Tektronix

Models 707B and 708B

Switching System

REFERENCE MANUAL

707B-901-01C January 2025

KEITHLEY

A Tektronix Company

Models 707B and 708B

Switching System

Reference Manual

Document number: 707B-901-01 Rev. C January 2025

KEITHLEY
A Tektronix Company

Safety precautions

The following safety precautions should be observed before using this product and any associated instrumentation. Although some instruments and accessories would normally be used with nonhazardous voltages, there are situations where hazardous conditions may be present.

This product is intended for use by personnel who recognize shock hazards and are familiar with the safety precautions required to avoid possible injury. Read and follow all installation, operation, and maintenance information carefully before using the product. Refer to the user documentation for complete product specifications.

If the product is used in a manner not specified, the protection provided by the product warranty may be impaired.

The types of product users are:

Responsible body is the individual or group responsible for the use and maintenance of equipment, for ensuring that the equipment is operated within its specifications and operating limits, and for ensuring that operators are adequately trained.

Operators use the product for its intended function. They must be trained in electrical safety procedures and proper use of the instrument. They must be protected from electric shock and contact with hazardous live circuits.

Maintenance personnel perform routine procedures on the product to keep it operating properly, for example, setting the line voltage or replacing consumable materials. Maintenance procedures are described in the user documentation. The procedures explicitly state if the operator may perform them. Otherwise, they should be performed only by service personnel.

Service personnel are trained to work on live circuits, perform safe installations, and repair products. Only properly trained service personnel may perform installation and service procedures.

Keithley products are designed for use with electrical signals that are measurement, control, and data I/O connections, with low transient overvoltages, and must not be directly connected to mains voltage or to voltage sources with high transient overvoltages. Measurement Category II (as referenced in IEC 60664) connections require protection for high transient overvoltages often associated with local AC mains connections. Certain Keithley measuring instruments may be connected to mains. These instruments will be marked as category II or higher.

Unless explicitly allowed in the specifications, operating manual, and instrument labels, do not connect any instrument to mains.

Exercise extreme caution when a shock hazard is present. Lethal voltage may be present on cable connector jacks or test fixtures. The American National Standards Institute (ANSI) states that a shock hazard exists when voltage levels greater than 30 V RMS, 42.4 V peak, or 60 VDC are present. A good safety practice is to expect that hazardous voltage is present in any unknown circuit before measuring.

Operators of this product must be protected from electric shock at all times. The responsible body must ensure that operators are prevented access and/or insulated from every connection point. In some cases, connections must be exposed to potential human contact. Product operators in these circumstances must be trained to protect themselves from the risk of electric shock. If the circuit is capable of operating at or above 1000 V, no conductive part of the circuit may be exposed.

Do not connect switching cards directly to unlimited power circuits. They are intended to be used with impedance-limited sources. NEVER connect switching cards directly to AC mains. When connecting sources to switching cards, install protective devices to limit fault current and voltage to the card.

Before operating an instrument, ensure that the line cord is connected to a properly-grounded power receptacle. Inspect the connecting cables, test leads, and jumpers for possible wear, cracks, or breaks before each use.

When installing equipment where access to the main power cord is restricted, such as rack mounting, a separate main input power disconnect device must be provided in close proximity to the equipment and within easy reach of the operator.

For maximum safety, do not touch the product, test cables, or any other instruments while power is applied to the circuit under test. ALWAYS remove power from the entire test system and discharge any capacitors before connecting or disconnecting cables or jumpers, installing or removing switching cards, or making internal changes, such as installing or removing jumpers.

Do not touch any object that could provide a current path to the common side of the circuit under test or power line (earth) ground. Always make measurements with dry hands while standing on a dry, insulated surface capable of withstanding the voltage being measured.

For safety, instruments and accessories must be used in accordance with the operating instructions. If the instruments or accessories are used in a manner not specified in the operating instructions, the protection provided by the equipment may be impaired.

Do not exceed the maximum signal levels of the instruments and accessories. Maximum signal levels are defined in the specifications and operating information and shown on the instrument panels, test fixture panels, and switching cards.

When fuses are used in a product, replace with the same type and rating for continued protection against fire hazard.

Chassis connections must only be used as shield connections for measuring circuits, NOT as protective earth (safety ground) connections.

If you are using a test fixture, keep the lid closed while power is applied to the device under test. Safe operation requires the use of a lid interlock.

If a screw is present, connect it to protective earth (safety ground) using the wire recommended in the user documentation.

The symbol on an instrument means caution, risk of hazard. The user must refer to the operating instructions located in the user documentation in all cases where the symbol is marked on the instrument.

The symbol on an instrument means warning, risk of electric shock. Use standard safety precautions to avoid personal contact with these voltages.

The symbol on an instrument shows that the surface may be hot. Avoid personal contact to prevent burns.

The symbol indicates a connection terminal to the equipment frame.

If this (Hg) symbol is on a product, it indicates that mercury is present in the display lamp. Please note that the lamp must be properly disposed of according to federal, state, and local laws.

The **WARNING** heading in the user documentation explains hazards that might result in personal injury or death. Always read the associated information very carefully before performing the indicated procedure.

The **CAUTION** heading in the user documentation explains hazards that could damage the instrument. Such damage may invalidate the warranty.

The **CAUTION** heading with the symbol in the user documentation explains hazards that could result in moderate or minor injury or damage the instrument. Always read the associated information very carefully before performing the indicated procedure. Damage to the instrument may invalidate the warranty.

Instrumentation and accessories shall not be connected to humans.

Before performing any maintenance, disconnect the line cord and all test cables.

To maintain protection from electric shock and fire, replacement components in mains circuits — including the power transformer, test leads, and input jacks — must be purchased from Keithley. Standard fuses with applicable national safety approvals may be used if the rating and type are the same. The detachable mains power cord provided with the instrument may only be replaced with a similarly rated power cord. Other components that are not safety-related may be purchased from other suppliers as long as they are equivalent to the original component (note that selected parts should be purchased only through Keithley to maintain accuracy and functionality of the product). If you are unsure about the applicability of a replacement component, call a Keithley office for information.

Unless otherwise noted in product-specific literature, Keithley instruments are designed to operate indoors only, in the following environment: Altitude at or below 2,000 m (6,562 ft); temperature 0 °C to 50 °C (32 °F to 122 °F); and pollution degree 1 or 2.

To clean an instrument, use a cloth dampened with deionized water or mild, water-based cleaner. Clean the exterior of the instrument only. Do not apply cleaner directly to the instrument or allow liquids to enter or spill on the instrument. Products that consist of a circuit board with no case or chassis (e.g., a data acquisition board for installation into a computer) should never require cleaning if handled according to instructions. If the board becomes contaminated and operation is affected, the board should be returned to the factory for proper cleaning/servicing.

Safety precaution revision as of June 2018.

Table of contents

Section 1

Introduction

In this section:

Overview

The six-slot 707B and single-slot 708B Semiconductor Switch Matrix Mainframes are optimized for semiconductor test applications. These mainframes have features and capabilities that speed and simplify system integration and test development. You can control up to 576 matrix crosspoints in real time. Their large matrix format makes them well suited for semiconductor device characterization, wafer level reliability, parallel test, and modeling.

Models 707B and 708B applications

Unlike a standalone instrument, the Keithley 707B and 708B switching systems intelligently connect other instruments to multiple devices. This manual provides application examples that guide you through several common instrument-to-device switching scenarios. These applications include:

- Work with the Model 4200-SCS or 4200A-SCS (on page 8-1): Demonstrates how to connect the Keithley Model 4200 or 4200A Parameter Analyzer to a Model 707B or 708B. This enables the 4200-SCS or 4200A-SCS to automate measurements to multiple devices for DC, C-V, and pulse I-V characterization.
- Work with the Series 2600B (on page 9-1): Demonstrates how to use the Model 707B or 708B with a Keithley Series 2600B System SourceMeter™ instrument to perform an I-V sweep across a diode.
- Work with the Series 2400 (on page 10-1): Demonstrates the basic interaction of the Model 707B or 708B with a Keithley Model 2400 SourceMeter instrument, including how to use digital I/O triggering to synchronize source and measure actions between the Model 707B or 708B and the Model 2400. The concepts in these examples can be applied when using the Model 707B or 708B with other SCPI-based instruments.

Extended warranty

Additional years of warranty coverage are available on many products. These valuable contracts protect you from unbudgeted service expenses and provide additional years of protection at a fraction of the price of a repair. Extended warranties are available on new and existing products. Contact your local Tektronix office, sales partner, or distributor for details.

Contact information

If you have any questions after you review the information in this documentation, please contact your local Tektronix office, sales partner, or distributor. You can also call the Tektronix corporate headquarters (toll-free inside the U.S. and Canada only) at 1-800-833-9200. For worldwide contact numbers, visit tek.com/contact-tek.

Product documentation, drivers, and software

The 707B and 708B documentation is available for download as PDFs from tek.com/keithley. The documentation includes:

- **Quick Start Guide:** Provides unpacking instructions, card installation, connection basics, and basic operating information. Refer to the Quick Start Guide to take the steps needed to unpack, set up, and verify operation.
- **User's Manual:** Provides descriptions of installation, the front and rear panels, basic operating information, and maintenance and troubleshooting.
- **Reference Manual:** This manual. Includes advanced operation topics and application examples. Programmers looking for a command reference and users looking for an in-depth description of the way the instrument works (including troubleshooting and optimization) should refer to the Reference Manual.
- **Additional product information:** Product datasheets, product specifications, and accessories information.

Additional drivers and software are available for download from the Product and Downloads Support web page, including:

- **Keithley Test Script Builder (TSB):** A software tool that simplifies building and creating test scripts. It includes built-in color coding, debugging tools, and error handling.
- **Keithley I/O Layer:** Software that manages the communications between Keithley instrument drivers and software applications and the instrument.
- Drivers for NI™ LabVIEW™.
- Firmware updates.

To identify IP addresses of instruments connected to the local area network (LAN) that support VXI-11 discovery protocol, you can also use the LXI Discovery Tool, available from the Resources tab of the LXI Consortium website (lxistandard.org).

Display the serial number

The instrument serial number is on a label on the rear panel of the instrument. You can also access the serial number from the front panel using the front-panel keys and menus.

To display the serial number on the front panel:

1. If the Model 707B or 708B is in remote operation, press the **EXIT (LOCAL)** key once to place the instrument in local operation.
2. Press the **MENU** key.
3. Use the navigation wheel ◷ to scroll to the **UNIT-INFO** menu item.
4. Press the **ENTER** key. The UNIT INFORMATION menu is displayed.
5. Scroll to the **SERIAL#** menu item.
6. Press the **ENTER** key. The Model 707B or 708B serial number is displayed.

Section 2

Switch and scan

In this section:

Work with channels

The 707B and 708B have specific settings that open and close channels. It also contains information about pseudocards.

This section describes how to work with channels, including a discussion of channel types, channel selection, how to open and close channels, and how to set common channel attributes and channel patterns.

NOTE

To install the matrix card, refer to instructions in the *707B and 708B Quick Start Guide*.

CAUTION

Hot switching can dry weld reed relays such that they will always be on. Hot switching is recommended only when external protection is provided.

Specify a channel

The channels on the matrix cards that you can use with the Model 707B or 708B are referred to by a channel specifier. You use the specifier to identify channels for use with close and open operations, scans, and channel patterns. The specifier is used for all interfaces (front panel, web, and remote command).

A channel specifier is a four or five-digit alphanumeric sequence. The first digit is always the number of the slot in which the card is installed in the instrument. The remaining digits vary depending on the type of card.

