,<rated current>)".

Note

More detailed information on power supply characteristics is obtainable in numeric format, using PIL_PsuInfo.

Battery Simulator

Battery Simulator

This section details the use in Visual C++ of functions specific to battery simulator models.

Models 41-750-001 and 41-751-001

No special-purpose functions are implemented for these models - they are operable using general-purpose input-output functions. See:

40-750-001

40-751-001

Model 41-752-001

Model 41-752-001 is implemented as an array of BATT sub-units, employing the following special-purpose functions for normal operation:

- Set output voltage: PIL_BattSetVoltage
- Obtain the present output voltage setting: PIL_BattGetVoltage
- Set sink current: PIL_BattSetCurrent
- Obtain the present sink current setting: PIL_BattGetCurrent
- Set output enable states: PIL_BattSetEnable
- Obtain present output enable states: PIL_BattGetEnable
- Obtain the present state of the hardware interlock: PIL_BattReadInterlockState

Battery Simulator - set voltage (Visual C++)

Description

Sets the output voltage of battery simulator (BATT type) sub-units.

Prototype

DWORD _stdcall PIL_BattSetVoltage(DWORD CardNum, DWORD SubNum, double Voltage);

Parameters:

CardNum - card number

SubNum - sub-unit number

Voltage - the output voltage to set, in Volts

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function sets the voltage of that sub-unit alone.

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), all of the card's BATT sub-units are set to the given voltage.

The voltage value specified is rounded to the precision of the sub-unit's DAC. The actual voltage setting can be obtained using PIL_BattGetVoltage.

Battery Simulator - get voltage (Visual C++)

Description

Obtains the voltage setting of a battery simulator (BATT type) sub-unit.

Prototype

DWORD _stdcall PIL_BattGetVoltage(DWORD CardNum, DWORD SubNum, double *Voltage);

Parameters:

CardNum - card number

SubNum - sub-unit number

Voltage - pointer to variable to receive the output setting, in Volts

Returns:

Zero for success, or non-zero error code.

Notes

The result is the nominal value to which the output has been set, not necessarily the actual voltage being output (which could be affected by conditions such as current-limiting).

Battery Simulator - set current (Visual C++)

Description

Sets the output sink current of battery simulator (BATT type) sub-units.

Prototype

DWORD _stdcall PIL_BattSetCurrent(DWORD CardNum, DWORD SubNum, double Current);

Parameters:

CardNum - card number

SubNum - sub-unit number

Current - the output sink current to set, in Amps

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function sets the sink current of that sub-unit alone.

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), all of the card's BATT sub-units are set to the given current.

For non-zero values, output sink current is set to the nearest available value **greater** than that specified, typically using a low-precision DAC (e.g. 4-bit). The actual sink current setting can be obtained using PIL_BattGetCurrent.

Battery Simulator - get current (Visual C++)

Description

Obtains the current sink setting of a battery simulator (BATT type) sub-unit.

Prototype

DWORD _stdcall PIL_BattGetCurrent(DWORD CardNum, DWORD SubNum, double *Current);

Parameters:

CardNum - card number

SubNum - sub-unit number

Current - pointer to variable to receive the output setting, in Amps

Returns:

Zero for success, or non-zero error code.

Battery Simulator - set enable (Visual C++)

Description

Sets the output enable pattern of battery simulator (BATT type) sub-units.

Prototype

DWORD _stdcall PIL_BattSetEnable(DWORD CardNum, DWORD SubNum, DWORD Pattern);

Parameters:

CardNum - card number

SubNum - sub-unit number

Pattern - the pattern of output enables to set

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function sets the output enable state of that sub-unit alone according to the least significant bit of Pattern (0 = OFF, 1 = ON).

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), enable states of all the card's BATT sub-units are set; bits in the supplied Pattern are utilised in ascending order of BATT sub-unit, i.e.

Pattern bit 0 = enable state of lowest numbered BATT sub-unit (0 = OFF, 1 = ON)

Pattern bit 1 = enable state of next numbered BATT sub-unit (0 = OFF, 1 = ON)

etc.

Note that the operation can fail (returning ER_EXECUTION_FAIL) if a necessary hardware interlock is disconnected.

The present enable pattern can be obtained using PIL_BattGetEnable.

Battery Simulator - get enable (Visual C++)

Description

Obtains the output enable pattern of battery simulator (BATT type) sub-units.

Prototype

```
DWORD _stdcall PIL_BattGetEnable(DWORD CardNum, DWORD SubNum,
DWORD *Pattern);
```

Parameters:

CardNum - card number

SubNum - sub-unit number

Pattern - pointer to variable to receive the output enable pattern

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function gets the output enable state of that sub-unit alone in the least significant bit of Pattern (0 = OFF, 1 = ON).

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), enable states of all the card's BATT sub-units are obtained; bits in Pattern are assigned in ascending order of BATT sub-unit, i.e.

Pattern bit 0 = enable state of lowest numbered BATT sub-unit (0 = OFF, 1 = ON)

Pattern bit 1 = enable state of next numbered BATT sub-unit (0 = OFF, 1 = ON)

etc.

Battery Simulator - read interlock state (Visual C++)

Description

Obtains the present state of a hardware interlock associated with battery simulator (BATT type) sub-units.

Prototype

DWORD _stdcall PIL_BattReadInterlockState(DWORD CardNum, DWORD SubNum, BOOL *Interlock);

Parameters:

CardNum - card number

SubNum - sub-unit number

Interlock - pointer to variable to receive the interlock state

Returns:

Zero for success, or non-zero error code.

Notes

When SubNum corresponds to a BATT sub-unit, the function gets the state of the hardware interlock associated with that sub-unit:

0 = interlock is "down"

1 = interlock is "up"

If SubNum = 0 (BATT_ALL_BATT_SUB_UNITS), the function gets the summary state of all BATT sub-unit interlocks :

0 = one or more interlocks is "down"

1 = all interlocks are "up"

Model 41-752-001 has a single global interlock affecting all channels, and both modes above yield the same result.

Interlock "up" state is hardware-latched from the physical wired interlock by the action of PIL_BattSetEnable, when that function succeeds. Hence:

- If the "up" state is indicated, the physical interlock has remained intact and outputs are enabled as previously set by PIL_BattSetEnable.
- If the "down" state is indicated, the physical interlock has been broken and all outputs will have been disabled automatically through hardware.

Mode Control

Mode Control

This section details the use in Visual C++ of functions controlling the driver's operation.

This feature is implemented through a single function: PIL_SetMode.

Set Mode (Visual C++)

Description

Allows control flags affecting the driver's global behaviour to be set and read. This function gives access to low-level control features of the Pilpxi driver and is intended for 'expert' use only - the default driver behaviour should be satisfactory for the great majority of applications.

Prototype

DWORD _stdcall PIL_SetMode(DWORD ModeFlags);

Parameters:

ModeFlags - new value for driver mode flags

Returns:

The driver's mode flags prior to executing this function.

Flag Bit Definitions

Flag bits are as follows:

0x00000000 - MODE_DEFAULT (standard operating mode)

0x00000001 - MODE_NO_WAIT (sequencing and settling time delays disabled)

0x00000002 - MODE_UNLIMITED (closure limits disabled - see **Warning** below)

0x00000004 - MODE_REOPEN (allow re-opening without clearing cards)

0x00000008 - MODE_IGNORE_TEST (enable card operation even if selftest fails - see **Warning** below)

Corresponding enumerated constants are provided in Pilpxi.h.

Warning - MODE_UNLIMITED

Use of MODE_UNLIMITED to disable the limit on the maximum number of switch closures permitted on high-density cards is **not** recommended, because it carries the danger of overheating and consequent damage to both the card itself and the system in which it is installed. See Closure Limits.

Warning - MODE_IGNORE_TEST

The MODE_IGNORE_TEST feature should be used with **extreme caution**. If a defective card is forcibly enabled, under some fault conditions a large number of outputs could be energised spuriously, resulting in overheating and consequent damage to both the card itself and the system in which it is installed. The intended purpose of this feature is to allow continued operation of a BRIC unit from which a daughtercard has been removed for maintenance. See BRIC Operation.

Borland C++

Borland C++

The following files are required for Borland C++:

- Pilpxi.h
- Pilpxi.lib
- Pilpxi.dll

Use the Help information for Visual C++. The Visual C++ code sample PILDEMO.C is also usable in Borland C++.

Pilpxi.h and Pilpxi.lib must be accessible by Borland C++ at compile-time. Typically, copies of these files can be placed in the folder containing your application's source files; alternatively your Borland C++ project may be configured to access them in their installed location (or some other centralized location). Pilpxi.lib must be added to the list of linked files for the project.

Pilpxi.dll must be accessible by your application at run-time. Windows searches a number of standard locations for DLLs in the following order:

1. The directory containing the executable module.
2. The current directory.
3. The Windows system directory.
4. The Windows directory.
5. The directories listed in the PATH environment variable.

Placing Pilpxi.dll in one of the Windows directories has the advantage that a single copy serves any number of applications that use it, but does add to the clutter of system DLLs stored there. The Pickering Setup program places a copy of Pilpxi.dll in the Windows system directory.

Note

The version of Pilpxi.lib for Borland C++ differs from that for Visual C++. Link errors will result if the wrong version is used.

Pilpxi and LabWindows/CVI

Since LabWindows/CVI is based on VISA, the pipx40 VISA driver will usually be preferred for use with it.

However use of the Pilpxi driver does permit standalone applications to be created that are not reliant on VISA, provided it is not required by other devices in the system. Such use is supported by Function Panel library pilpxi.fp.

Note that the Pilpxi driver is incompatible with the LabWindows/CVI Real-Time Module, for which use of pipx40 is essential.

LabWindows is a trademark of National Instruments Corporation.

Pilpxi and LabVIEW

Since LabVIEW is based on VISA, the pipx40 VISA driver will usually be preferred for use with it.

However use of the Pilpxi driver does permit standalone applications to be created (using Application Builder) that are not reliant on VISA, provided it is not required by other devices in the system. Such use is supported by LabVIEW library PILPXI.llb.

Note that the Pilpxi driver is incompatible with the LabVIEW Real-Time Module, for which use of pipx40 is essential.

LabVIEW is a trademark of National Instruments Corporation.

Utility Programs

Utility Programs

The Pilpxi driver is supported by a number of utility programs:

- Test Panels
- Terminal Monitor
- Demonstration Program
- Diagnostic Utility

Test Panels

The Test Panels application allows any combination of cards to be controlled using a graphical interface.

Terminal Monitor

PILMon is a simple terminal monitor program for Pickering PXI cards. Use the HE command within PILMon to obtain help.

PILMon requires Pilpxi.dll and Ucomm32.dll.