The following sections describe the channel specifier in more detail and provide generic examples.

Channel types

The Models 707B and 708B support matrix cards with matrix channels. The documentation for your matrix card lists the available channel types.

Matrix card channel specifiers

The channels on the matrix cards are referred to by their slot, bank, row, and column numbers:

- **Slot number:** The number of the slot in which the card is installed.
- **Bank number:** The bank number, if used by your card. See your card documentation.
- **Row number:** The row number is either 1 to 8 or A to Z. See your card documentation.
- **Column number:** Always two digits. For columns greater than 99, use A, B, C and so on to represent 10, 11, 12, …; the resulting sequence is: 98, 99, A0, A1, …, A8, A9, B0, B1, …

Matrix card channel examples

Specifier	Slot number	Bank number	Row number	Column number
1A05 or 1105*	1	N/A	1	05
1C05	1	N/A	3	05
3C12	3	N/A	3	12

* Specifier depends on matrix card. See your matrix card documentation for detail.

Channel list parameter for remote commands

The channel list parameter is a string-type parameter that is used when controlling the relays of the Model 707B or 708B using a remote command interface. You can specify a list of individual channels or a range of channels in the channel list parameter.

In the command descriptions, the channel list parameter is shown as `channelList`.

When sending this parameter:

- Enclose the contents of the channel list in either single (') or double (") quotes. The beginning and end quotes must be the same type of quote.
- Use a comma or semicolon to separate the channel list or channel patterns (on page 2-14).
- The string may contain a single channel, single channel pattern, multiple channels, or multiple channel patterns that are comma-delimited.
- Use a colon between the start and end channel to specify a range of channels. The lowest channel must be first and the highest last.

Examples:

- To perform an open or close operation on row 1 and columns 3 and 5 of slot 1, use `("1A03, 1A05")` for the *`channelList`* parameter.
- To perform an open or close operation on all channels in the range of row 1 and columns 1 through 5 of slot 1, use `("1A01:1A05")` for the *`channelList`* parameter.

Queries that return a list of channels

For queries that return a channel list parameter, the Model 707B or 708B separates the channels by a comma or semicolon, depending on the command. When multiple channels are used in the query, the information for the lowest numbered channel is listed first, increasing to the highest numbered channel.

When multiple slots are used in the query, the information for the lowest slot number is listed first and increases to the highest slot and channel.

Connection methods for close operations

You can dictate the order in which relays are opened and closed using the channel connect rule.

WARNING

When the connection rule is set to break before make, the instrument ensures that all switch channels open before any switch channels close. This behavior covers the most common applications and is considered the safest connection rule because the tested device is completely decoupled from the instrument. This is the default behavior. When switch channels are both opened and closed, this command executes not less than the addition of both the open and close settle times of the indicated switch channels.

When the connection rule is set to make before break, the instrument ensures that all switch channels close before any switch channels open. This behavior should be applied with caution because it will connect two test devices together for the duration of the switch close settle time. When switch channels are both opened and closed, the command executes not less than the addition of both the open and close settle times of the indicated switch channels.

With no connection rule (set to `channel.OFF`), the instrument attempts to simultaneously open and close switch channels in order to minimize the command execution time. This results in faster performance at the expense of guaranteed switch position. During the operation, multiple switch channels may simultaneously be in the close position. Make sure your device under test can withstand this possible condition. When switch channels are both opened and closed, the command executes not less than the greater of either the open or close settle times of the indicated switch channels.

Cold switching is highly recommended.

CAUTION

Hot switching can dry weld reed relays such that they will always be on. Hot switching is recommended only when external protection is provided.

The channel connect rule determines the order in which multiple channels are opened and closed on the instrument. This attribute applies to electromechanical, reed, and solid-state relay switching cards.

You can set the channel connect rule to be:

- **BBM** (break before make): The instrument ensures that all switch channels open before any switch channels close. It is used to avoid momentary shorting of two voltage sources. This is the default.
- **MBB** (make before break): The instrument ensures that all switch channels close before any switch channels open. It is used to eliminate transients caused by switching between current sources. MBB should be applied with caution because it connects two test devices together for the duration of the switch close settle time.
- **OFF**: Permits the instrument to initiate close and open operations simultaneously. This minimizes settling time for the close operation.

NOTE

You cannot guarantee the sequence of open and closure operations when the channel connect rule is set to OFF. It is highly recommended that you implement cold switching when the channel connect rule is set to OFF.

To set the channel connect rule through the front-panel interface:

1. Press the **MENU** key.
2. Use the navigation wheel to scroll to the CHANNEL menu item.
3. Press the **ENTER** key (or the navigation wheel) to display the CONNECT MENU.
4. From this menu, select the **RULE** menu item.
5. Set the rule to **BBM**, **MBB**, or **OFF**.
6. Use the **ENTER** key to apply the selection.
7. Use the **EXIT** key to leave the menu.

To set the channel connect rule through the web interface:

1. On the Unit page, in the upper left corner, select the channel connect rule menu.
2. Select **Break Before Make**, **Make Before Break**, or **OFF**.

To set the channel connect rule through the remote command interface:

Use channel.connectrule (on page 7-19).

Set up a sequential connection

During normal operation, the instrument attempts to minimize the duration of any channel action for the card type and connect rule. This can result in multiple channels closing or opening simultaneously.

To prevent simultaneous closing and opening, you can use a sequential connection. A sequential connection ensures an orderly closing or opening of single channels in a channel list. An orderly action provides for:

- Repeatable and deterministic channel operating times
- Minimized power usage

Sequential channel connections help you better calculate timing. You incur settling times at each close or open operation. If sequential connection is not selected, action settling times may vary depending on the card type. The total settling time is the sum of the settling times for each specified channel, plus any user delays that have been set for any closed channels. For example, if you close three channels and each takes 4 ms to close, with sequential on, it takes 12 ms. With sequential off, it may be 4, 8 or 12 ms, depending on whether or not the card can close multiple channels at once.

Opening and closing relays in a sequential manner also uses minimum power. Since only one relay is closed or opened at any given time, the power used for that action is for a single relay and not additive.

By default, sequential connections are turned off. The order in which channels are opened or closed is not guaranteed. This feature also applies to scanning.

The sequential setting affects all channels in the instrument.

To enable sequential connections through the front-panel interface:

1. Press the **MENU** key.
2. Use the navigation wheel to scroll to the CHANNEL menu item.
3. Press the **ENTER** key.
4. Select the **SEQUENTIAL** menu item.
5. Select **ON** or **OFF**.
6. Use the **ENTER** key to apply the selection.
7. Use the **EXIT (LOCAL)** key to leave the menu.

To enable sequential connections through the web interface:

1. Open the **UNIT** page.
2. In the upper left corner, select Sequential (next to the Channel Connect Rule list).

To enable sequential connections through the remote command interface:

Send the command:

```
channel.connectsequential = channel.ON
```

Refer to channel.connectsequential (on page 7-21) for detail.

Determine the number of relay closures

The 707B and 708B keeps an internal count of the number of times each switch card relay has been closed. The total number of relay closures is stored in nonvolatile memory on the switch card. Use this count to determine when relays require replacement. Refer to the card documentation for information regarding the contact life specifications.

Relay closures are counted only when a relay transitions from open to closed state. If you send multiple close commands to the same channel without sending an open command, only the first closure is counted.

NOTE

The counts reported for the following cards indicate the number of closures since the last power cycle of the card:

- 7072
- 7072-HV
- 7173-50
- 7174A

For all other cards, the counts are the closures that have occurred over the lifetime of the card.

To view the close count for a channel from the front panel:

1. Use the navigation wheel ◷ to select the channel.
2. Press the **CONFIG** key.
3. Press the **CHAN** key.
4. Use the navigation wheel to scroll to the **COUNT** menu item.
5. Press the **ENTER** key (or the navigation wheel) to display the close counts.
6. Use the **EXIT** key to leave the menu.

To view the close count for a channel from the web interface:

1. From the list on the left, select a slot with an installed card.
2. Right-click a channel. The Channel Configuration dialog is displayed.
3. Check the value in Closure Count.

To view the close count through the remote command interface:

```
counts = channel.getcount("slot2")
print(counts)
```

Refer to channel.getcount() (on page 7-27) for more detail.

Identify installed switching cards

To identify installed matrix cards from the front panel:

Press the **SLOT** key to scroll through the model numbers, descriptions, and firmware revisions of the installed matrix cards.

To identify installed matrix cards from the web interface:

1. Select the **Unit** page.
2. In the Report area, select the slots that you want information about.
3. Select **Firmware Revision**.
4. Select **Generate Report**. Information about the cards in the slots is displayed below the button.

To identify installed matrix cards from the remote command interface:

Use slot[slot].idn (on page 7-171) to query and identify installed matrix cards.

For example, to get a list of all matrix cards installed in the slots of a Model 707B or 708B, send the following command over the remote command interface:

```
for x=1,6 do print (slot[x].idn) end
```

The response is similar to the following:

```
7174, 8x12 Fast Low-I Matrix, 01.00a, <Module Serial Number>
7072, 8x12 Semi Matrix, 01.00a, <Module Serial Number>
Empty Slot
Empty Slot
Empty Slot
Empty Slot
```

Close and open channels

If you have switching channels, you can close and open the relay channels using the front panel, web interface, or remote commands.

NOTE

You can use scans to perform a user-specified sequence of close and open operations on multiple channels. Refer to Scan and trigger (on page 2-21) for information on scan operations.

Operate a channel from the front panel

CAUTION

Hot switching can dry-weld reed relays, causing them to always be on. Hot switching is recommended only when external protection is provided.

You can perform operations on a single channel from the front panel.

To select a channel:

1. If the instrument is being controlled remotely, press **EXIT** to allow control from the front panel.
2. Turn the navigation wheel ◷ until the channel specifier is displayed on the front panel.
3. To:
   - Close a channel without affecting any other channels: Select **CLOSE**.
   - Open the channel: Press **OPEN**.
   - Close a channel and open any other closed channels on the instrument: Select **CHAN** and select **EXCLOSE**. Press **ENTER** to close the selected channel.
   - Close a channel and open any other closed channels on the slot that contains the selected channel: Select **CHAN**, and then select **EXSLOTCLOSE**. Press **ENTER** to close the selected channel.

NOTE

Once a channel is selected, it is the selected channel for any subsequent front-panel operations.

Open and close channels from the Channel Action Menu

You can use the options in the Channel Action Menu to open and close channels from the front panel.

To use the Channel Action Menu to open and close channels:

1. Go to channel view.
2. Select the channel you want to open or close.
3. Press **CHAN**.
4. Use the navigation wheel ◔ to select the option. To:
   - Close the selected channel: Select **CLOSE**.
   - Open the channel: Select **OPEN**.
   - Close a channel and open any other closed channels on the instrument: Select **EXCLOSE**.
   - Close the selected channel and open any other closed channels on the slot: Select **EXSLOTCLOSE**.
5. Press the navigation wheel ◔ to open or close the channel.

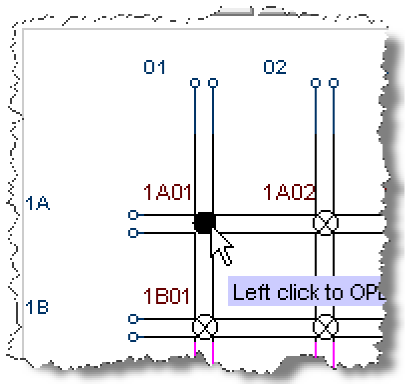
Select, close, and open a channel from the web interface

You can perform operations on a single channel from the web interface.

To select a channel:

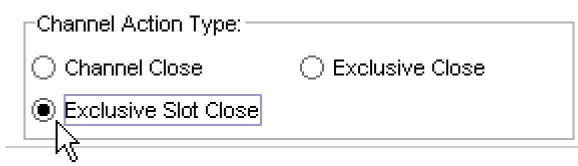
1. You must log into the instrument to work with the channels. See Log in to the instrument (on page 4-41). After logging in, you can access the channel controls.
2. From the instrument home page, from the navigation on the left, select the slot that contains the channels you want to work with.
3. To close a channel, select the channel. The display of the channel depends on the card that you have installed. An example is shown in the following figure.

Figure 1: Close a channel



4. To open a closed channel, select it again.
5. To perform an exclusive close on a channel:
   a. Select **Exclusive Slot Close** in Channel Action Type.

Figure 2: Select exclusive slot close



   b. Select a channel to close that channel and open all other channels.

Select, close, and open a channel using remote commands

You can open and close channels using the following commands:

channel.close() (on page 7-18)
channel.exclusiveclose() (on page 7-23)
channel.exclusiveslotclose() (on page 7-24)
channel.open() (on page 7-35)

For example, to close channel 1001 over the remote interface, send the command:

```
channel.close("1001")
```

View the close or open status of a channel

To determine whether a channel is closed or open, you can view its status using the front-panel interface, remote command query, or instrument web page.

View status from the front panel

Closed channels are shown on the display of the instrument, separated by commas. If more than one line of closed channels is displayed, you can press **DISPLAY** to display the full list. Use the navigation wheel to scroll through the list.

NOTE

For a four-pole operation, the paired channel is not displayed on the front panel of the 707B and 708B.

View status from the instrument web page

To view status from the instrument web page, from the list on the left, select the slot that contains the channel. The status is displayed on the web page for the slot.

View status from the remote command interface

To view a list of closed channels, use channel.getclose() (on page 7-26) command.
For example:

```
print(channel.getclose("allslots"))
```

To view the close and open status of channels, use the channel.getstate() (on page 7-33) command.

Channel attributes

You can use the front panel and command options to set attributes for specific channels. Some of the attributes you can set are adding a delay, forbidding closure of a channel, and setting channel labels, which are described in the following sections.

Set additional delay

You can set an additional delay to incur after the relay settles when closing. The total delay for channel operation is the user delay plus the relay settling time.

To set additional delay time from the front panel:

1. Display a channel (you might need to press **DISPLAY**).
2. Select the channel for which you want to set attributes.
3. Press **CONFIG**, then press **CHAN**.
4. Select **DELAY** and set the additional delay to incur after the relay settles. Enter the value for the delay in seconds.

To set additional delay time from the web interface:

1. From the list on the left, select the slot that contains the channel for which you want to set an additional delay.
2. Right-click the channel.
3. Enter the delay time (in seconds) in the delay time field on the right side of the dialog. Select **OK**.

To set additional delay time through the remote interface, use the command:

channel.setdelay() (on page 7-43)

Forbid closure of a channel

You can prevent closure of a channel from any interface by setting it to forbidden.

NOTE

If the channel that is to be forbidden is used in a channel pattern, the pattern is deleted when you set the channel to be forbidden to close.

To forbid closing of a channel from the front panel:

1. Display a channel (you might need to press **DISPLAY** first).
2. Select the channel for which you want to set attributes.
3. Press **CONFIG**, then press **CHAN**.
4. Use the navigation wheel ◷ to select **FORBID**.
5. Select **Yes** to prevent a channel from being closed or **No** to allow closure.
6. Press the navigation wheel ◷ to save the change.

To forbid closing of a channel from the web interface:

1. From the list on the left, select the slot that contains the channel.
2. Right-click the channel.
3. Select **forbidden**.
4. Select **OK**.

To forbid closing of a channel from the remote interface:

You can also set this attribute using the following commands:

- channel.setforbidden() (on page 7-44)
- channel.clearforbidden() (on page 7-17)

Set up labels

You can define labels for rows, columns, and channels.

Labels must be unique. They cannot have the same as the name of another row, column, channel, or channel pattern. Labels cannot contain spaces. They do not persist through a power cycle.

Channel labels can be up to 19 characters. Row and columns labels can be up to 8 characters. On the crosspoint display, the first four characters of the label are displayed. On the bottom display, the full label is displayed.

You can only set labels for slots and channels that are installed in the instrument.

To set up labels from the front panel:

1. Display a channel (you might need to press **DISPLAY** first).
2. Select the channel for which you want to set labels.
3. Press **CONFIG**, then press **CHAN**.
4. Use the navigation wheel ◷ to select the type of label you want to define:
   - **LABEL:** Sets the label that is displayed on the front panel for the specified crosspoint.
   - **LABEL-ROW:** Sets the label that is displayed on the front panel for the specified row.
   - **LABEL-COL:** Sets the label that is displayed on the front panel for the specified column.
5. Change the name using the navigation wheel ◷.
6. Press the navigation wheel ◷ to save the change.

To set up labels from the web interface:

1. From the list on the left, select the slot that contains the channel.
2. Right-click the channel.
3. In **Label**, enter the label.
4. Select **OK**.

To set up labels from the remote interface:

Use the commands:

- channel.setlabel() (on page 7-45)
- channel.setlabelrow() (on page 7-48)
- channel.setlabelcolumn() (on page 7-46)

You can use labels to refer to the channels in commands. For example, if you set the label for channel 3005 to `start`, you could use `start` to close and open the channel, as shown in the following example.

```
channel.setlabel("1A01","start")
channel.close("start")
channel.setlabelrow("1B01", "SMU2")
channel.setlabelcolumn("1B01", "DUT2")
channel.close("SMU2+DUT2")
print(channel.getclose("allslots"))
```

Channel patterns

You can use channel patterns to refer to a group of switching channels with a single alphanumeric name. When you perform close or open operations on a channel pattern, only the channels that are in the channel pattern are affected.

There is no speed difference when performing close and open operations on channel patterns compared to performing the same operations on individual channels or a list of channels.

NOTE

Channel patterns inherit the delay times of the individual channels that comprise the pattern. For information on the sequence of close operations on multiple channels, refer to Connection methods for close operations (on page 2-3).

When you create a channel pattern, make sure to:

- Include all of the channels that are needed for that channel pattern.
- Check that channels contained in the pattern are correct.
- Check that channels contained in the pattern create the correct path connection.
- Make sure that channels that you want to include in the pattern are not set to forbidden to close.

When naming the channel pattern, be aware:

- The first character of the name must be alphabetic (upper or lower case)
- Names are case sensitive
- Pattern names must be different than row, column, and channel labels

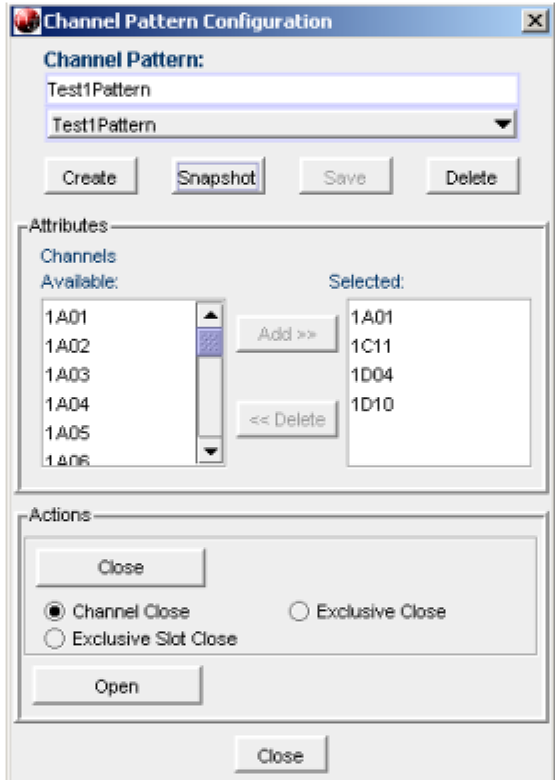
To create a channel pattern from the front panel:

1. Close the channels you want to include in the channel pattern.
2. Press the **PATT** key.
3. From this menu, select the **CREATE** menu item.
4. From this menu, select the **SNAPSHOT** menu item.
5. At the prompt, enter a pattern name using the navigation wheel ◷.
6. Use the **ENTER** key to apply the selection.
7. Use the **EXIT** key to leave the menu.

To create a channel pattern from the web interface:

1. From the left navigation, select a slot.
2. Select **Pattern** (above Channel Action Type). The Channel Pattern Configuration dialog is displayed.

Figure 3: Channel pattern configuration dialog



3. In **Channel Pattern**, enter a name.
4. From the Channels Available list, select the channels you want to add. You can use Ctrl+click and Shift+click to select multiple channels.
5. Select **Add**.
6. When the Selected channel list is complete, select **Create**.

NOTE

If you close the channels you want to add to the new pattern, you can enter a name and select **Snapshot** to create the new pattern. The closed channels are added to a new pattern.

To create a channel pattern from the remote command interface:

You can work with channel patterns using the following commands:

channel.pattern.catalog() (on page 7-36)
channel.pattern.delete() (on page 7-37)
channel.pattern.getimage() (on page 7-37)
channel.pattern.setimage() (on page 7-38)
channel.pattern.snapshot() (on page 7-40)

Perform close and open operations on channel patterns

WARNING

Careless channel pattern operation could create an electric shock hazard that could result in severe injury or death. Improper operation can also cause damage to the matrix cards and external circuitry. The control of multiple channels using channel patterns should be restricted to experienced test engineers who recognize the dangers associated with multiple channel closures.

You can close and open channel patterns the same way you do for individual channels.

To perform a particular operation on a channel pattern, use the appropriate open or close command with the channel pattern name for the *`channelList`* parameter.

When you request a close or open operation, the Model 707B or 708B verifies that the channels exist for a pattern, but does not verify that the switch path connection is correct. You must ensure the requested operation is safe for a channel pattern and that a good connection will result for your application with the channel pattern.

To close or open the channels in a channel pattern from the front panel:

1. Press the **PATT** key to display a channel pattern (you might need to press **DISPLAY** first).
2. Select the channel pattern you want to open or close.

NOTE

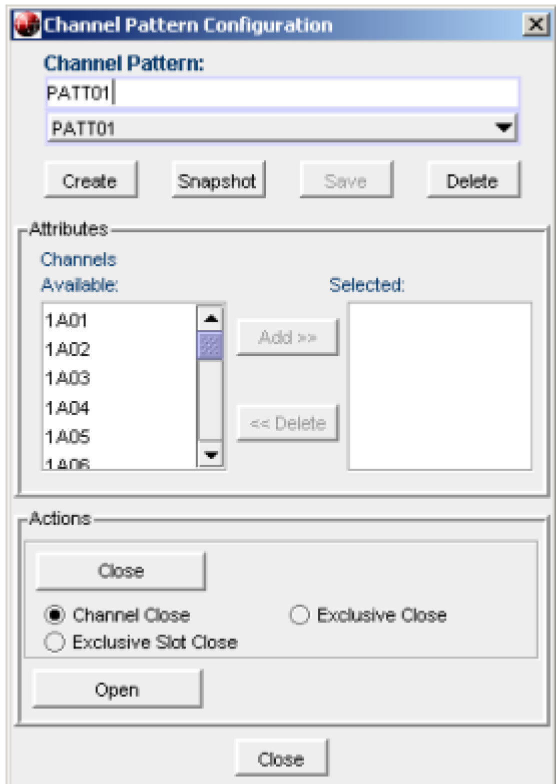
Model 707B only: Note that when you select a pattern, on the crosspoint display, the lights for the channels included in the pattern are dimly lit.