PILMon has a number of command-line options when starting the program. For instructions, in a Command Prompt window with the current directory set to that containing PILMon, type:

PILMON -?

```
C:\Pickering\Utils>pilmon -?

Program:   PIL PXI Monitor

Syntax:    PILMon [-cN] [-r] [-n]

Arguments: -cN specifies the number of the COM port (1 thru 9) to use

                in lieu of the console. COM settings are 9600/8/N/1.

           -r specifies that when run PILMon should attempt to open

              the cards without clearing them. This may or may not be

              possible.

           -n specifies that when run PILMon should NOT automatically

              open the cards. Overrides -r if both are used.

Options are accepted in any order.

Example:   PILMon -c2 -r -n
```

The action of many PILMon commands corresponds closely to Pilpxi driver functions (hyperlinks here access the Visual C++ function references):

	Pilpxi driver function	**Corresponding PILMon command**

Initialise		
Initialise all cards	PIL_OpenCards	OC
Initialise single card	PIL_OpenSpecifiedCard	VO (see note 1)
Close		
Close all cards	PIL_CloseCards	CC
Close single card	PIL_CloseSpecifiedCard	VC (see note 1)
Card ID, Properties and Status		
Get card identification	PIL_CardId	See note 2
Get card location	PIL_CardLoc	See note 2
Get sub-unit closure limit	PIL_ClosureLimit	CL
Get count of unopened cards	PIL_CountFreeCards	CF
Get diagnostic information	PIL_Diagnostic	DI
Get sub-unit counts	PIL_EnumerateSubs	See note 2
Get locations of unopened cards	PIL_FindFreeCards	LF
Get sub-unit settling time	PIL_SettleTime	SE
Get card status	PIL_Status	ST
Get sub-unit information	PIL_SubInfo	See note 2
Get sub-unit description	PIL_SubType	See note 2
Get driver version	PIL_Version	See Note 3
Output Operations		
Clear outputs of all open cards	PIL_ClearAll	RS
Clear a single card's outputs	PIL_ClearCard	AR
Clear a sub-unit's outputs	PIL_ClearSub	CS
Set or clear a single output	PIL_OpBit	SC and SO
Set or clear a matrix crosspoint	PIL_OpCrosspoint	XC and XO
Set a sub-unit's output pattern	PIL_WriteSub	SB
	PIL_WriteSubArray	
Get a single output's state	PIL_ViewBit	SV
Get a matrix crosspoint's state	PIL_ViewCrosspoint	XV
Get a sub-unit's output	PIL_ViewSub	BV

pattern		
	PIL_ViewSubArray	
Output Masking		
Clear a sub-unit's mask	PIL_ClearMask	CM
Set or clear a single output's mask	PIL_MaskBit	SM
Set or clear a matrix crosspoint's mask	PIL_MaskCrosspoint	XM
Set a sub-unit's mask pattern	PIL_WriteMask	MB
	PIL_WriteMaskArray	
Get a single output's mask state	PIL_ViewMaskBit	MS
Get a matrix crosspoint's mask state	PIL_ViewMaskCrosspoint	XS
Get a sub-unit's mask pattern	PIL_ViewMask	MV
	PIL_ViewMaskArray	
Output Calibration		
Read an output's calibration value	PIL_ReadCal	RC
Write an output's calibration value	PIL_WriteCal	WC
Input Operations		
Read single input	PIL_ReadBit	IS
Read input sub-unit pattern	PIL_ReadSub	BR
Mode Control		
Set driver mode	PIL_SetMode	DM

Notes

1. Normally when PILMon is started it immediately takes control of all cards using the PIL_OpenCards mechanism. In order to use the PIL_OpenSpecifiedCard mechanism, PILMon must be started from the command-line with the "-n" option specified.
2. Where noted, the information obtained by this function is displayed as part of the output from the PILMon LS command.
3. The value returned by PIL_Version is displayed when PILMon is started as "Pilpxi Driver version number".

Demonstration Program

PILDemo is a simple console-based demonstration program that exercises all installed Pickering cards, using many of the driver's functions.

The operations performed are as follows:

- the installed cards are listed
- each input sub-unit (if any) of each card is read once

Then, if one or more output sub-units is present:

- where possible, for each sub-unit in turn: all outputs are activated simultaneously, then de-activated (using PIL_WriteSub())
- where possible, for each sub-unit in turn: all outputs are activated simultaneously, then de-activated (using PIL_WriteSubArray())
- the program cycles indefinitely, activating each output individually in turn

A dwell delay (nominally 10 milliseconds) is provided between each state-change.

The program requires Pilpxi.dll.

WARNING

THIS PROGRAM ACTIVATES OUTPUTS BOTH INDIVIDUALLY AND IN COMBINATIONS. IT SHOULD NOT BE RUN UNDER ANY CONDITIONS WHERE DAMAGE COULD RESULT FROM SUCH EVENTS. FOR GREATEST SAFETY IT SHOULD BE RUN ONLY WHEN NO EXTERNAL POWER IS APPLIED TO ANY CARD.

Diagnostic Utility

The Plug & Play functionality of PXI cards generally ensures trouble-free installation. However in the event of any problems, it may be helpful to know how cards have been configured in the system. The PipxDiag Windows diagnostic utility generates an extensive report showing the allocations of PCI/PXI system resources and specific details of installed Pickering cards, highlighting any potential configuration issues.

In the diagnostic report, all the installed Pickering cards should be listed in the "Pilpxi information" section - if one or more cards is missing it may be possible to determine the reason by referring to the PCI configuration dump contained in the report, but interpretation of this information is far from straightforward, and the best course is to contact Pickering support: support@pickeringtest.com, if possible including a copy of the diagnostic report.

In the "VISA information" section, if VISA is not installed its absence will be reported. This does not affect operation using the Direct I/O driver, and is not a problem unless you also wish to use VISA. VISA is a component of National Instruments LabWindows/CVI and LabVIEW, or is available as a standalone environment.

If VISA is present and is of a sufficiently recent version, the section "Pipx40 information" should contain a listing similar to "Pilpxi information".

Please note that the Diagnostic Utility cannot access cards if they are currently opened by some other application, such as the Test Panels or Terminal Monitor.

Application Notes

Application Notes

This section contains application notes on the following topics:

- BRIC Operation
- Closure Limits
- Execution Speed
- Isolation Switching
- Multiprocessing and Multithreading
- Simple Programmable Resistor Cards
- Segmented Matrix
- Unsegmented Matrix

BRIC Operation

BRIC closure limits

As with other high-density units, for a BRIC the Pilpxi driver imposes a limit on the maximum number of channel closures - see Closure Limits. Although PIL_SetMode offers a means of disabling this limit, the extraordinarily high packing density in BRIC units makes observation of maximum closure limits particularly important. The consequences of turning on an excessive number of crosspoints can be appreciated from the fact that each activated crosspoint may consume around 10mA at 5V (50mW, or 1W per 20 crosspoints). The power consumption of a large BRIC with all crosspoints energised would be beyond the capacity of the system power supply and backplane connectors, never mind its cooling capabilities. For this reason BRIC units are fuse-protected against overcurrent. However, it cannot protect against local hot-spots within a BRIC if too large a block of physically adjacent crosspoints is energised. Although the fuse is self-resetting under moderate overload, a massive overload may cause it to rupture permanently.

BRIC daughtercard removal

In the event of a BRIC daughtercard being removed for servicing, operation of the entire unit is normally disabled. It is possible to allow continued operation in spite of this fault condition using the MODE_IGNORE_TEST option bit in PIL_SetMode. When this mode is set, the tests performed when the card is opened will still detect the fault and flag it in the card's PIL_Status value (bit STAT_HW_FAULT = set); however it will no longer be flagged as disabled (bit STAT_DISABLED = clear), allowing continued operation.

Multifunction BRICs

Multifunction BRICs have independently controlled isolation switches. In operating these units it is advised that where hot-switching occurs programmers ensure that matrix crosspoint relays hot-switch, and isolation relays cold-switch. This avoids concentrating the contact wear caused by hot-switching in the isolation relays, which could lead to a reduction in their operational life. The preferred operating sequences for hot-switching are:

- When closing a crosspoint, first close the isolation switch, then the crosspoint switch
- When opening a crosspoint, first open the crosspoint, then the isolation switch

Closure Limits

The high switch density attained in certain System 40/45/50 cards, particularly high-density matrix types, necessitates close packing of relays and airflow is quite restricted. If excessive numbers of relays were energised for a prolonged period overheating could occur. For example, in model 40-531 simultaneous energisation of all 256 relays would yield a power dissipation of around 17W. In BRIC units the situation is even more extreme - see BRIC Operation. To guard against this danger the software driver places a limit on the number of crosspoints that can be energised simultaneously. The limits imposed by the driver are set with regard to operating temperature levels and will not cause any difficulty for typical matrix usage, where only a small proportion of crosspoints are simultaneously ON. A sub-unit's closure limit can be discovered using the PIL_ClosureLimit function (see reference for Visual Basic / Visual C++).

In some models, energisation of too many relays would cause the card's supply current to exceed the maximum available from the system backplane, with the potential for overheating and damage to the card and backplane connectors.

The software driver does however provide a method of disabling this protection. Calling the function PIL_SetMode (see reference for Visual Basic / Visual C++) with the bit MODE_UNLIMITED set allows an unlimited number of crosspoints to be energised simultaneously. This feature should be used with **EXTREME CAUTION**. Although it may be safe to energise larger numbers of crosspoints where ON times are short and duty cycle is low, it must be borne in mind that if the user's program were to halt in the ON state (for example at a breakpoint when debugging) the danger of overheating is present.

Some models incorporate fuses to protect against simultaneous activation of a hugely excessive number of channels. These are self-resetting in moderate overload, and operation will be restored when the fault condition clears.

Execution Speed

Internal optimisations

Generally, the Pilpxi driver optimises a card's internal switch operations as far as possible. For example in a single-channel multiplexer (MUX type) with isolation switching, if a channel-change is requested the isolation switch is not cycled. This saves both time and mechanical wear on the switch.

Break-before-make action

By default, the Pilpxi driver enforces Break-Before-Make (BBM) action and settling delays (to cope with contact bounce) on all switching operations. This ensures 'clean' switching actions and minimises the danger of switch damage due to conflicting contact closures.

For time-critical applications the driver can be set to omit all sequencing delays using the MODE_NO_WAIT option of PIL_SetMode - see reference for Visual Basic / Visual C++. This causes the driver to return control to the application program in the shortest possible time. The function PIL_Status (see reference for Visual Basic / Visual C++) can then be used at a later time to determine when operations on a particular card have completed (indicated by the bit STAT_BUSY becoming clear). By this method a number of switching operations (and/or other program activity) can be executed in parallel rather than sequentially. However the programmer must guard against switch conflicts that might transiently cause, say, the shorting of a power supply and consequent switch damage.

In some cards (for example model 40-745), making an individual channel selection involves several physical relays. Normally, sequencing delays are imposed to ensure that no unwanted transient connections occur. Setting MODE_NO_WAIT bypasses these delays, and the programmer must bear in mind the potential for transient conflicts.

Default driver action is restored by executing PIL_SetMode with the MODE_NO_WAIT bit clear.

Many System 40/45/50 relay cards exhibit very short basic execution times in the order of a few tens of microseconds; however BBM and settling delays associated with relays may extend from a few hundred microseconds (for small reed relays) to some tens of milliseconds (for microwave switches). Here, setting MODE_NO_WAIT and appropriate programming can free a significant amount of CPU time for other purposes.

There are some exceptions to the above: for example digital outputs generally have zero settling time and MODE_NO_WAIT offers no performance advantage.

To summarise, where execution speed is of paramount importance setting MODE_NO_WAIT can offer significant advantages for many cards; however it is more demanding for the programmer, requiring an understanding of the operational characteristics of specific card types and taking greater account of conditions in the switched circuits.

Processor speed

A faster processor might be expected to yield faster operation. However for many cards much of a function's execution time is spent waiting for switch contacts to stabilise, so unless MODE_NO_WAIT is invoked little improvement will be seen. Further, modern processors are capable of operating many cards near or beyond their hardware limits, and the Pilpxi driver includes timing control to ensure reliable operation. Therefore increases in processor speed beyond about 3GHz may well give no actual improvement in operating speed.

Isolation Switching

Isolation switching is incorporated in particular models for a variety of reasons:

- Reducing capacitive loading on a node. In low-frequency units, reduced capacitive loading gives faster response times when medium to high impedance signals are being carried.
- Reducing circuit leakage current. Reduced leakage current in the switch circuits is advantageous where low-current measurements are involved.
- Reducing the length of circuit stubs on a node. In high-frequency units, reduced stub lengths give better RF performance.
- Providing alternate switching functionality. Some versatile models utilise isolation switching to support additional operating modes.

A related feature is loopthru switching, which provides a default connection path when no other path is selected.

Automatic isolation and loopthru switching

Isolation and loopthru switches are normally controlled automatically by the Pilpxi driver, and their operation is entirely transparent to the user.

In some applications or for fault diagnostic purposes it may be desirable to control isolation and loopthru switches independently. There are two ways of achieving this:

1. In matrix types having auto-isolation and/or auto-loopthru, function PIL_OpSwitch permits explicit control of individual switches.
2. Cards can usually be reconfigured to allow independent control of isolation or loopthru switches using the ordinary control functions - if you have such a requirement please contact support@pickeringtest.com.

Multiprocessing and Multithreading

Multiprocessing involves operation of cards by multiple software processes (i.e. programs); multithreading uses multiple execution threads within a single program. Multithreading is a feature of certain programming environments and can also be managed through the standard Windows API.

Process-safety

The Pilpxi driver is process-safe.

The mechanisms for opening and closing Pickering cards allow a particular card to be controlled by only one process at any time.

Using the PIL_OpenCards mechanism, a process awaiting the release of cards by another process can repetetively call PIL_OpenCards: the function will return zero until control can be obtained. Using the PIL_OpenSpecifiedCard mechanism, repeated calls to PIL_OpenSpecifiedCard return an error until the card becomes available.

Multiprocess operation can be investigated by running two copies of the PILMon terminal monitor program concurrently. If you wish to test the action of the PIL_OpenSpecifiedCard mechanism, PILMon must be started from the command-line with the "-n" option specified to prevent it taking control of the cards with PIL_OpenCards.

Thread-safety

The Pilpxi driver is thread-safe.

Execution of a Pilpxi driver function by one thread simply blocks its execution by other threads or processes. This includes any settling delay periods, ensuring that no unwanted overlaps occur in operation.

Functions PIL_OpenCards and PIL_OpenSpecifiedCard

Using the PIL_OpenCards mechanism, a process takes control of all Pickering cards that are not currently under the control of some other process. PIL_OpenSpecifiedCard just takes control of the chosen card. Only one of these mechanisms can be employed at any time; after loading Pilpxi.dll the first use of one mechanism disables the other. Thus if multiple applications are to access cards they must all employ the same mechanism for opening and closing them.

Function PIL_SetMode

The settings made by PIL_SetMode are process-specific, i.e. multiple processes can operate with different settings. One mode flag used in PIL_SetMode, MODE_REOPEN, affects cross-process behaviour - see below.

Closing and re-opening cards

Normally when cards are opened using PIL_OpenCards all the cards are cleared. The reason for this is that there is no facility to read the state of a card's outputs from the card itself, so that when taking control the software driver has no way of discovering the pre-existing state of the card.

The Pilpxi driver does however provide a mechanism that allows cards to be re-opened by PIL_OpenCards with their existing states intact. This permits cards to be opened and set up by one application, then closed and re-opened by a second application with their states undisturbed. Note that this facility is not available when using PIL_OpenSpecifiedCard.

The requirements for this mode of operation are:

1. The application performing the set-up must have called PIL_SetMode with the bit MODE_REOPEN set prior to releasing the cards with PIL_CloseCards.
2. The application taking control of the cards must call PIL_SetMode with the bit MODE_REOPEN set prior to executing PIL_OpenCards.
3. Pilpxi.dll must remain loaded between execution of PIL_CloseCards by the releasing process and execution of PIL_OpenCards by the process taking control.

If these requirements are not met, cards will be cleared as normal by PIL_OpenCards.

The process re-opening the cards can of course be the same one that released them. The method employed involves disk access so the operation does take a significant time, which depends to some extent on the number of cards installed.

Simple Programmable Resistor Cards

Applicable to models:

- 40-290
- 40-291
- 40-295
- 40-296
- 50-295

Simple programmable resistor cards employ a series chain of individual fixed resistors, each having an associated shorting switch. In standard models the fixed resistor values are arranged in a binary sequence. The discussion below relates to 16-bit models; some considerations may be either more or less significant in models with higher or lower resolution.

Application considerations: 16-bit models

The binary resistor chain employed in a 16-bit programmable resistor card provides a notional resolution of about 0.002% (or 15ppm) of the total resistance.

In exploiting this high resolution there are a number of factors which should be taken into account:

- The absolute accuracy of the resistors fitted may be only 1% or 0.5% (i.e. less than 8 bits).
- For 'custom' resistor-chain values, components having the precise nominal values required may be unobtainable, and the nearest available preferred values may have to be used.
- The resistors have a non-zero temperature coefficient, typically of ±50ppm/°C, though values down to ±15ppm/°C may be obtainable.
- The closed-contact resistance of the switch shunting each resistor is of the order of 100 milliohms. In the reed switches employed in these cards this value is highly stable, provided switches are not subjected to overcurrent. This includes transient currents, such as may occur if a pre-charged capacitive circuit is discharged through a low resistance.
- Wiring and connectors impose a small resistance in series with the resistor chain, of perhaps 200 milliohms.

Some implications of these factors are:

- The relationship between the switch pattern and the programmed resistance value is not guaranteed to be monotonic (i.e. a change in switch pattern that might be expected to yield an increase in resistance value may in fact decrease it, and vice-versa).

- A resistance value of zero ohms is unobtainable. The lowest value that can be achieved is composed of the closed-contact resistances of 16 relays in series, together with wiring and connector resistance. A value of around 1.8 ohms is typical.
- Temperature effects can significantly exceed the notional resolution. For example, a temperature change of only 5°C may cause a resistance change of ±250ppm, or 17 times the notional resolution. The resistance of wiring and closed switch contacts is also affected by temperature.

The cards have the facility to store in non-volatile memory a 16-bit value associated with each resistor. These values can be used to calibrate the card to provide greater setting accuracy than the basic absolute accuracy of the resistors employed in the chain. Usage and interpretation of stored values is entirely user-specific: the software driver merely provides a mechanism (functions PIL_WriteCal and PIL_ReadCal) for storing and retrieving them.

A possible scheme for utilising the stored calibration values might be:

- Employ the stored values to somehow represent the deviation of each resistor's actual value from its nominal value (say, as a percentage: treated as a signed quantity the 16-bit value might be chosen to represent a range of ±32.767%).
- Use a calibration procedure to obtain and store an appropriate value for each individual resistor.
- Software must then make use of the stored calibration data when programming specific resistance values, taking into account extraneous circuit resistances. Because of the non-monotonic relationship between switch pattern and resistance value, some calculation is necessary to obtain a pattern matching a chosen value. A simple C program ProgResFind.c demonstrates a possible approach to this.

ProgResFind.c

This program demonstrates a possible algorithm for use in obtaining a specific resistance value in a 16-bit programmable resistor card, using stored calibration values for enhanced accuracy.

```
/* Program: ProgResFind.c */

/* Programmable resistor: find a 16-bit code to give a particular
resistance value */

/* D.C.H  16/8/01 */

/* Overall accuracy is determined by the accuracy of the calibration
values employed */



#include <stdio.h>



/* To  output debug info... */

/* *** #define DEBUG */



/* === SEARCH VALUES
====================================================== */

/* The resistance value to search for, ohms */

double search_res = 1000.0;

/* The required tolerance (fractional) */

double search_tol    = 0.0005;        /* = 0.05% */



/* === CALIBRATION VALUES
==================================================== */

/* Offset resistance value, ohms: includes connector and wiring.

   This example includes a 50R offset resistor. */

/* For accuracy, this should ideally be a CALIBRATED value */

double res_offset = 50.2;
```

```
/* The installed resistor values, ohms */

/* For accuracy better than resistor tolerance these must be
CALIBRATED values,

   not NOMINAL ones. */

double res_value[16] =

{

    0.12,

    0.22,

    0.56,

    1.13,

    2.26,

    4.42,

    8.2,

    18.0,

    37.4,

    71.5,

    143.0,

    287.0,

    576.0,

    1130.0,

    2260.0,

    4530.0

};



/* Relay closed-contact resistance, ohms: assumed identical for all
relays */

double res_contact = 0.1;

/*
======================================================================
=== */
```

```
/* Prototype */

long find_code(double value, double tolerance);



int main(void)

{

    long code;

    printf("Programmable Resistor Code Finder\n");

    printf("=================================\n");

    printf("D.C.H  16/8/01\n\n");

    printf("Search for %8.2f ohms (+/- %1.3f%%)...\n", search_res,
search_tol * 100);

    code = find_code(search_res, search_tol);

    if (code < 0)

        printf("No code matches this value within the specified
tolerance\n");

    else

        printf("Code 0x%04X\n", code);

    return 0;

}



/* Function: parallel resistor calculation */

double parallel_resistance(double r1, double r2)

{

    return ((r1 * r2) / (r1 + r2));

}



/* Function: find the first code whose actual value  matches the
search value
```

```
   within the specified tolerance band.

   Returns the code (0x0000 thru 0xFFFF).

   If no code generates a value that lies within the specified
tolerance band,

   returns -1.

   The method simply searches all codes - some optimisation is
possible. */

long find_code(double value, double tolerance)

{

    long code;

    int bit;

    double res;

    /* Search all codes */

    for (code = 0; code < 0x10000L; code++)

    {

        res = res_offset;

        for (bit = 0; bit < 16; bit++)

        {

            if (code & (1 << bit))

            {

                /* This bit is ON (switch closed) */

                res += parallel_resistance(res_value[bit],
res_contact);

            }

            else

            {

                /* This bit is OFF (switch open) */

                res += res_value[bit];

            }
```

```
        }

        if ( res > (value * (1.0 - tolerance)) && res < (value * (1.0
+ tolerance)) )

        {

#ifdef DEBUG

            printf("Code 0x%04X = %8.2f ohms\n", code, res);

#endif

            return code;

        }

    }

    return -1;

}
```

Segmented Matrix

Segmented Matrix

A segmented matrix is one in which groups of lines on an axis are served by separate sets of isolation switches on the opposing axis.