3. Perform any of the following actions:
   - **Open the channels in the channel pattern:** Press **OPEN**.
   - **Close the channels in the channel pattern without affecting any other channels:** Press **CLOSE**.
   - **Close the channels in the channel pattern and open any other closed channels on the instrument:** Select **PATT** and select EXCLOSE. Press **ENTER** to open or close the channels.
   - **Close the channels in the channel pattern and open any other closed channels on the slot:** Press **PATT** and select EXSLOTCLOSE. Press **ENTER** to open or close the channels.

To close or open the channels in a channel pattern from the web interface:

1. From the list on the left, select a slot with an installed card.
2. Select **Pattern**. The Channel Pattern Configuration dialog is displayed.

Figure 4: Channel Pattern Configuration dialog



3. Select the pattern.
4. In Actions, to:
   - **Open the channels in the channel pattern:** Select **OPEN**.
   - **Close the channels in the channel pattern without affecting any other channels:** Select Channel Close, then select **CLOSE**.
   - **Close the channels in the channel pattern and open any other closed channels on the instrument:** Select Exclusive Close and select **CLOSE**.
   - **Close the channels in the channel pattern and open any other closed channels on the slot:** Select Exclusive Slot Close and select **CLOSE**.

To close or open the channels in a channel pattern from the remote interface:

channel.close() (on page 7-18)
channel.exclusiveclose() (on page 7-23)
channel.exclusiveslotclose() (on page 7-24)
channel.open() (on page 7-35)

Channel pattern storage

Channel patterns are:

- Part of the script that is created with Create Config Script.
- Deleted when the instrument is reset.
- Deleted when a channel associated with the pattern is reset.
- Allocated 32KB of memory in the 707B and 708B instrument for all channel patterns.

To see how much of the channel pattern memory is available or used, send the command:

```
print(memory.available())
```

or

```
print(memory.used())
```

Refer to memory.available() (on page 7-135) or memory.used() (on page 7-136) for additional detail.

Reset a channel

You can reset a channel to its factory default settings. When you reset a channel:

- A closed channel opens
- Additional user delay is set to zero
- Labels return to default values
- Forbidden to close setting is cleared

Using this function to reset a channel involved in scanning invalidates the existing scan list. The list has to be recreated before scanning again.

CAUTION

Resetting a channel deletes any channel patterns that contain that channel.

To reset a channel from the front panel:

1. Display a channel.
2. Select the channel you want to reset.
3. Press **CHAN**.
4. Select **RESET**.
5. Select **SELECTED**, **ALL**, or **CANCEL**.
6. Press the navigation wheel ◷ to reset the channel.

To reset all channels on a slot from the web interface:

1. Select the slot that contains channels you want to reset.
2. Click **RESET SLOT**.
3. All channels on the slot are reset.

To reset a channel from the remote interface:

Send the command channel.reset() (on page 7-42).

Pseudocards

You can perform most open, close, and scan operations and configure your system without having an actual matrix card installed in your instrument. Using the remote interface, you can assign a pseudocard, allowing the instrument to operate as if a matrix card were installed.

NOTE

While most operations can be simulated with a pseudocard, some operations, such as channel delays, cannot be simulated.

A pseudocard cannot be configured from the front panel. However, once the remote configuration is complete, you can use the front panel to use the pseudocard. Press the **EXIT (LOCAL)** key to take the instrument out of remote mode.

When the instrument is turned off, the pseudocard is no longer assigned to the slot.

For example, send the following command to set slot 2 to use the 7072 8×12 Semiconductor Matrix pseudocard:

```
slot[2].pseudocard = 7072
```

Refer to slot[slot].pseudocard (on page 7-174) for additional detail.

Scan and trigger

A scan is a series of steps that opens and closes switches sequentially for a selected group of channels. During each step, actions occur, such as waiting for a trigger, making a measurement, and completing a step count. Scans automate actions that you want to perform consistently and repeatedly on a set of channels.

Triggers are events that prompt the instrument to move from one step to another in a scan. Triggers can come from different sources, such as a key press, digital input, or expiration of a timer. The sequence of actions and events that occur during the scan is called the trigger model.

Scanning and triggering allow you to synchronize actions across channels. You can set up a scan using the trigger model to precisely time and synchronize the Model 707B or 708B between channels and multiple instruments. You can also use triggers without the trigger model to set up a scan to meet the needs of a specific application that does not fit the trigger model.

Trigger model

When you run a scan, the scan sequence follows a trigger model. The trigger model is shown in the following flowchart.

NOTE

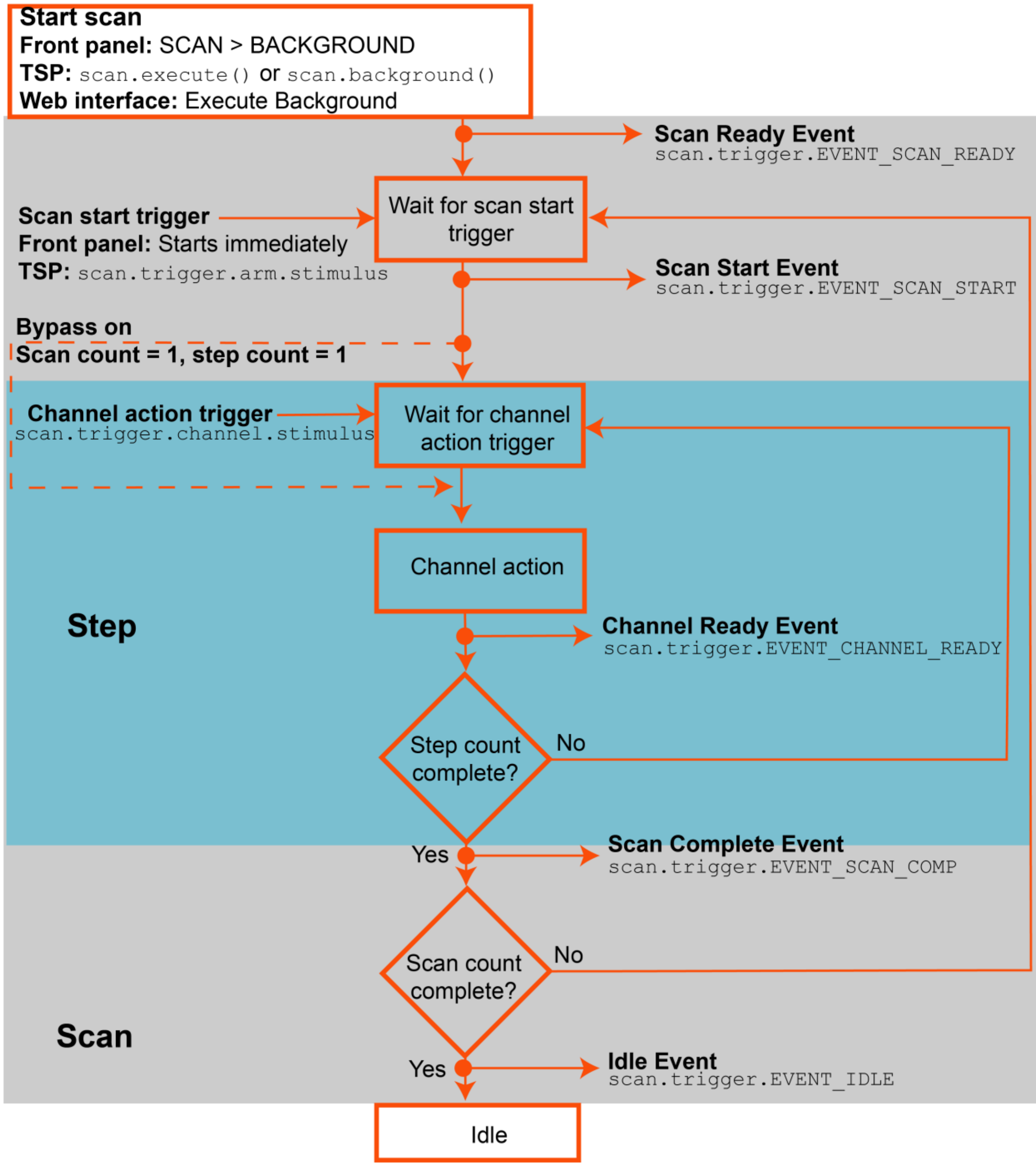
In 707B and 708B, only scanning operations use the trigger model. Individual open, close, and measure commands do not affect the trigger model.

The trigger model is used during a scan only. For front-panel operation, you use the **SCAN** and **STEP** keys to perform scan actions. For remote operation, you use the scan function and attribute commands, for example, `scan.execute()` and `scan.mode`.

NOTE

You cannot use an external trigger event (for example, digital I/O) for the channel stimulus setting of the trigger model when using the front-panel STEP key.

Figure 5: Models 707B and 708B triggering overview



Trigger model components

The individual components of the trigger model are explained in the following paragraphs.

Start scan

When a scan is initiated, the instrument leaves the idle state and prepares to start scanning. To prepare for scanning, it verifies that channel settings match the scan settings (such as opening all channels or opening only channels on specific slot). After preparation is complete, the instrument generates the Scan Ready Event and either starts immediately or waits for the arm stimulus event.

When the scan is complete, the instrument returns to the idle state.

Wait for scan start trigger

The scan can start immediately after receiving the Scan Ready Event, or the instrument can wait for a start trigger (also called the arm action). If it waits for a start trigger, the scan does not start until it receives the start trigger.

When the scan starts, the instrument generates the Scan Start Event.

Wait for channel action trigger

The channel action trigger is an event that tells the instrument to begin processing the next channel action.

You can bypass the channel action trigger if this is the first step of the first scan count. Bypass is available from the front-panel **CONFIG+SCAN** menu or by using the command `scan.bypass`.

Channel action

During the channel action, the instrument opens or closes channels as needed.

When the channel actions are complete, the instrument generates the Channel Ready Event.

Step count complete

The trigger model repeats these actions for every step in the scan list. The instrument loops back to wait for channel action trigger for each step until the steps are complete.

When all the steps are complete, the instrument generates the Scan Complete Event.

Scan count complete

The scan can be set to repeat. If the scan count is not complete, the instrument loops back to wait for the scan start trigger.

Idle

After the instrument has completed the scan list the requested number of times, the instrument generates the Idle Event. This is the completion of the scan.

Trigger model events and associated commands

The Models 707B and 708B trigger model has the following events and associated command attributes. You can use these events and other events in the system, to configure stimulus settings.

For example, the channel ready event (`scan.trigger.EVENT_CHANNEL_READY`) may be set to pulse digital I/O line 3 when it is generated. The command message for this is:

```
digio.trigger[3].stimulus = scan.trigger.EVENT_CHANNEL_READY
```

As another example, you can use the digital I/O line 5 trigger event to satisfy the scan trigger channel stimulus, which causes the channel action to occur when a trigger is detected on line 5. The command message for this is:

```
scan.trigger.channel.stimulus = digio.trigger[5].EVENT_ID
```

Event	Associated attribute
Scan Ready Event	`scan.trigger.EVENT_SCAN_READY`
Scan Start Event	`scan.trigger.EVENT_SCAN_START`
Channel Ready Event	`scan.trigger.EVENT_CHANNEL_READY`
Scan Complete Event	`scan.trigger.EVENT_SCAN_COMP`
Idle Event	`scan.trigger.EVENT_IDLE`

NOTE

Scanning operations run through the trigger model, but individual open and close commands have no interaction with the trigger model.