When operated by functions such as:

- PIL_OpBit
- PIL_WriteSub
- PIL_OpCrosspoint

isolation switching is handled automatically by the driver, and the sub-unit's internal structure is immaterial to a user; use of PIL_OpSwitch however requires an understanding of this.

Examples of segmented matrix types:

- 40-725-511: 8 x 9, segmented on both axes
- 40-726-751-LT: 12 x 8, segmented on both axes, with loopthru on Y-axis only
- 40-560-021: 50 x 8 specimen BRIC configuration, segmented on X-axis (Y-isolation only)

Segmented Matrix 40-725-511

40-725-511 is an 8 x 9 matrix, segmented on both axes.

In its standard configuration as a single 8 x 9 matrix sub-unit, when channel selections are made using functions such as:

- PIL_OpBit
- PIL_WriteSub
- PIL_OpCrosspoint

operation of isolation switches is automated to optimise connections for X - Y signal routing. PIL_OpSwitch allows access to individual switches for other routing schemes or fault diagnostic purposes.

Note that an alternate logical configuration treats the card as multiple sub-units, giving independent access to all switches via the ordinary control functions: for that configuration PIL_OpSwitch is not applicable.

Attribute values

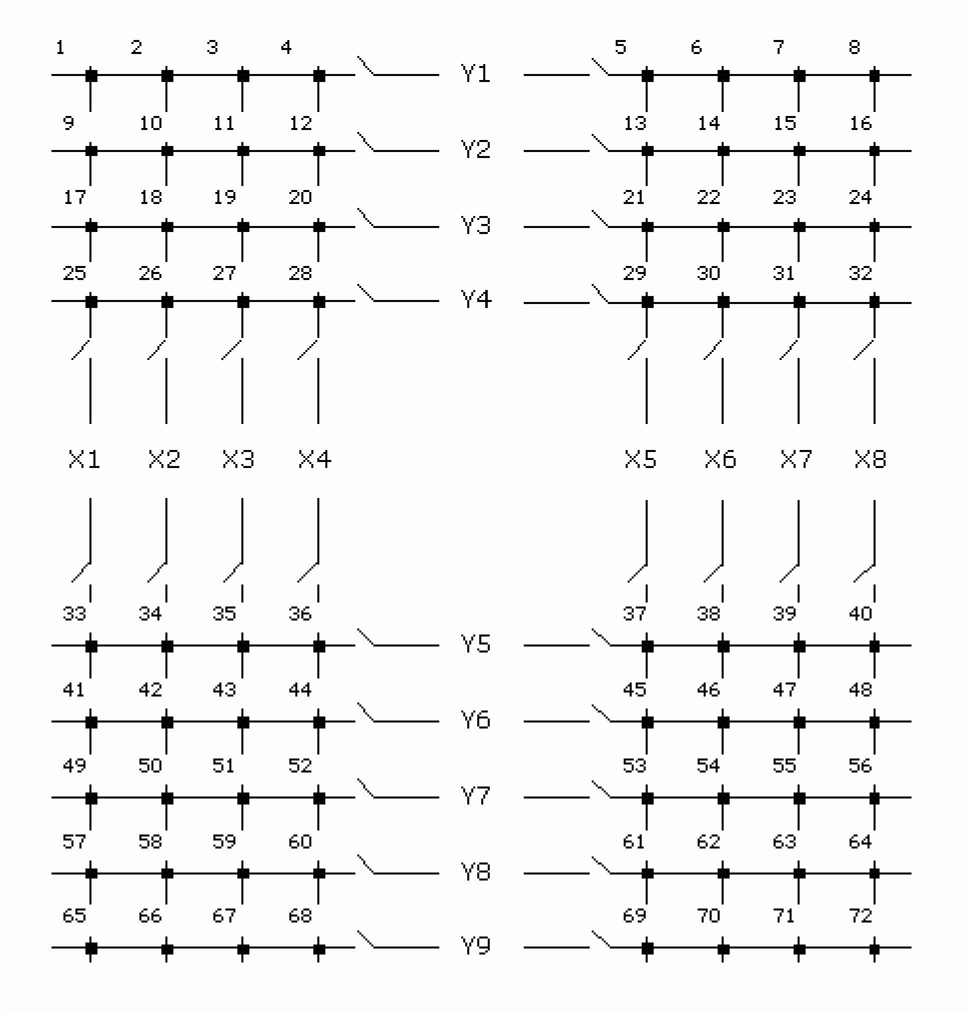
The relevant values obtained by PIL_SubAttribute when configured for auto-isolation are:

Attribute code	Attribute value
SUB_ATTR_CHANNEL_SUBSWITCHES	1
SUB_ATTR_X_ISO_SUBSWITCHES	1
SUB_ATTR_Y_ISO_SUBSWITCHES	1
SUB_ATTR_NUM_X_SEGMENTS	2
SUB_ATTR_X_SEGMENT01_SIZE	4
SUB_ATTR_X_SEGMENT02_SIZE	4
SUB_ATTR_NUM_Y_SEGMENTS	2
SUB_ATTR_Y_SEGMENT01_SIZE	4
SUB_ATTR_Y_SEGMENT02_SIZE	5

Global crosspoint switch numbers

These numbers correspond to the channel numbers used with PIL_OpBit and are valid for PIL_OpSwitch when:

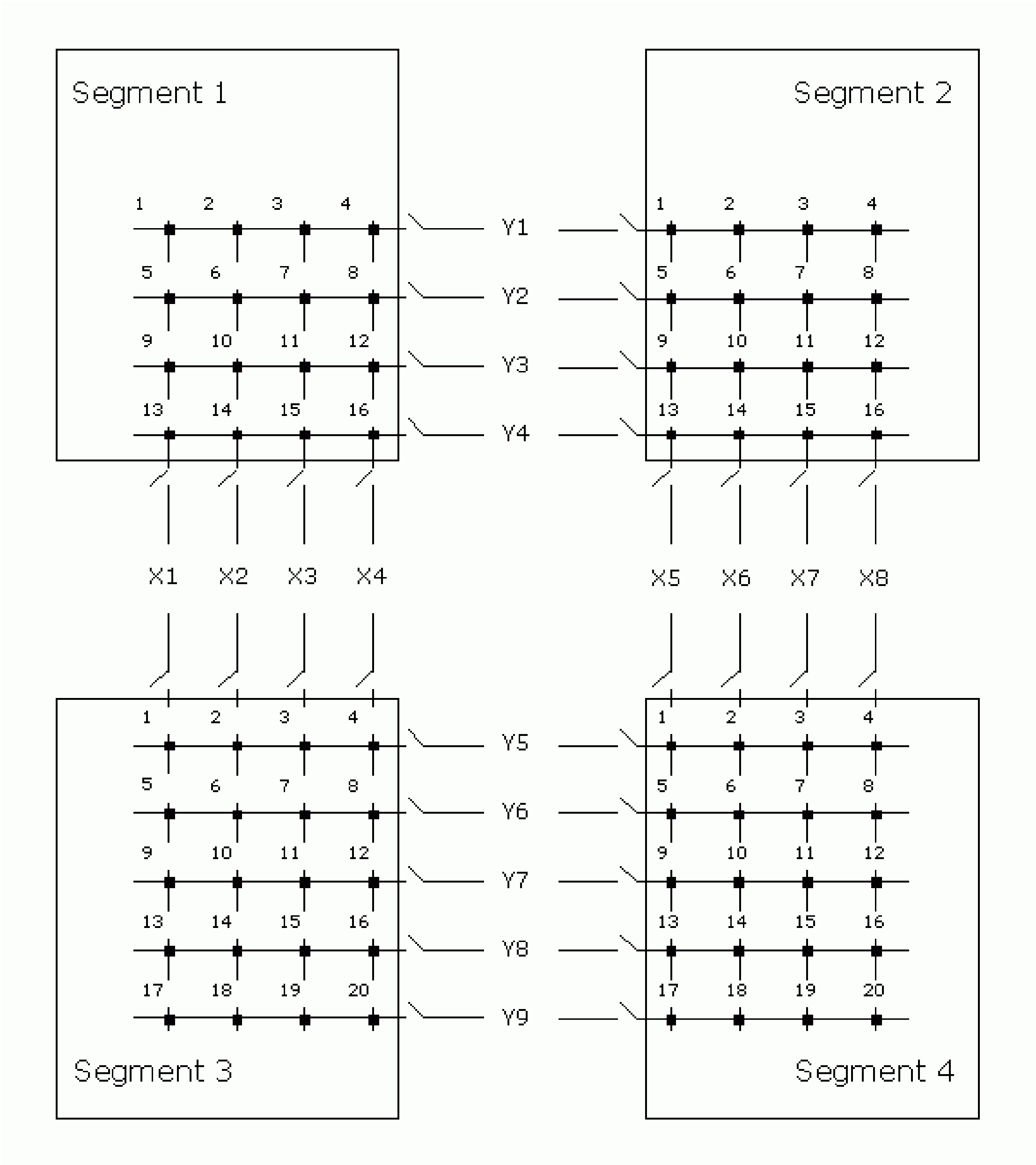
- SwitchFunc = SW_FUNC_CHANNEL
- SegNum = 0



Segment-local crosspoint switch numbers

These switch numbers are valid for PIL_OpSwitch when:

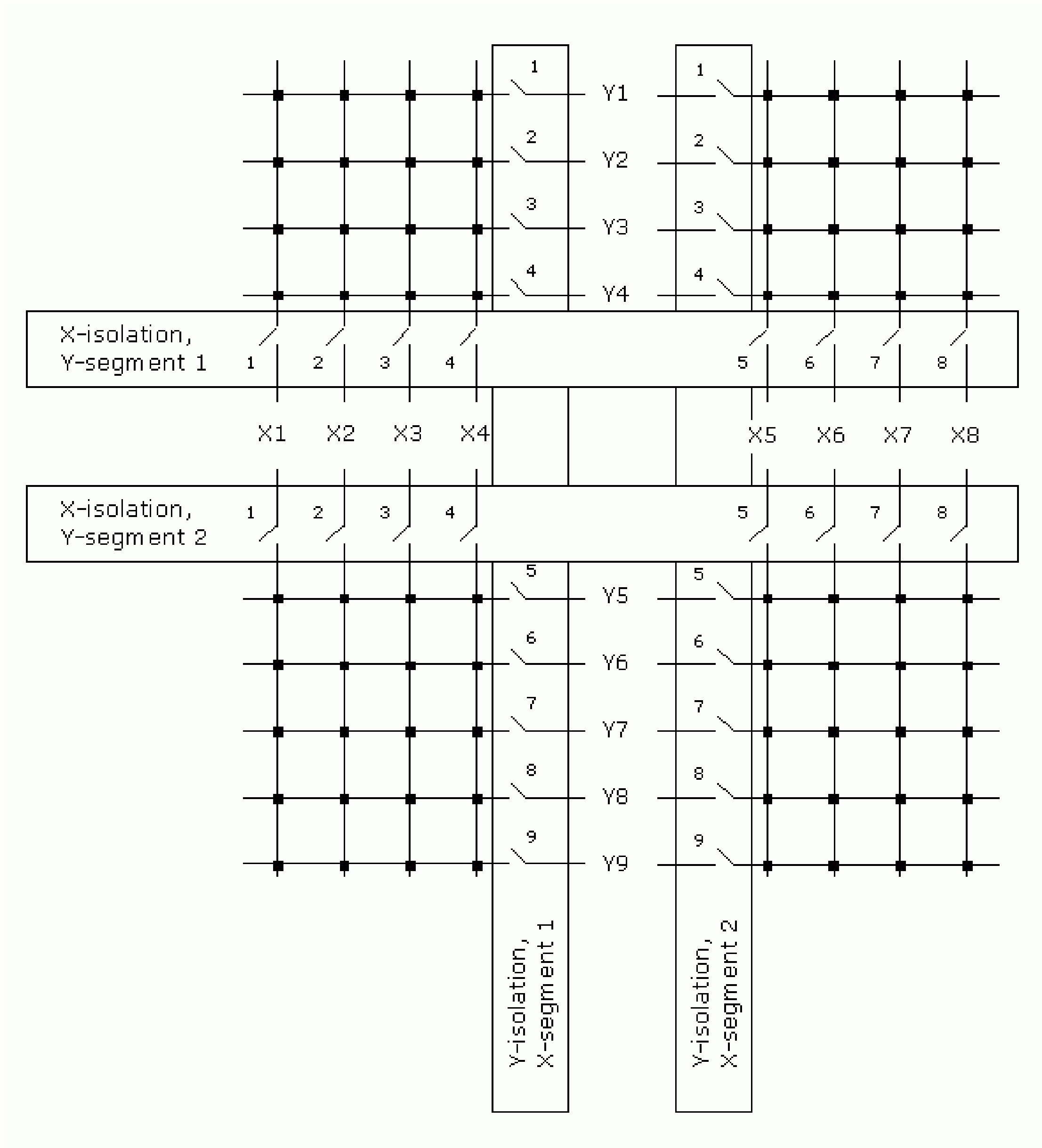
- SwitchFunc = SW_FUNC_CHANNEL
- SegNum = 1 thru 4



Isolation switch numbers

These switch numbers are valid for PIL_OpSwitch when:

- SwitchFunc = SW_FUNC_X_ISO or SW_FUNC_Y_ISO
- SegNum = 1 or 2


Y1
Y2
Y3
Y4
X-isolation,
Y-segment 1
X1
X2
X3
X4
X5
X6
X7
X8
X-isolation,
Y-segment 2
Y5
Y6
Y7
Y8
Y9
Y-isolation,
X-segment 1
Y-isolation,
X-segment 2

Segmented Matrix 40-726-751-LT

Operation of this model's crosspoint and isolation switches by PIL_OpSwitch is similar to that of 40-725-511, which only differs dimensionally - the size of each segment in 40-726 being 6 x 4.

In addition, this model incorporates loopthru switches on all lines of its Y-axis.

Note that an alternate logical configuration treats the card as multiple sub-units, giving independent access to all switches via the ordinary control functions: for that configuration PIL_OpSwitch is not applicable.

Attribute values

The relevant values obtained by PIL_SubAttribute when configured for auto-isolation and auto-loopthru are:

Attribute code	Attribute value
SUB_ATTR_CHANNEL_SUBSWITCHES	1
SUB_ATTR_X_ISO_SUBSWITCHES	1
SUB_ATTR_Y_ISO_SUBSWITCHES	1
SUB_ATTR_X_LOOPTHRU_SUBSWITCHES	0
SUB_ATTR_Y_LOOPTHRU_SUBSWITCHES	1
SUB_ATTR_NUM_X_SEGMENTS	2
SUB_ATTR_X_SEGMENT01_SIZE	6
SUB_ATTR_X_SEGMENT02_SIZE	6
SUB_ATTR_NUM_Y_SEGMENTS	2
SUB_ATTR_Y_SEGMENT01_SIZE	4
SUB_ATTR_Y_SEGMENT02_SIZE	4

Segmented Matrix 40-560-021

This documents a specimen 40-560-021 BRIC configuration, as a 50 x 8 matrix using two 46 x 8 daughtercards; the second daughtercard being partially populated as 4 x 8. This design is segmented only on the X-axis (each daughtercard having Y-isolation switches only).

In its standard configuration as a single 50 x 8 matrix sub-unit, when channel selections are made using functions such as:

- PIL_OpBit
- PIL_WriteSub
- PIL_OpCrosspoint

operation of isolation switches is automated to optimise connections for X - Y signal routing. PIL_OpSwitch allows access to individual switches for other routing schemes or fault diagnostic purposes.

Note that an alternate logical configuration is possible, the unit being treated as multiple sub-units and giving independent access to all switches via the ordinary control functions: for that configuration PIL_OpSwitch would not be applicable.

In a unit employing a larger number of daughtercards, the number of X-segments is correspondingly increased.

Attribute values

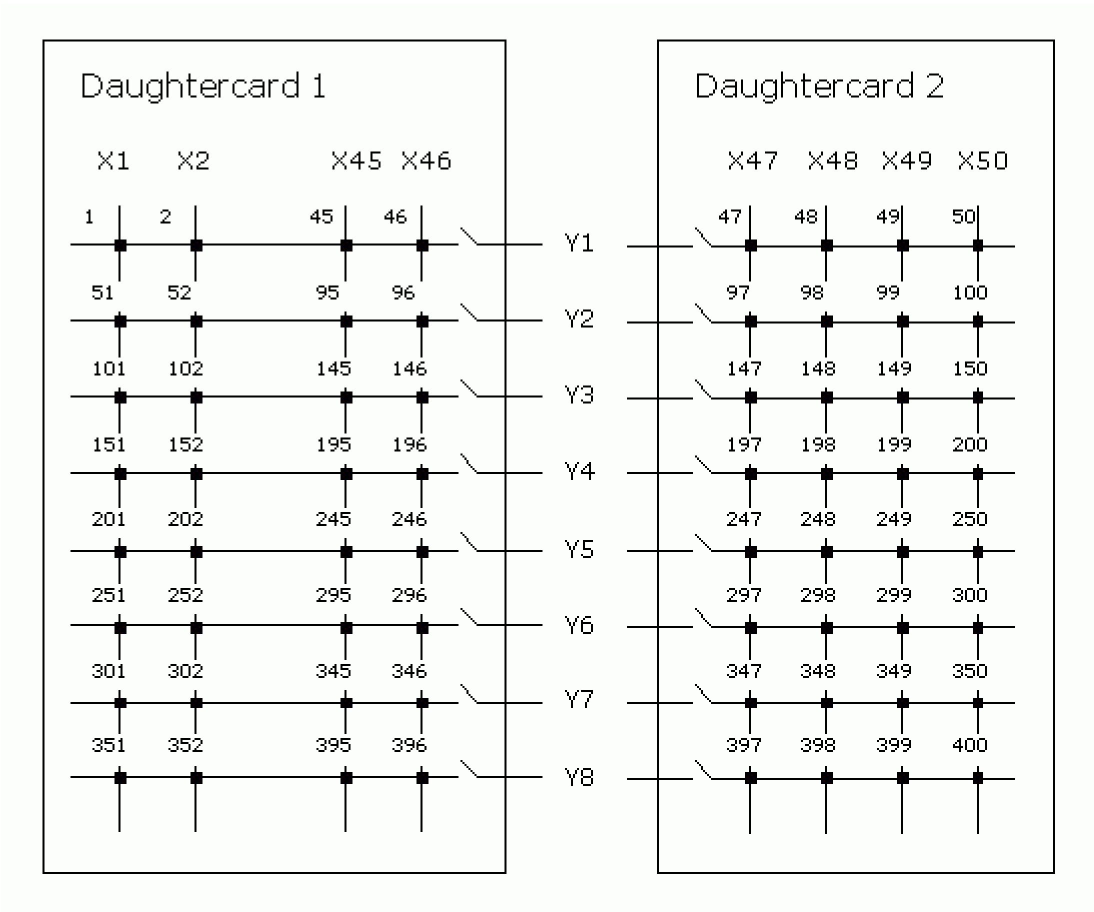
The relevant values obtained by PIL_SubAttribute when configured for auto-isolation are:

Attribute code	Attribute value
SUB_ATTR_CHANNEL_SUBSWITCHES	1
SUB_ATTR_X_ISO_SUBSWITCHES	0
SUB_ATTR_Y_ISO_SUBSWITCHES	1
SUB_ATTR_NUM_X_SEGMENTS	2
SUB_ATTR_X_SEGMENT01_SIZE	46
SUB_ATTR_X_SEGMENT02_SIZE	4
SUB_ATTR_NUM_Y_SEGMENTS	1
SUB_ATTR_Y_SEGMENT01_SIZE	8