Scan and step counts

When running a scan, it may be necessary to determine the scan progress. You can use `scan.state()` to read the scan and step count.

The scan count is the number of the present iteration through the scan portion of the trigger model. This number does not increment until after the scan begins. Therefore, if an instrument is waiting for an input to trigger a scan start, the scan count represents the previous number of scan iterations. If no scan has begun, the scan count is zero.

The step count represents the number of times the scan has completed a pass through the channel action portion of the trigger model. This number does not increment until after the action completes. Therefore, if the instrument is waiting for an input to trigger a channel action, the step count represents the previous step. If no step has yet to complete, the step count is zero. If the step count has yet to complete the first step in a subsequent pass through a scan, the scan count represents the last step in the previous scan pass.

Scan list

Before you can run a scan, you must create a scan list. A scan list is a set of steps that runs in a specific order during a scan. Each step contains a channel, channels, or channel patterns that you want to measure in that step. Each step is acted on separately during the scan.

You can mix channel patterns and individual channels in a scan list. The steps are executed in the order in which they are added to the scan.

NOTE

Before setting up a scan list, make sure your channels and channel patterns are configured. Refer to Work with channels (on page 2-1) for detail.

If you change the channel configurations or channel patterns after the scan list has been set up, you may not see expected results. If the change prevents the scan from functioning properly (such as deleting something referenced by the scan), an error message is logged.

Basic scan procedure

NOTE

It is always better to configure all channel attributes before creating a scan.

You cannot use an external trigger event, like digital I/O, for the channel stimulus setting of the trigger model when using the front-panel **STEP** key.

To perform a scan:

1. Configure the channels for scanning as needed.
2. Build the scan list:
   - **Front panel**: Press the **INS** key. The steps are executed in the order in which they are added.
   - **Remote interface:** Send the `scan.create()`, `scan.add()`, or `scan.addimagestep()` command.
3. Configure the scan settings (for example, scan count, bypass, and mode).
4. To start the scan:
   - **Front panel**: Press the **STEP** key or the **SCAN** key and select the **BACKGROUND** menu item. The instrument leaves the idle state and performs the channel action associated with the first step in the scan list.
   - **Remote interface**: Send the command `scan.execute` or `scan.background`. The trigger model leaves the idle state and performs actions on the channels involved in scanning.

NOTE

While scanning is enabled, pressing most front-panel keys causes the instrument to display error code 5522, "Scan Running, Must Abort Scan."

5. The channels are scanned or stepped in the order they were added to the list.
   - **Front panel**: If you are stepping through the scan, press the **STEP** key to proceed to the next step in the list.
   - **Remote interface**: You cannot step a scan remotely over the bus.
6. To abort the scan:
   - **Front panel**: Press the **EXIT** key.
   - **Remote interface**: Send the `scan.abort()` command.

NOTE

If a scan is aborted, channels remain in the open or closed they were in when the abort occurred.

If configured to scan the channels in the scan list again, the Model 707B or 708B waits at the control source for another trigger event. After the scan is complete, the Model 707B or 708B outputs another trigger pulse, if configured to do so. After all requested scans are complete, the instrument returns to the idle state with the channels associated with last scan step closed.

Change attributes of an existing scan

When a scan already exists, if you change channel attributes, the scan is also changed. Once a scan list has been defined, the Model 707B or 708B tries to incorporate your changes into the scan. If the change impacts the ability of the scan to function properly (such as deleting something referenced by the scan), an error message is logged and the scan list may be cleared.

To see how the scan list may have changed, view the existing scan list:

1. Press the **SCAN** key when on the main display.
2. Select the **LIST** option and press the **ENTER** key.
3. Use the navigation wheel or **CURSOR** keys to scroll through the list.

For remote operation, use the scan.list() (on page 7-149) function.

You can clear an existing scan list before making any changes after making a scan list. From the front panel, press the **SCAN** key and select the **CLEAR** option. For bus operation, use the scan.create() (on page 7-148) function.

To configure a scan from the SCAN ATTR MENU, while in an active scan list:

1. Press the **CONFIG** key.
2. Press the **SCAN** key. Modify any of the following menu items as needed:
   - **ADD**: Displays **Use INS** key. The related command is scan.add (on page 7-144).
   - **BYPASS**: Enables (`ON`) or disables (`OFF`) bypass of the first step of the first scan pass. Related command: scan.bypass (on page 7-147).
   - **MODE**: Sets the scan mode value to one of the following: `OPEN_ALL` (default setting), `OPEN_SELECT`. Related command: scan.mode (on page 7-151).
3. Press the **EXIT** key to leave the menu.

Front-panel scan options

After channels are added to the scan list, press the **SCAN** key to display the SCAN ACTION MENU. If no scan list exists, pressing the **SCAN** key causes the instrument to briefly display `No Scan List. Use INSERT to add selection`.

The menu contains the following items:

- **BACKGROUND:** Runs scan list in the background.
- **CREATE**: Displays **Use INS key**.
- **LIST:** Displays the existing scan list steps. Turn the navigation wheel to scroll through the list.
- **CLEAR:** Clears the existing scan list.
- **RESET:** Resets the scan settings of the instrument, which include scan count, clearing the scan list, and scan stimulus settings like the scan trigger arm.

Press the **INS** key to add the selected channels or pattern to the existing scan list.

Press the **DEL** key to remove the selected channels or pattern from the existing scan list. Only the first occurrence of the selected item is removed.

When removing channels, channel patterns are not checked to determine if the channel being removed is associated with its image. To remove a channel pattern in a scan list, select the channel pattern to be removed, and then press the **DEL** key.

Press the **STEP** key to single step through a scan list.

Foreground and background scan execution

You can execute a scan in the foreground or background. Background execution allows you to query settings. If a scan is running in the foreground, it needs to finish or be aborted before you can query any settings.

When a scan is running in the background, you can send commands to be processed. The commands that you can use include most of the command messages that you use to query for settings, for example:

```
print(scan.state())
```

Most of the configuration commands log the following error message to the error queue:

```
5522, Scan Running, Must Abort Scan
```

If you use the front panel to run a script that starts a scan in the background, make sure the scan completes before the script ends. When scripts initiated from the front panel complete, the instrument transitions from remote to local, which will abort any background operations, including the scan.

Include multiple channels in a single scan step

Through the remote control interface, you can use `scan.addimagestep` to combine a list of channels into a scan step.

The following example creates five scan steps with the indicated channels.

```
scan.create()
scan.addimagestep("1A01, 1B01, 1C03")
scan.sddimagestep("1A03, 1B03, 1C03")
scan.addimagestep("1A05, 1B05, 1C03")
scan.sddimagestep("1A07, 1B07, 1C03")
scan.addimagestep("1A09, 1B09, 1C03")
```

Scan and trigger commands

The following list contains commands associated with triggers and bus operation scanning:

- trigger.blender[N].clear() (on page 7-207)
- trigger.blender[N].orenable (on page 7-208)
- trigger.blender[N].overrun (on page 7-209)
- digio.trigger[N].clear() (on page 7-58)
- digio.trigger[N].pulsewidth (on page 7-61)
- digio.trigger[N].stimulus (on page 7-63)
- digio.trigger[N].wait() (on page 7-64)
- lan.trigger[N].assert() (on page 7-115)
- lan.trigger[N].clear() (on page 7-116)
- lan.trigger[N].overrun (on page 7-121)
- lan.trigger[N].stimulus (on page 7-123)
- lan.trigger[N].wait() (on page 7-125)
- scan.add() (on page 7-144)
- scan.background() (on page 7-146)
- scan.bypass (on page 7-147)
- scan.create() (on page 7-148)
- scan.execute() (on page 7-149)
- scan.list() (on page 7-149)

- scan.mode (on page 7-151)
- scan.reset() (on page 7-152)
- scan.scancount (on page 7-153)
- scan.state() (on page 7-154)
- scan.stepcount (on page 7-155)
- scan.trigger.arm.clear() (on page 7-155)
- scan.trigger.arm.set() (on page 7-156)
- scan.trigger.arm.stimulus (on page 7-156)
- scan.trigger.channel.clear() (on page 7-157)
- scan.trigger.channel.set() (on page 7-158)
- scan.trigger.channel.stimulus (on page 7-158)
- scan.trigger.clear() (on page 7-160)

Remote scan examples

Assume you have a 7072 card installed in slot 1 of your instrument and you want to scan column 5 on rows 1 to 3. To create this scan, send the following command:

```
scan.create("1A05, 1B05, 1C05")
```

To see the scan list generated from this command, send:

```
print(scan.list())
```

The following is output:

```
 Init) OPEN...
  1) STEP: 1A05
     CLOSE: 1A05
  2) STEP: 1B05
     OPEN: 1A05
     CLOSE: 1B05
  3) STEP: 1C05
     OPEN: 1B05
     CLOSE: 1C05
```

This indicates that the scan list includes three steps:

1. The running scan closes `1A05`.
2. The scan opens `1A05` and closes `1B05`.
3. The scan opens `1B05` and closes `1C05`.

The following table illustrates how scan count works with this scan list to determine the total number of channel closures during a scan.

Number of steps	Scan count value	Total number of step channel closures
3	1 (default)	3 (each step channel once – 1A05, 1B05, 1C05)
3	2	6 (each step channel twice – 1A05, 1B05, 1C05, then 1A05, 1B05, 1C05)
3	3	9 (each step channel three times – 1A05, 1B05, 1C05 then 1A05, 1B05, 1C05 then 1A05, 1B05, 1C05)
`scan.stepcount`	`scan.scancount`	`scan.stepcount X scan.scancount`

Therefore, scan count represents how many times to loop on the total number of steps in a defined scan list.

If you want to pace the steps of closing and opening channels by detecting a trigger on digital I/O trigger line 3, the following flowchart indicates the sequence of events to achieve this. This example uses the same scan list as above and uses a scan count of 3.

As the following flowchart shows, to transition between start of scan to step 1, step 1 and step 2, step 2 and step 3, a trigger needs to be detected on digital I/O trigger line 3. At the end of step 3, if the scan count has reached three, the scan completes. If the scan count has not reached 3 then, loop back around to repeat steps 1 to 3 again.

The following represents a script you can enter using the TSB Embedded page on the web to experiment with the scan points discussed above.

```
reset()
scan.add("1A05, 1B05, 1C05")
digio.trigger[3].mode = digio.TRIG_FALLING
scan.trigger.channel.stimulus =
   digio.trigger[3].EVENT_ID
scan.scancount = 3
print(scan.list())
delay(5)
scan.background()
ScanState, ScanCount, StepCount = scan.state()
print(ScanState, "", ScanCount , "", StepCount)
delay(3)
print("state", " scan count", "step count")
for x = 1, (scan.scancount * scan.stepcount) do
   digio.trigger[3].assert()
   ScanState, ScanCount, StepCount =
    scan.state()
   print(ScanState, "", ScanCount, "",
    StepCount)
   delay(3)
end
```

Reset the instrument.
Create a three channel scan list.
Detect falling edges on digital I/O trigger line 3.
Configure digital I/O trigger line 3 to pace the scan sequence.
Set the scan count to 3.
Show the configured scan list.
Delay 5 seconds to allow viewing of scan list.
Start the scan and let it run in the background.
Get the present scan state information
Show the state of scanning.
Delay 3 seconds to allow viewing of state.
Header for output data from the for loop.
Loop for scan count times step count to step the scan sequence.
Simulate the trigger occurring on digital I/O trigger line 3.
Get the present scan state information.
Show the state of scanning.
Delay 3 seconds to allow viewing of state.