Global crosspoint switch numbers

These numbers correspond to the channel numbers used with PIL_OpBit and are valid for PIL_OpSwitch when:

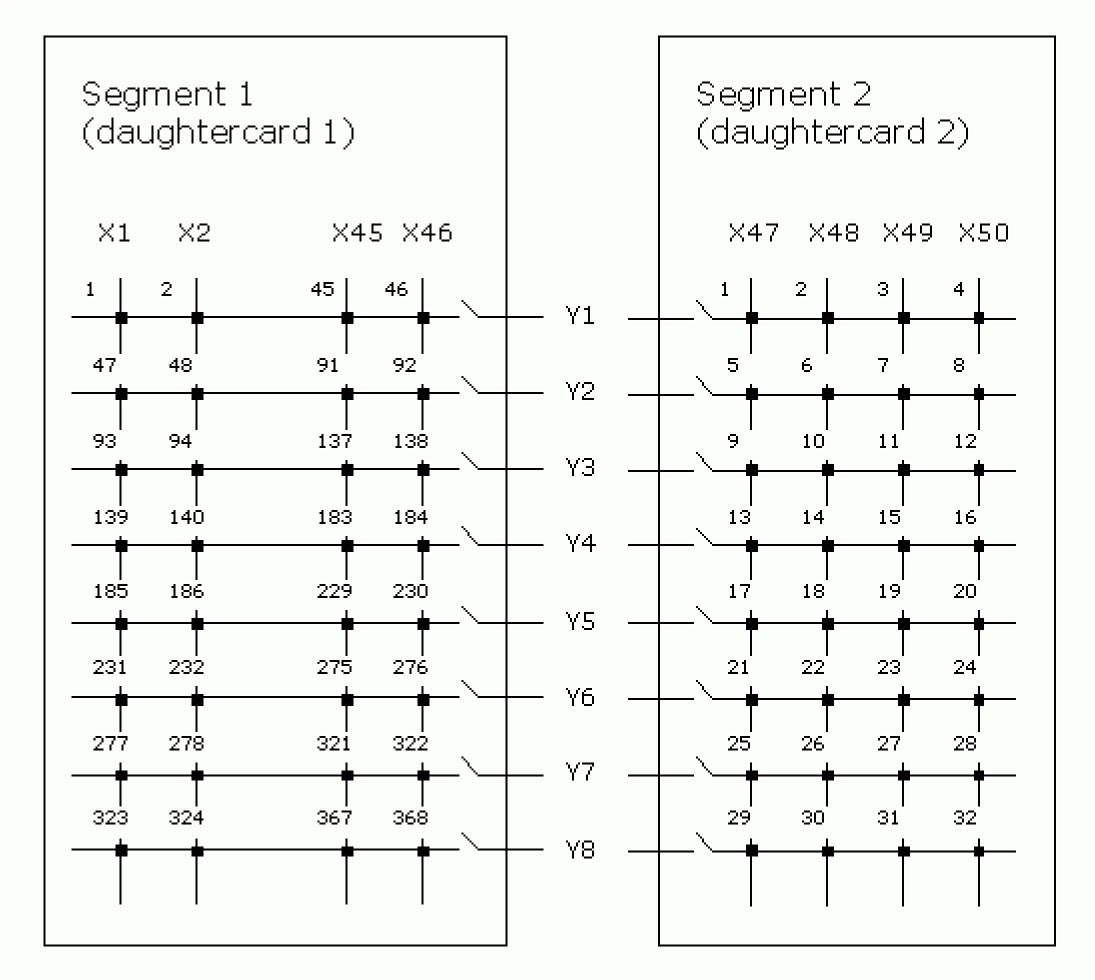
- SwitchFunc = SW_FUNC_CHANNEL
- SegNum = 0



Segment-local crosspoint switch numbers

These switch numbers are valid for PIL_OpSwitch when:

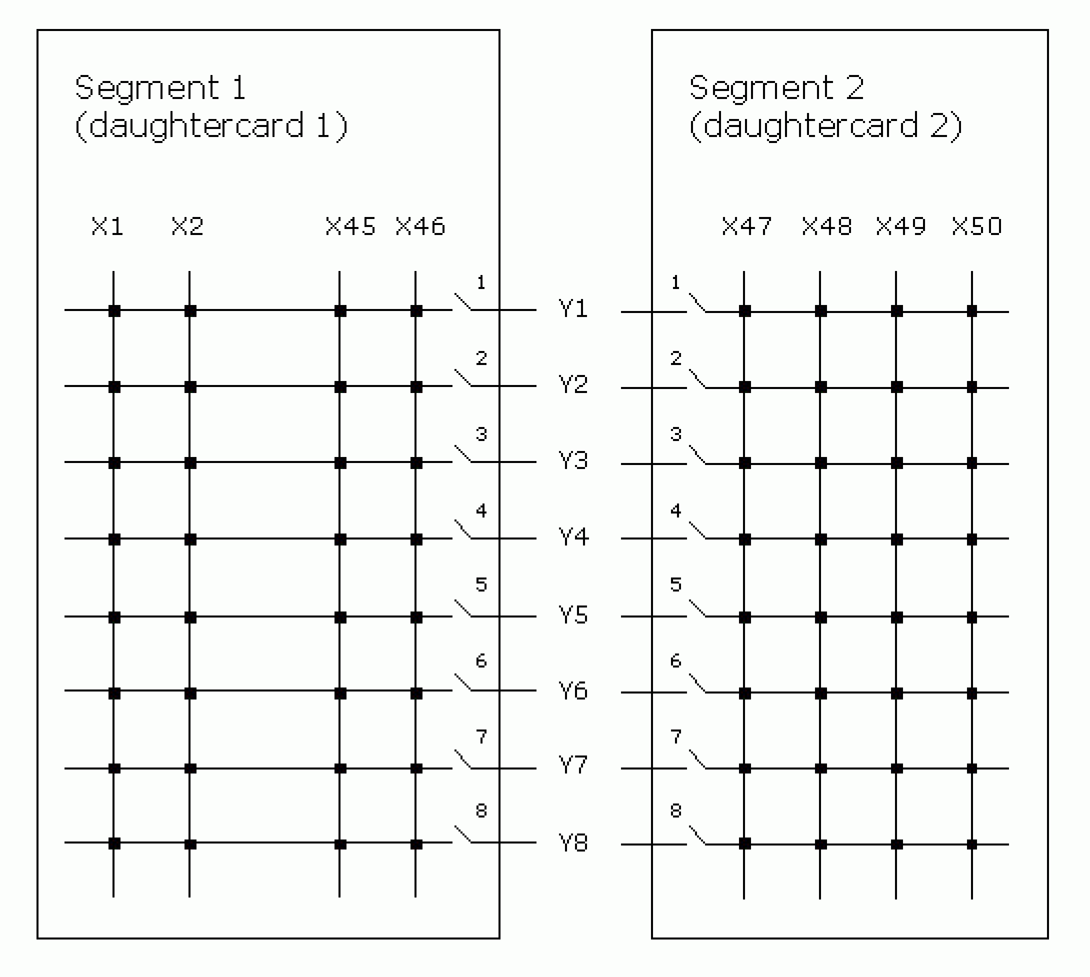
- SwitchFunc = SW_FUNC_CHANNEL
- SegNum = 1 or 2



Isolation switch numbers

These switch numbers are valid for PIL_OpSwitch when:

- SwitchFunc = SW_FUNC_Y_ISO
- SegNum = 1 or 2


Segment 1
(daughtercard 1)
X1 X2 X45 X46
Segment 2
(daughtercard 2)
X47 X48 X49 X50
1
2
3
4
5
6
7
8
Y1
Y2
Y3
Y4
Y5
Y6
Y7
Y8

Unsegmented Matrix

An unsegmented matrix is one in which all lines on an axis are served by a single set of isolation switches on the opposing axis.

Examples:

- there is currently no real example of this configuration

Index

Pickering Interfaces PXI

Register-level Programming Manual

Version date: 19 May 2006

Purpose

This manual describes the general principles of register-level operation of Pickering Interfaces PXI Switching cards in the System 40, System 45 and System 50 (PCI) ranges. It is also applicable to certain models in the System 41 PXI Instrument range.

Supplementary Information

For information on PCIbus operation, and its CompactPCI and PXI implementations, consult the relevant standard documents.

Details of a particular card's register assignments and operational characteristics can be found in its register-level datasheet.

Datasheets for the EEPROM and I/O devices employed in cards may also be required.

Common Elements

PCIbus Identification
Pickering Interfaces PXI cards are identified using PCIbus Subsystem IDs in conformance with PCIbus specification version 2.2.

The PCI interface of some models is implemented in an FPGA device, whose ID values in the card's configuration space are:

Address Offset	ID	16-bit Value
0000h	Vendor ID	1761h
0002h	Device ID	4411h
002Ch	Subsystem Vendor ID	1761h
002Eh	Subsystem ID	card-specific

All models bearing these IDs employ the SERIALFPGA architecture - see below.

In other models the Vendor ID and Device ID values are those assigned by the manufacturer of the PCI interface chip. The Subsystem Vendor ID identifies the card vendor as Pickering Interfaces; the Subsystem ID identifies the specific card type. Card identification and other PCIbus characteristics are set by an onboard EEPROM device dedicated to this purpose.

The ID values are located at the following offsets in the card's configuration space:

Address Offset	ID	16-bit Value
0000h	Vendor ID	10B5h
0002h	Device ID	9050h or 9030h
002Ch	Subsystem Vendor ID	1761h
002Eh	Subsystem ID	card-specific

Note that the card-specific Subsystem ID identifies a particular card model, but does not necessarily indicate its precise functionality (for example model 40-630-022 is identified, but not its configuration as a single or dual multiplexer, or its channel count).

Pickering cards designed for the PLX PCI9050 chip may instead be fitted with PCI9052. This chip's PCI IDs are identical to those of the PCI9050, except for its Revision ID (offset 0008h, bits 7:0) being altered from 01h to 02h; the two devices are functionally equivalent in Pickering designs.

A legacy ID scheme exists in which cards of different types share a common Subsystem ID:

Address Offset	ID	16-bit Value
0000h	Vendor ID	10B5h
0002h	Device ID	9050h
002Ch	Subsystem Vendor ID	10B5h
002Eh	Subsystem ID	1150h

The legacy scheme is no longer used, and should only be found in cards manufactured prior to mid-2001. The specific type of a card bearing these legacy IDs can only be determined by interrogating the card's data EEPROM. Such cards can be updated to the current scheme on request. Pickering software drivers continue to support both schemes.