On the TSB Embedded page, select **Save Script** to save the script. For example, name the script `ScanExample`. In the User Scripts list, select `ScanExample`. Select **Run**. The output of the script is shown in the following table.

Init) OPEN...
  1) STEP: 1A05
    CLOSE: 1A05
  2) STEP: 1B05
      OPEN: 1A05
     CLOSE: 1B05
  3) STEP: 1C05
      OPEN: 1B05
     CLOSE: 1C05	

2.000000000e+000 1.000000000e+000 0.000000000e+000	Shows scan is running (state = 2) and no steps have completed for scan count of 1
state scan count step count	
2.000000000e+000 1.000000000e+000 1.000000000e+000	Shows scan is running (state = 2) and completed step 1 for scan count of 1
2.000000000e+000 1.000000000e+000 2.000000000e+000	Shows scan is running (state = 2) and completed step 2 for scan count of 1
2.000000000e+000 1.000000000e+000 3.000000000e+000	Shows scan is running (state = 2) and completed step 3 for scan count of 1
2.000000000e+000 2.000000000e+000 1.000000000e+000	Shows scan is running (state = 2) and completed step 1 for scan count of 2
2.000000000e+000 2.000000000e+000 2.000000000e+000	Shows scan is running (state = 2) and completed step 2 for scan count of 2
2.000000000e+000 2.000000000e+000 3.000000000e+000	Shows scan is running (state = 2) and completed step 3 for scan count of 2
2.000000000e+000 3.000000000e+000 1.000000000e+000	Shows scan is running (state = 2) and completed step 1 for scan count of 3
2.000000000e+000 3.000000000e+000 2.000000000e+000	Shows scan is running (state = 2) and completed step 2 for scan count of 3
6.000000000e+000 3.000000000e+000 3.000000000e+000	Shows scan has completed (state = 6) and completed step 3 for scan count of 3

Hardware trigger modes

Use the hardware trigger modes to integrate Keithley and non-Keithley instruments into an efficient test system. The hardware synchronization lines are classic trigger lines. The Model 707B or 708B contains 14 digital I/O lines and 3 TSP-Link synchronization lines that you can use for input or output triggering. The following table provides a summary for each hardware trigger mode.

Trigger mode	**Output**		**Input**	**Notes**
	Unasserted	**Asserted**	**Detects**	
Bypass	N/A	N/A	N/A	Use the `writebit` and `writeport` commands for direct line control
Either edge	High	Low	Either	Short input pulses can cause a trigger overrun
Falling edge	High	Low	Falling	
Rising edge	N/A	N/A	N/A	▪ The programmed state of the line determines if the behavior is similar to RisingA or RisingM ▪ High similar to RisingA ▪ Low similar to RisingM
Rising A	High	Low	Rising	
RisingM	Low	High	None	
Synchronous	High latching	Low	Falling	▪ Behaves similar to SynchronousA ▪ Trigger overrun detection is disabled ▪ To mirror the SynchronousA trigger mode, set the pulse duration to 1 µs or any small nonzero value
SynchronousA	High latching	High	Falling	Ignores the pulse duration
SynchronousM	High	Low	Rising	

Each trigger mode controls the input trigger detection and output trigger generation. The input detector monitors for and detects all edges, even if the node that generates the output trigger causes the edge.

A trigger overrun is generated if an input trigger is received before the previous input trigger processes. To determine if a trigger overrun has occurred, reference the trigger overrun attributes.

For additional information on the hardware trigger modes, see TSP commands (on page 7-7).

NOTE

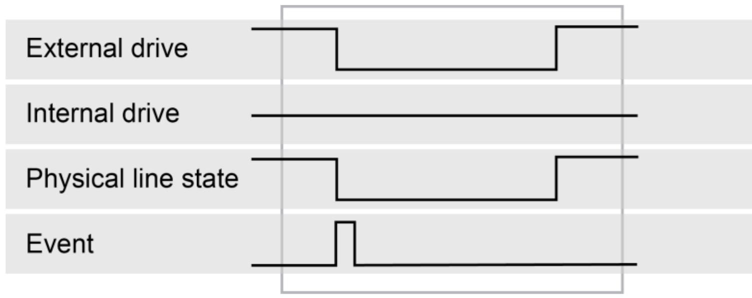
To have direct control of the line state, use the bypass trigger mode.

Falling edge trigger mode

The falling edge trigger mode generates low pulses and detects all falling edges.

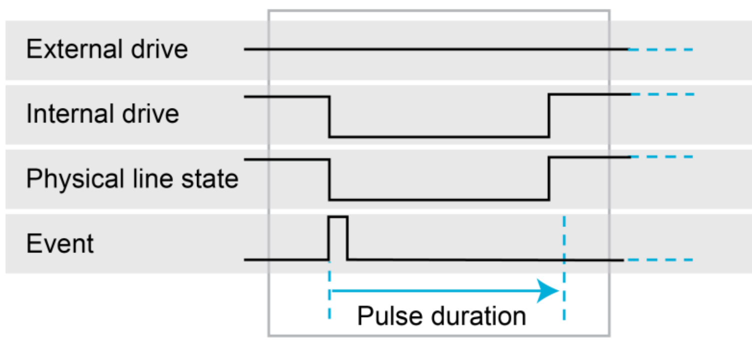
The following figure shows the characteristics of the falling edge input trigger. It detects all falling edges as input triggers.

Figure 6: Falling edge input trigger



The following figure shows the characteristics of the falling edge output trigger. When the trigger is asserted, it generates a low pulse for the programmed pulse duration.

Figure 7: Falling edge output trigger



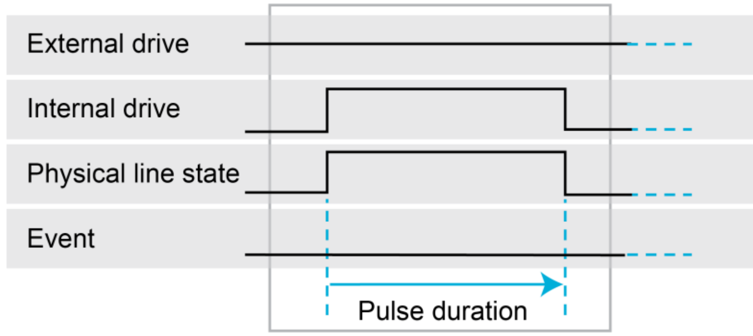
Rising edge master trigger mode

Use the rising edge master trigger mode (RisingM) to synchronize with non-Keithley instruments that require a high pulse. Input trigger detection is not available in this trigger mode. You can use the RisingM trigger mode to generate rising edge pulses.

NOTE

The RisingM trigger mode does not function properly if the line is driven low by an external drive.

Figure 8: Rising edge master trigger mode output trigger

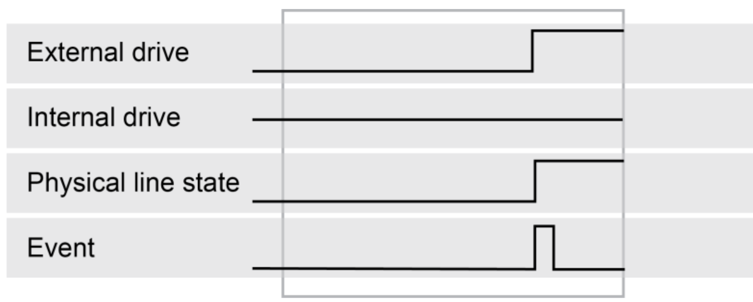


Rising edge acceptor trigger mode

The rising edge acceptor trigger mode (RisingA) generates a low pulse and detects rising edge pulses.

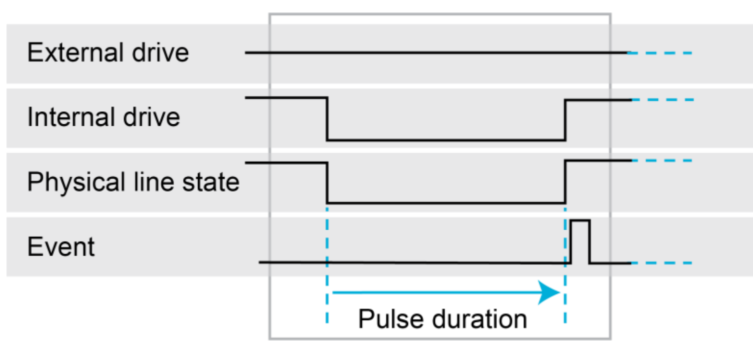
The following figure shows the RisingA input trigger. All rising edges generate an input event.

Figure 9: Rising edge acceptor trigger mode input trigger



The following figure shows the Rising A output trigger. When the trigger is asserted, it generates a low pulse that is similar to the falling edge trigger mode.

Figure 10: Rising edge acceptor trigger mode output trigger

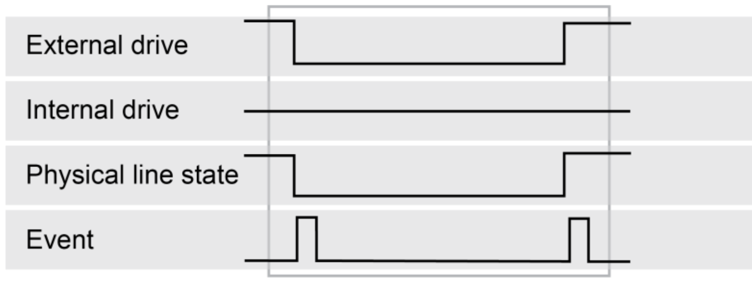


Either edge trigger mode

The either edge trigger mode generates a low pulse and detects both rising and falling edges.

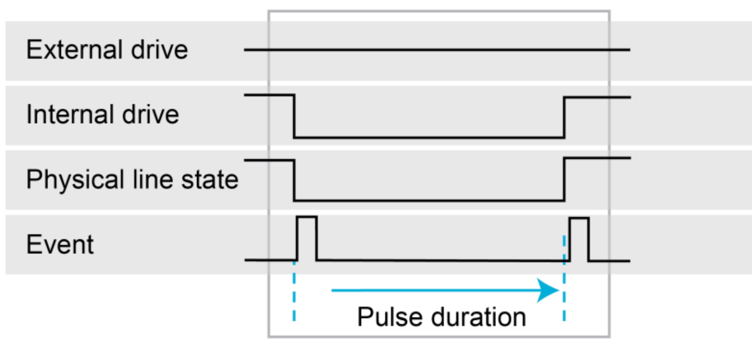
The following figure shows the either edge trigger mode input trigger. All rising or falling edges generate an input trigger event.

Figure 11: Either edge trigger mode input trigger



The following figure shows the output trigger. When the trigger is asserted, it generates a low pulse that is similar to the falling edge trigger mode.

Figure 12: Either edge trigger mode output trigger



About the synchronous trigger modes

Use the synchronous trigger modes to implement bidirectional triggering, to wait for one node, or to wait for a collection of nodes to complete all triggered actions.

All non-Keithley instrumentation must have a trigger mode that functions similar to the SynchronousA or SynchronousM trigger modes.

To use synchronous triggering, configure the trigger master to SynchronousM trigger mode or the non-Keithley equivalent. Configure all other nodes in the test system to SynchronousA trigger mode or a non-Keithley equivalent.

Synchronous master trigger mode

Use the synchronous master trigger mode to generate falling edge output triggers, to detect the rising edge input triggers, and to initiate an action on one or more external nodes with the same trigger line.