Address Spaces
Cards carrying a PCI9050, PCI9052 or PCI9030 device utilise a 128-byte memory window at PCI BAR0. This space is claimed by the PCI interface chip, allowing access to its own internal registers. There should be no need to access this area in normal operation. If the contents of the chip's internal registers are of any interest please consult the PLX PCI9050, PCI9052 or PCI9030 data book.

PCI BAR1 is unused in all cards. A PCI9050, PCI9052 or PCI9030 interface chip only allows its use as an I/O-mapped image of its internal registers, which is of little use in modern systems.

All cards claim a memory window at PCI BAR2, corresponding to the card's Local Address Space 0 (LAS0). Some cards claim additional memory windows at PCI BAR3 thru PCI BAR5 as necessary, corresponding to Local Address Spaces LAS1 thru LAS3, i.e.
PCI BAR2 = LAS0
PCI BAR3 = LAS1
PCI BAR4 = LAS2
PCI BAR5 = LAS3

Card Data EEPROM
All cards have additional onboard EEPROM memory. Data held in this EEPROM describes the card's characteristics to the Pickering software drivers.

In some models such as 40-290 programmable resistor cards the data EEPROM also provides non-volatile storage of user calibration values.

Status and Control Registers
All cards have a read/write register located at offset 0 in the card's LAS0 space. The read register is designated the Status Register (SR), and the write register is designated the Control Register (CR). The minimum implementation of these registers is their 8 least-significant bits. Higher-order bits are implemented as necessary in specific designs.

Card Reset State
A PCIbus reset condition causes:
- all relays to be turned OFF
- all TTL digital outputs to go low
- all open-collector digital outputs to open

Card Architectures
Three basic card architectures are in use: parallel, serial, and SERIALFPGA. These architectures are described below.

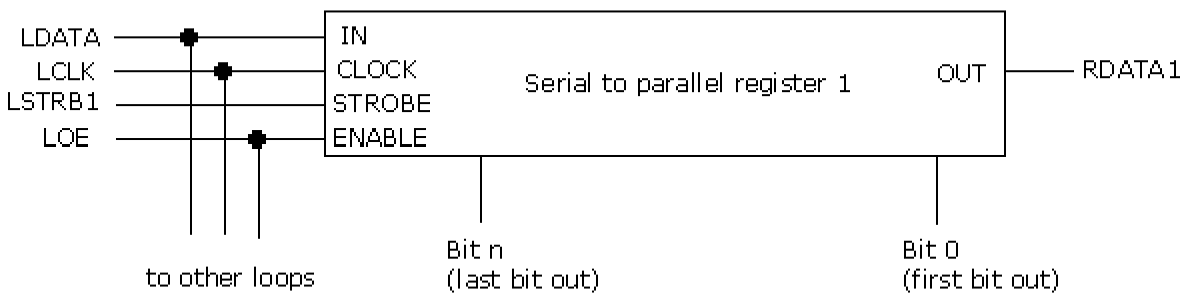
Parallel Architecture

In parallel architecture, I/O is performed by accessing parallel read (for input) and/or write (for output) registers upto 32-bits wide located at offset 0 in LAS1 space, and similar registers at offset 0 in higher-order spaces where necessary.

In some parallel cards there is a straightforward association between a register bit and the corresponding I/O function, e.g.
Bit 0 = channel 1
Bit 1 = channel 2 etc.
In other cases register bits do not correspond directly with their I/O function, and a lookup table must be incorporated in the driver.

Serial Architecture

In serial architecture, I/O is performed using parallel-serial registers (for input) or serial-parallel registers (for output) whose control signals are operated by bits in the Status and Control registers.



The bits associated with serial I/O control for a card having a single serial register are:
CR Bit 1 = LCLK (serial I/O clock, shared with EEPROM)
CR Bit 2 = LDATA (serial I/O data, shared with EEPROM)
CR Bit 3 = LSTRB1 (I/O loop #1 strobe)
CR Bit 4 = LOE (Output enable, all output loops)
SR Bit 1 = RDATA1 (I/O loop #1 receive data)

Cards having more than one serial I/O register implement additional LSTRB, RDATA signals. See the CR and SR bit assignment tables.

The output of each serial register is made available in its RDATA bit. For an input function this is essential, allowing the input data to be read. For an output function it facilitates a measure of self-test, allowing the integrity of the register to be confirmed. Although hardware failure is quite unlikely, it does at least allow a failed interconnect to be detected.

If necessary CR signals are inverted in hardware to match the behaviour of a particular I/O device: the LOE signal is sometimes affected, to suit devices having an active-low enable signal. From a software viewpoint, LOE is always active-high.

Refer to the applicable device datasheet for details of serial I/O register operation.

In general serial register bits do not correspond directly with their I/O function, and a lookup table must be incorporated in the driver.

Enabling Outputs
At RESET, the outputs of serial architecture cards are disabled. After clearing output registers and strobing this data to the outputs, LOE (CR bit 4) must be taken high to enable them.

Note
Present-generation PCs are capable of applying a software-generated clock rate that exceeds the specification of I/O hardware in some models. Appropriate measures must be taken to ensure that this does not occur. Bear in mind that PC speeds can only be expected to increase further in future.

Note
It should not be assumed that I/O devices can be clocked to the limit of their specifications, because circuit constraints may impose a lower limit. Consult the card's register-level data sheet for this specification.

SERIALFPGA architecture

This architecture employs storage registers similar to those in the software-driven serial implementation above; however they are accessed via an FPGA that provides a memory-mapped image into which (for outputs) desired data patterns can be written, and the FPGA then instructed to perform the serial transfer to the storage registers. This frees the CPU from the intensive processing required by the software-driven method.

Register-level operation of this architecture is currently beyond the scope of this manual.

Accessing the Data EEPROM

The number and type of serial EEPROM devices fitted depends on the requirements of a particular card. Low-density cards employ a single 93C56, 93C66 or 93C86 device. High-density models employ one or more 93C86 devices. Use of 93C86 EEPROM (which has a different instruction length) is flagged in bit 5 of the card's Status Register:
SR Bit 5 = '0': EEPROM type 93C56 or 93C66
SR Bit 5 = '1': EEPROM type 93C86

EEPROM is accessed by generating appropriate serial bitstream data on the associated Control Register bits:
CR Bit 0 = EEPROMCS0 - primary EEPROM chip-select
CR Bit 1 = LCLK (clock, shared with serial I/O)
CR Bit 2 = LDATA (data, shared with serial I/O)
and the output bitstream is obtained on:
SR Bit 0 = EEPROMDO (EEPROM output bit)

Where more than one EEPROM is employed they are enabled using additional chip-select signals supplied by further Control Register bits. See the CR bit assignment table. Output from the enabled EEPROM is obtained on SR bit 0.

Refer to the applicable device datasheet for details of EEPROM operation. EEPROMs are configured for 8-bit operation.

Note
Where multiple EEPROMs are employed, only one EEPROM may be enabled at any time.

Note
Present-generation PCs are capable of applying a software-generated clock rate that exceeds the specification of the EEPROM device. Appropriate measures must be taken to ensure that this does not occur. Bear in mind that PC speeds can only be expected to increase further in future.

Note
It should not be assumed that EEPROM devices can be clocked to the limit of their specifications, because circuit constraints may impose a lower limit. Consult the card's register-level data sheet for this specification.

Note
Writing arbitrary values in the EEPROM area containing Pickering configuration data will render a card inoperable by the Pickering software drivers.

EEPROM Configuration Data

Full interpretation and usage of the card configuration data held in EEPROM is beyond the present scope of this manual, and it is not expected that a register-level user will attempt it. Whereas Pickering software drivers are capable of operating the entire range of cards, it is likely that a user's register-level driver will need to handle at most a few different models, whose characteristics can be embedded in, or otherwise supplied to, the driver. The only significant drawback with this method is that the driver cannot automatically accommodate future card revisions.

Some basic card information can however be obtained from the first few EEPROM locations:

Byte Offset	Interpretation
0	Byte 1 (MSB) of 16-bit card model code
1	Byte 0 of card model code (e.g. 40-**632**-021)
2	Byte 1 (MSB) of 16-bit card variant code
3	Byte 0 of card variant code (e.g. 40-632-**021**)
4	Card minor variant code character (e.g. 40-632-021-**S**)
5	Byte 3 (MSB) of 32-bit card serial number
6	Byte 2 of card serial number
7	Byte 1 of card serial number
8	Byte 0 of card serial number
9	Byte 1 (MSB) of 16-bit card revision number
10	Byte 0 of card revision number (100 = version 1.00)
11	Byte 1 (MSB) of 16-bit minimum driver version number
12	Byte 0 of minimum driver version (100 = version 1.00)
13	Reserved

Byte Offset	Interpretation
14	Architecture code (1 or 2 = parallel; 3 = serial; 4 = SERIALFPGA)
15	Series number (0 = System 40, other value = series number, e.g. **50**-125-121)
16	Reserved
17	Reserved
18	Number of input sub-units
19	Number of output sub-units, only if number of input sub-units is zero

Configuration data locations above this contain values describing the card's physical and logical configuration in greater detail, and their interpretation is less straightforward.

Card Minor Variant Code
A null value indicates that the card has no minor variant suffix. A non-zero code is customarily interpreted as an ASCII character. A value not corresponding to a printable ASCII character may be interpreted in other ways, though no such values are currently in use.

Card Revision Number
This number will increase if any significant revision is made to the card's hardware.

Minimum Driver Version
This number indicates the earliest version of the Pickering software driver by which the card can be operated.

Calibration Data
Where supported, user calibration data is stored in EEPROM locations above those used for Pickering configuration data. Its position will be specified in the card's register-level data sheet. The interpretation of calibration values is entirely user-specific - Pickering software drivers simply provide support for storing and retrieving them.

Status Register Bit Assignments

Bit	Label	Function
0	EEPROMDO	EEPROM Data Out
1	RDATA1	Receive Data, I/O loop 1
2	RDATA2	Receive Data, I/O loop 2
3	RDATA3	Receive Data, I/O loop 3
4	RDATA4	Receive Data, I/O loop 4
5	EEPROMSZ	'0' for 93C56/93C66, '1' for 93C76/93C86
6	-	Reserved, read as '0'
7	-	Reserved, read as '0'
8	RDATA5	Receive Data, I/O loop 5
9	RDATA6	Receive Data, I/O loop 6
10	RDATA7	Receive Data, I/O loop 7
11	RDATA8	Receive Data, I/O loop 8

Bit	Label	Function
12	RDATA9	Receive Data, I/O loop 9
13	RDATA10	Receive Data, I/O loop 10
14	RDATA11	Receive Data, I/O loop 11
15	RDATA12	Receive Data, I/O loop 12
16	RDATA13	Receive Data, I/O loop 13
17	RDATA14	Receive Data, I/O loop 14
18	RDATA15	Receive Data, I/O loop 15
19	RDATA16	Receive Data, I/O loop 16
20	-	Usage undefined
21	-	Usage undefined
22	-	Usage undefined
23	-	Usage undefined
24	-	Usage undefined
25	-	Usage undefined
26	-	Usage undefined
27	-	Usage undefined
28	-	Usage undefined
29	-	Usage undefined
30	-	Usage undefined
31	-	Usage undefined

Control Register Bit Assignments

Bit	Label	Function
0	EEPROMCS0	Primary EEPROM chip-select
1	LCLK	Serial clock, EEPROM and all I/O loops
2	LDATA	Transmit data, EEPROM and all I/O loops
3	LSTRB1	I/O strobe, loop 1
4	LOE	Output enable, all output loops
5	LSTRB2	I/O strobe, loop 2
6	LSTRB3	I/O strobe, loop 3
7	LSTRB4	I/O strobe, loop 4
8	EEPROMCS1	Auxiliary EEPROM #1 chip-select
9	EEPROMCS2	Auxiliary EEPROM #2 chip-select
10	EEPROMCS3	Auxiliary EEPROM #3 chip-select
11	EEPROMCS4	Auxiliary EEPROM #4 chip-select
12	EEPROMCS5	Auxiliary EEPROM #5 chip-select
13	LSTRB5	I/O strobe, loop 5
14	EEPROMCS6	Auxiliary EEPROM #6 chip-select
15	LSTRB6	I/O strobe, loop 6

Bit	Label	Function
16	EEPROMCS7	Auxiliary EEPROM #7 chip-select
17	LSTRB7	I/O strobe, loop 7
18	EEPROMCS8	Auxiliary EEPROM #8 chip-select
19	LSTRB8	I/O strobe, loop 8
20	EEPROMCS9	Auxiliary EEPROM #9 chip-select
21	LSTRB9	I/O strobe, loop 9
22	EEPROMCS10	Auxiliary EEPROM #10 chip-select
23	LSTRB10	I/O strobe, loop 10
24	EEPROMCS11	Auxiliary EEPROM #11 chip-select
25	LSTRB11	I/O strobe, loop 11
26	EEPROMCS12	Auxiliary EEPROM #12 chip-select
27	LSTRB12	I/O strobe, loop 12
28	LSTRB13	I/O strobe, loop 13
29	LSTRB14	I/O strobe, loop 14
30	LSTRB15	I/O strobe, loop 15
31	LSTRB16	I/O strobe, loop 16

All Control Register bits are cleared at RESET.

Operational Considerations

Readback Capability
The output function of parallel and serial architecture cards is write-only; they have **NO** readback capability (i.e. the card's current output states are not readable by software). A software driver must maintain a soft-copy of the card's output states. By implication, when a software driver takes control of a card it must assume its state to be undefined. The normal course of action on taking control of a card would be to clear its outputs.

Cards using SERIALFPGA architecture do have readback capability.

Debounce Timing
Cards using the parallel and serial architectures have no onboard timer. Timing of switch settling periods can be done with reasonable accuracy using the system's performance counter, accessible using the Windows functions QueryPerformanceFrequency and QueryPerformanceCounter.

Cards using SERIALFPGA architecture have a programmable timer onboard. At present its condition can only be discovered by software polling.

Maximising Switch Life
All forms of metal-to-metal contact switch are subject to wear-out, but the behaviour of a software driver can have a big influence on their useful life. With the speed of the modern PC a software driver can manage switch operations intelligently with negligible time overhead. Stating the obvious, if a switch is operated twice when it need only operate once, it will wear out in half the time. As an example, when performing a channel change on a (single-channel) multiplexer with isolation switching, there is no need to break and re-make the isolation switch - saving the time and wear involved in cycling it. For such a device (and purely from a switching viewpoint), this is a drawback of drivers that force explicit disconnection of a selected channel before allowing the selection of a different one.

Isolation Switching
Where a card supports isolation switching, operation should ensure that the isolation switches **cold-switch**, in order to avoid concentrating contact wear on them and causing possible premature switch failure. This can be achieved by:
- when closing a channel, first close the isolation switch, then the channel switch
- when opening a channel, first open the channel switch, then the isolation switch

Break-Before-Make Action (BBM)
BBM is generally the safest mode of switch operation. However significant time savings are often possible by overlapping non-conflicting switch changes, particularly when using slow-operating switches such as microwave types. Make-Before-Break action may be desirable in some applications.

Default Channel Selection
Some RF and optical multiplexer units have no "disconnect" state: one channel remains connected even when all switches are turned OFF. A driver for such types should take account of this and report the card's state accordingly.

VISA Operation
The interpretation of cards' Vendor, Device and Subsystem IDs by different VISA implementations and/or revisions has not been consistent. To ensure compatibility with different VISA releases it is advised that VISA attributes such as:
- VI_ATTR_MANF_ID
- VI_ATTR_MODEL_CODE
- VI_ATTR_MANF_NAME
- VI_ATTR_MODEL_NAME
- VI_ATTR_PXI_SUB_MANF_ID (obsolescent in NI-VISA 3.0)
- VI_ATTR_PXI_SUB_MODEL_CODE (obsolescent in NI-VISA 3.0)

should **not** be used for card identification purposes. Instead the Vendor, Device and Subsystem IDs should be read directly from the card's Configuration Space (VI_PXI_CFG_SPACE) using the viIn16 function.

VI_PXI_CFG_SPACE offset	Attribute
0x00	Vendor ID
0x02	Device ID
0x2C	Subsystem Vendor ID
0x2E	Subsystem ID

NI-VISA 3.0 and later versions support the inclusion of Subsystem IDs in a card's VISA registration, allowing cards using Subsystem IDs to be properly distinguished in the VISA environment.

Operation of cards that employ software-driven serial architecture can be speeded up considerably by using VISA Low Level Access (LLA) in place of High Level Access (HLA).

Operational Warnings

Microwave Multiplexer Devices
In order to avoid overheating it is essential that no more than one channel of an individual multiplexer unit is energised at any time. Board GPIB269R0 has an onboard fuse to protect against energisation of a hugely excessive number of channels, but it cannot protect an individual switch unit from overheating.

High-density Matrix Devices
In order to avoid overheating it is essential that the specified maximum number of simultaneous crosspoint closures on such devices is not exceeded. Some matrix cards have

fuse protection against energisation of a hugely excessive number of crosspoints, but it may not protect against overheating if too large a contiguous block of relays is energised. Such a situation would not be expected in normal usage of these units, and is more likely to occur as a result of a programming error.

In many units it would be acceptable to energise larger numbers of relays that are not physically adjacent to one another, but this is difficult to manage in practice since the physical layout and ventilation characteristics of the unit become critical factors. Seemingly illogical operational constraints might well result.

Operational Issues

Inefficient use of system resources
The use of multiple memory spaces in parallel architecture cards is wasteful, and unjustified for cards of such simple functionality. The reason for this is historical.

In some architectures the size of memory spaces claimed is larger than strictly necessary. This originated as a workaround for a problem in the Device Manager of Windows 95 (RIP). In practice the size claimed is very unlikely to cause a memory allocation problem.

In cards using software-driven serial architecture, software clocking can place quite heavy demands on CPU time. This is more likely to be a concern in very high-density units.

Readback capability
Cards using the parallel and serial architectures have no readback capability. This presents problems if it is required to re-open cards (perhaps using a different application) with their previously-set output states intact. A software solution is usually possible, but may be quite cumbersome. Providing readback would require additional hardware, using up PCB real-estate and reducing a card's switching functionality.

Readback is supported in the SERIALFPGA architecture.

Software clock timing
With the increased speed of modern PCs, the frequency of a clock signal generated in software can easily exceed the specification of onboard hardware devices. Some form of timing control therefore becomes necessary.

Historically, the performance counter accessible through the Windows API function QueryPerformanceCounter had a frequency of about 1.2MHz, with a typical access time of around 5uS. In later machines the frequency was increased to 3.6MHz. This still did not provide sufficient resolution for optimal control of software-generated clock signals of 1MHz or above, and to obtain adequate performance it has sometimes been necessary to resort to software delay loops. However in some recent systems the performance counter frequency is increased to the full CPU clock rate and with faster CPUs also reducing access time, it is much better for this purpose. The function QueryPerformanceFrequency allows this value to be read in the target system. The time penalty for sub-optimal operation is obviously greatest in high-density units.

In some chipsets the Windows performance counter may exhibit erratic behaviour (ref. Microsoft Knowledge Base Article ID 274323).

Asynchronous operation
Cards using the parallel and serial architectures have no onboard timer or interrupt capability, so true asynchronous operation with callback on completion of switch operation is not available.

In SERIALFPGA architecture a programmable timer is available, but interrupt generation is not currently supported; it may be added in future.

PXI trigger functions
Trigger functionality is unavailable in cards using the parallel and serial architectures.

Trigger functionality is not currently available in SERIALFPGA architecture, but may be added in future.

33MHz/66MHz operation
PXI specifications have defined operation at 33MHz using 5V signalling, with cards being keyed accordingly (brilliant blue key): Pickering cards are keyed in this way. However there has been confusion over implementation of the M66EN signal in PXI cards and chassis. If either a card or chassis segment is incapable of 66MHz operation it should ground M66EN, but this has not been followed in all cases. Some Pickering cards and 8-slot chassis are affected, as well as some from other vendors. As a result, a bus controller that is 66MHz-capable may attempt to operate a segment or cards that are not 66MHz capable at that speed, usually leading to erratic behaviour. Where this occurs the preferred workaround, if the option exists, is to configure the bus controller for fixed operation at 33MHz. When the controller does not have this facility the cards and/or chassis can be modified to correct the problem.

The 66MHZ_CAPABLE bit in the PCI Status Register of Pickering cards is correctly implemented (value '0', not capable), but operating systems generally seem to ignore it.

Additional Support

For further assistance, please contact:

Pickering Interfaces Ltd.
Stephenson Road
Clacton-on-Sea
Essex
CO15 4NL
UK

Telephone: 44 (0)1255 687900
Fax: 44 (0)1255 425349

Regional contact details are available from our website: http://www.pickeringtest.com

Email (sales): sales@pickeringtest.com
Email (technical support): support@pickeringtest.com

Other Sources of Information

PCI Special Interest Group (PCI-SIG): http://www.pcisig.com
PLX Technology, Inc.: http://www.plxtech.com
PCI Industrial Computer Manufacturers Group (PICMG): http://www.picmg.com
PXI Systems Alliance (PXISA): http://www.pxisa.org
VXIplug&play Systems Alliance: http://www.vxipnp.org