In this mode, the output trigger consists of a low pulse. All non-Keithley instruments attached to the synchronization line in a trigger mode equivalent to SynchronousA must latch the line low during the pulse duration.

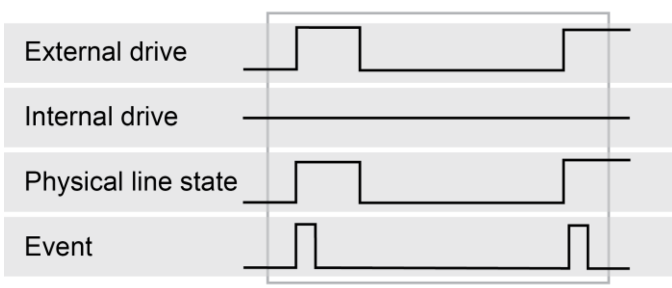
To use the SynchronousM trigger mode, configure the triggering master as SynchronousM and then configure all other nodes in the test system as Synchronous, SynchronousA, or to the non-Keithley instruments equivalent.

NOTE

Use the SynchronousM trigger mode to receive notification when the triggered action on all nodes is complete.

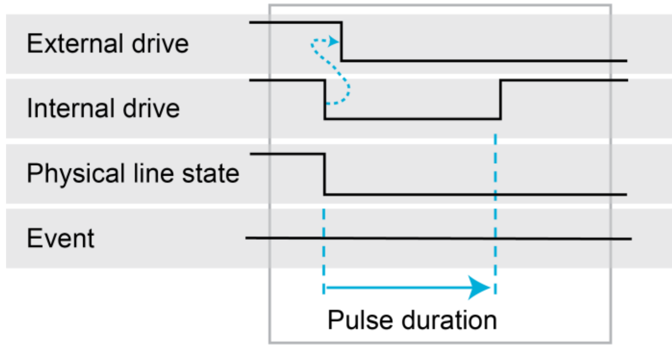
The following figure shows the synchronous master trigger mode input trigger. All rising edges are input triggers. When all external drives release the physical line, the rising edge is detected as an input trigger. A rising edge cannot be detected until all external drives release the line and the line floats high.

Figure 13: Synchronous master trigger mode input trigger



The following figure shows the output trigger. When the trigger is asserted, it generates a low pulse that is similar to the falling edge trigger mode.

Figure 14: Synchronous master trigger mode output trigger

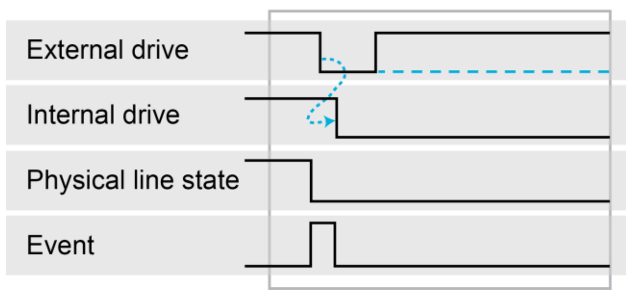


Synchronous acceptor trigger mode

Use the synchronous acceptor trigger mode (SynchronousA) on a trigger subordinate that operates with a trigger master configured for the SynchronousM trigger mode. The roles of the internal and external drives are reversed in the SynchronousA trigger mode.

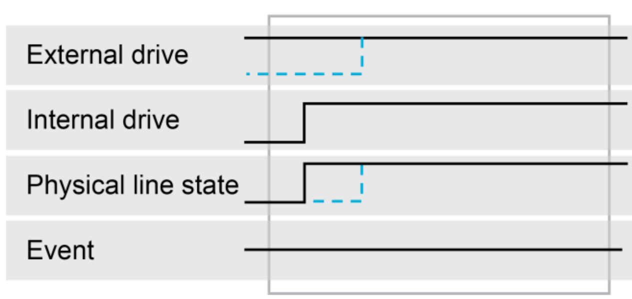
The following figure shows the SynchronousA input trigger. The falling edge is detected as the external drive pulses the line low, and the internal drive latches the line low.

Figure 15: Synchronous acceptor trigger mode input trigger



The following figure shows the SynchronousA output trigger. The physical line state does not change until all drives (internal and external) release the line.

Figure 16: Synchronous acceptor trigger mode output trigger

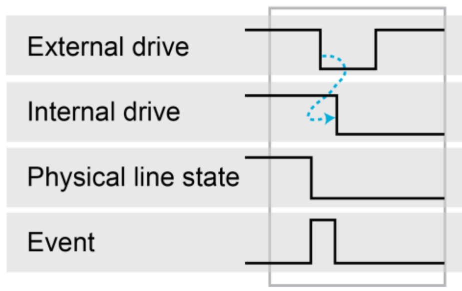


Synchronous trigger mode

The synchronous trigger mode is a combination of the SynchronousA and SynchronousM trigger modes.

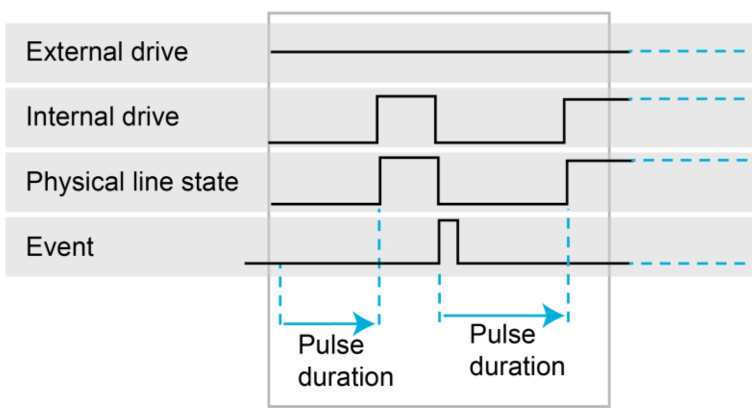
The following figure shows the synchronous trigger mode input trigger. The falling edge generates an input event and latches the internal drive low.

Figure 17: Synchronous trigger mode input trigger



The following figure shows the synchronous trigger mode output triggers. When the trigger is asserted, it generates a low pulse for the programmed pulse duration. If the line is latched low, a falling edge does not occur. When the trigger is asserted and the line is latched low, the pulse duration is enforced, and then the internal line drive is released. A normal falling edge pulse is generated when the internal drive is not latched low and the trigger is asserted.

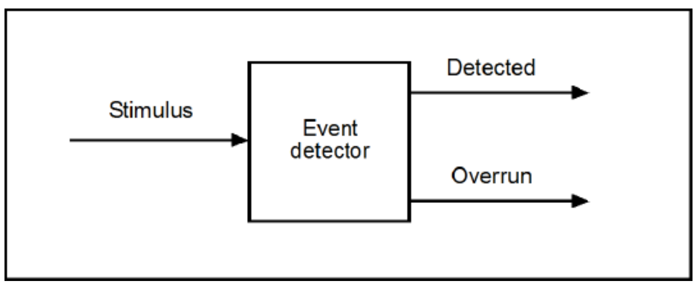
Figure 18: Synchronous trigger mode output trigger



Events

Event detectors monitor an event. They have one input signal (the stimulus), which is the event that they monitor (in some cases, the stimulus is an action in the system, like a timer expiring or a key press). They have two optional output signals, as shown in the following figure. “Detected” reflects the detection state of the event detector. If an event was detected, the detected signal is asserted. Event detectors are usually coupled to something that consumes the events. When an event is consumed, the detected state of the event detector is reset. If an event be detected while the event detector is in the detected state, the overrun signal is asserted. You can only clear the overrun signal by sending a TSP command.

Figure 19: Event detector



Event blenders

Advanced event handling requires a way to wait for one of several events (or all of several events). An event blender provides for this combining or blending of events. An event blender can combine up to four events in either an "or" mode or an "and" mode. In "or" mode, any one of the input events causes an output event to be generated. In "and" mode, all the input events must occur before an output event is generated.

When operating in "and" mode, if an event is detected more than once before all events necessary for the generation of an output event occur, an action overrun is generated. When operating in "or" mode, an action overrun is generated when two or more source events are detected simultaneously.

Each event blender has an associated event detector that can be accessed through script control. Event blenders can only be accessed over a remote interface (no front-panel control is available). The following remote commands provide additional information on available blenders:

trigger.blender[N].clear() (on page 7-207)
trigger.blender[N].orenable (on page 7-208)
trigger.blender[N].overrun (on page 7-209)
trigger.blender[N].stimulus[M] (on page 7-210)
trigger.blender[N].wait() (on page 7-212)

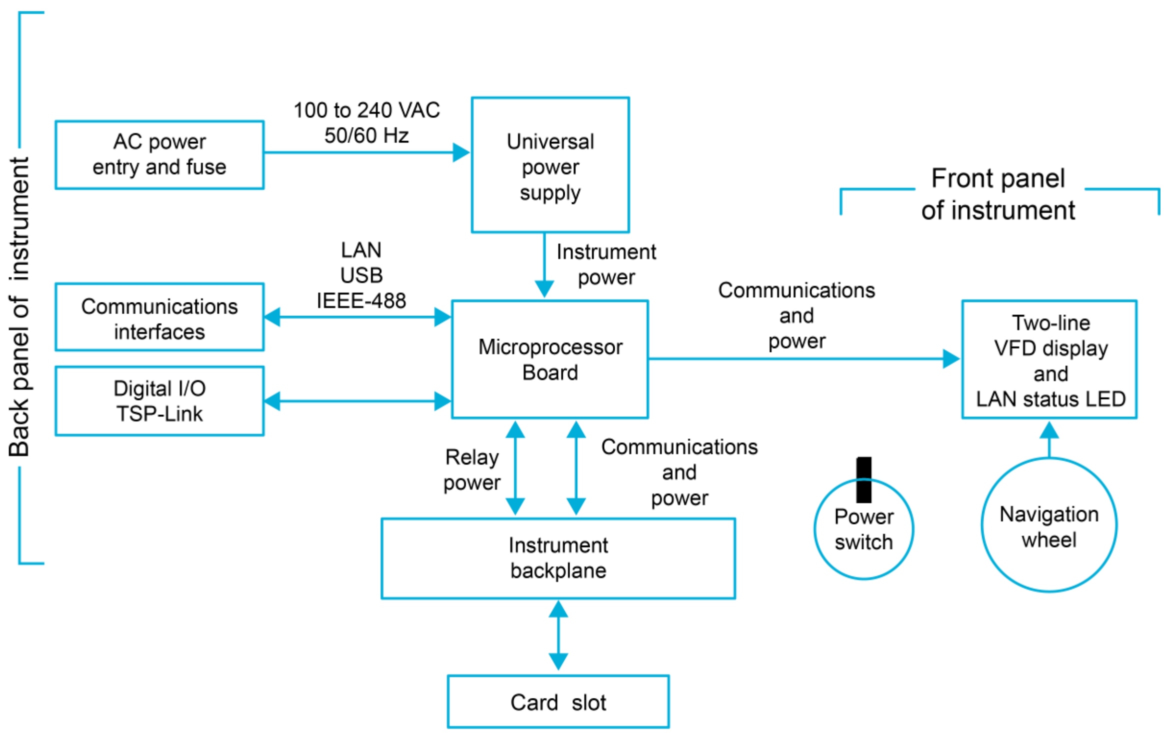
Theory of operation

In this section:

Overview

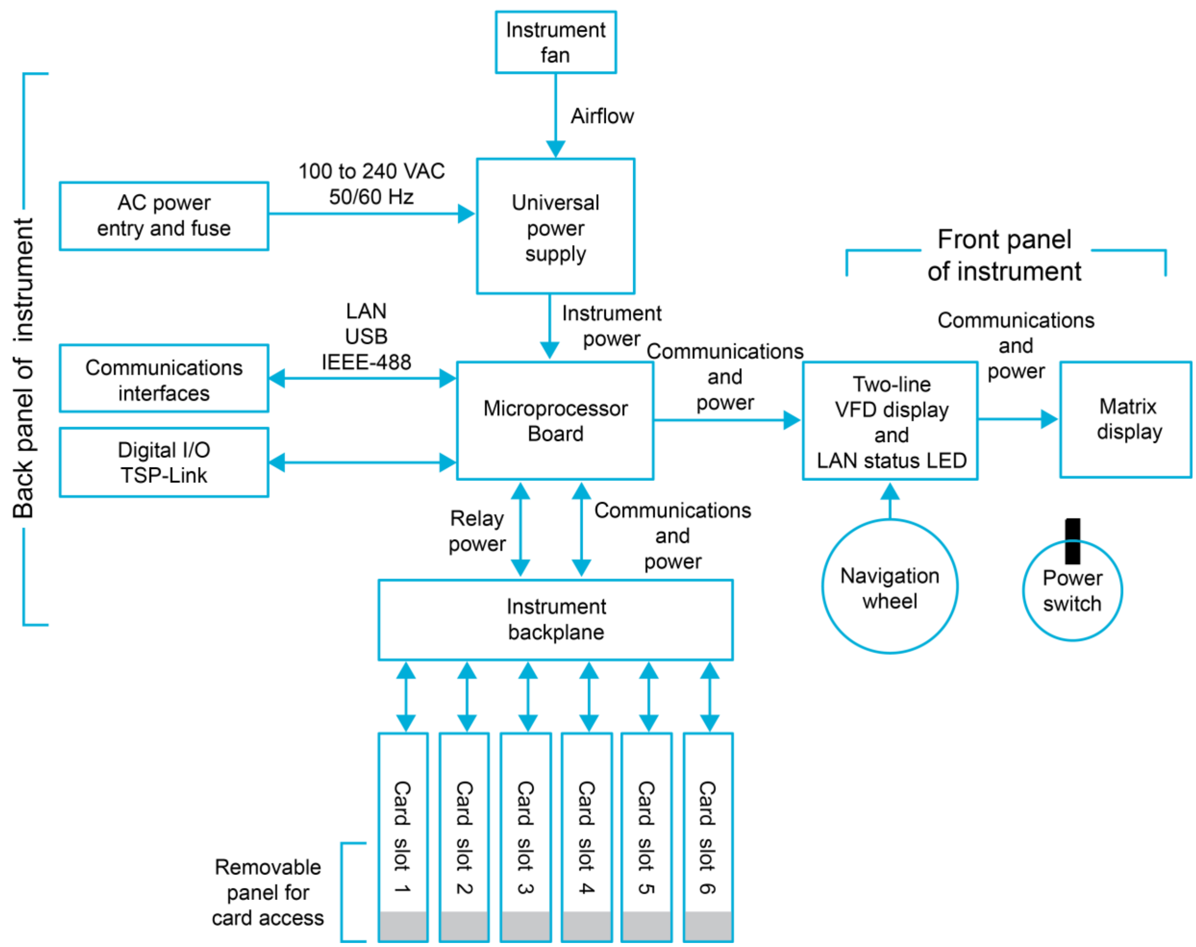
The 707B and 708B are composed of several components assembled into an aluminum frame. The Model 707B supports the operation of up to six relay matrix cards. The Model 708B supports the operation of a single relay matrix card. Each of the components is briefly described in the following.

Figure 20: Model 708B mainframe block diagram



Mainframe

Figure 21: Model 707B mainframe block diagram



Important test system safety information

This product is sold as a stand-alone instrument that may become part of a system that could contain hazardous voltages and energy sources. It is the responsibility of the test system designer, integrator, installer, maintenance personnel, and service personnel to make sure the system is safe during use and is operating properly.

You must also realize that in many test systems a single fault, such as a software error, may output hazardous signal levels even when the system indicates that there is no hazard present.

It is important that you consider the following factors in your system design and use:

- The international safety standard IEC 61010-1 defines voltages as hazardous if they exceed 30 V_{RMS} and 42.4 V_{PEAK} or 60 VDC for equipment rated for dry locations. Keithley products are only rated for dry locations.
- Read and comply with the specifications of all instruments in the system. The overall allowed signal levels may be constrained by the lowest rated instrument in the system. For example, if you are using a 500 volt power supply with a 300 VDC rated switch, the maximum allowed voltage in the system is 300 VDC.
- Make sure any test fixture connected to the system protects the operator from contact with hazardous voltages, hot surfaces, or sharp objects. This may be accomplished by shields, barriers, insulation, safety interlocks, or the like.
- Cover the device under test (DUT) to protect an operator from flying debris in the event of a system or DUT failure.
- Double insulate all electrical connections that an operator could touch. Double insulation ensures the operator is still protected even if one insulation layer fails. Refer to IEC 61010-1 for specific requirements.
- Make sure all connections are behind a locked cabinet door or other barrier. This protects the system operator from accidentally removing a connection by hand and exposing hazardous voltages.
- Use high-reliability fail-safe interlock switches to disconnect power sources when a test fixture cover is opened.
- Where possible, use automatic handlers so that operators are not required to access the DUT or other potentially hazardous areas.
- Provide training to all users of the system so that they understand all potential hazards and know how to protect themselves from injury.
- In many systems, during power up, the outputs may be in an unknown state until they are properly initialized. Make sure the design can tolerate this situation without causing operator injury or hardware damage.

Always read and follow all safety warnings provided with the specific instruments to keep system users safe.

For Model 707B or 708B, also see Safety Precautions.

Instrument fan (Model 707B only)

The Model 707B includes an internal fan that operates continuously when power is applied to the mainframe. The fan keeps the internal power supply and other electronics cool during operation.

AC power entry

The Model 707B or 708B is powered from standard AC mains supply through a power entry module that is located on the rear panel of the instrument.

Universal power supply

All the power for the internal electronics and relay cards is provided by a switched-mode power supply that is located inside the mainframe.

Total power consumption requirements are listed on the rear panel of the instrument.

Microprocessor board

At the heart of the Model 707B or 708B instrument is a microprocessor board that processes all communications. It outputs information and status messages and executes operational commands supplied by the operator.

For information on the commands that can be sent to the instrument, see Introduction to TSP operation (on page 5-1).

Remote communications interfaces

You can operate the Model 707B or 708B over one of several standard remote interfaces. The rear panel includes connectors for these communications interfaces:

- Universal Serial Bus (USB)
- Local area network (LAN)
- General purpose interface bus (GPIB or IEEE-488)

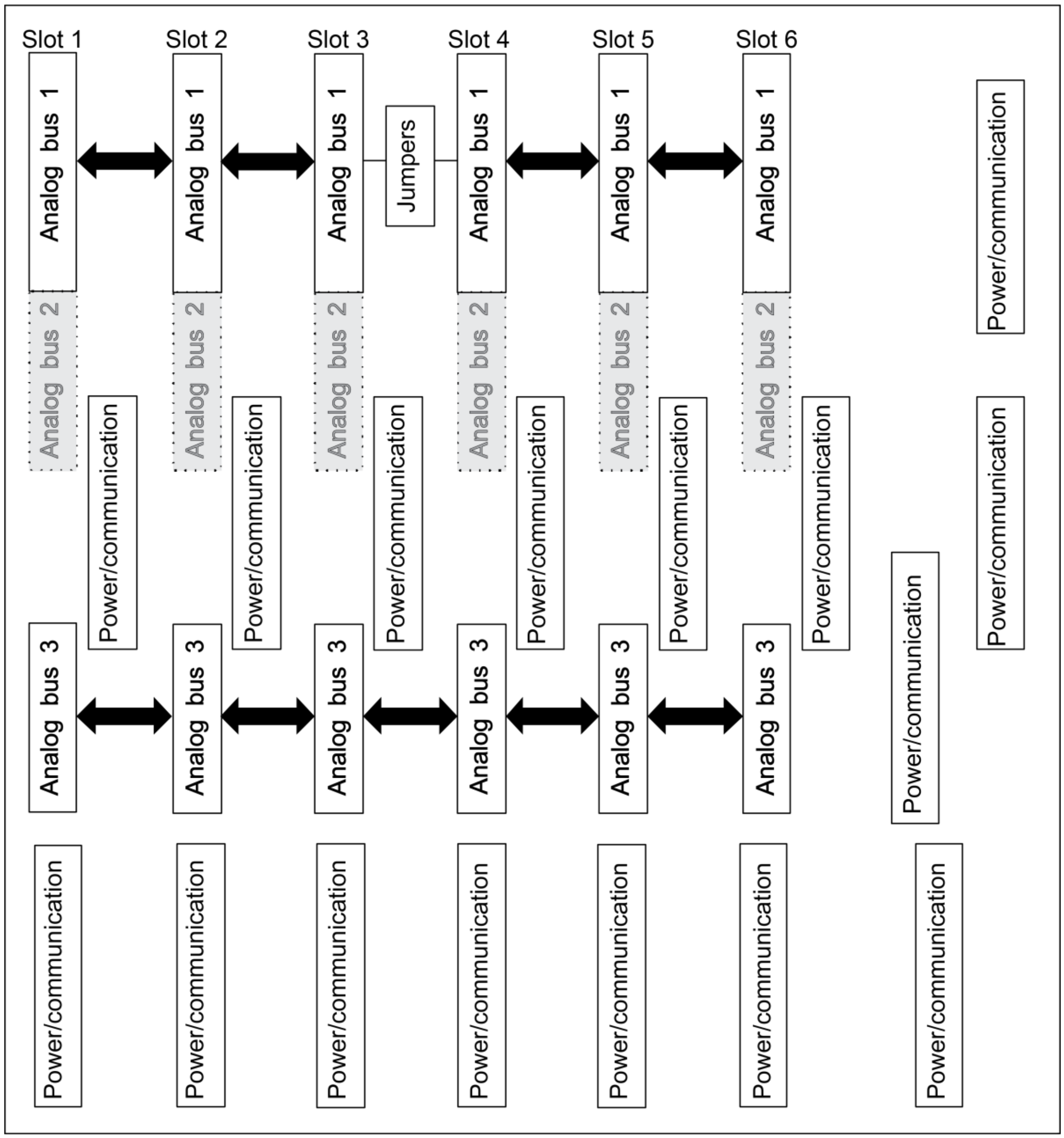
Trigger and control interfaces

You can use rear panel connections to control external digital circuits and instruments that are connected with TSP. See Digital I/O port (on page 3-9) and Connect the TSP-Link cable (on page 6-48).

Backplane

The Model 707B or 708B backplane is the interface between the installed relay cards and the instrument. The relay card interfaces with the backplane through a communications and power connector and possibly a separate connector for signal routing.

Figure 22: Model 707B backplane



Front panel

The front panel of the Keithley Model 707B or 708B contains:

- The display
- The crosspoint display (Model 707B only)
- The keys and navigation wheel ◷
- The LAN status indicator
- The POWER button

You can use the displays, keys, and the navigation wheel to change the selected channel or channel pattern. You can also use them to access, view, and edit the menu items. The crosspoint display on the Model 707B shows you which channels are open and closed.

The small display is a two-line vacuum fluorescent display (VFD).

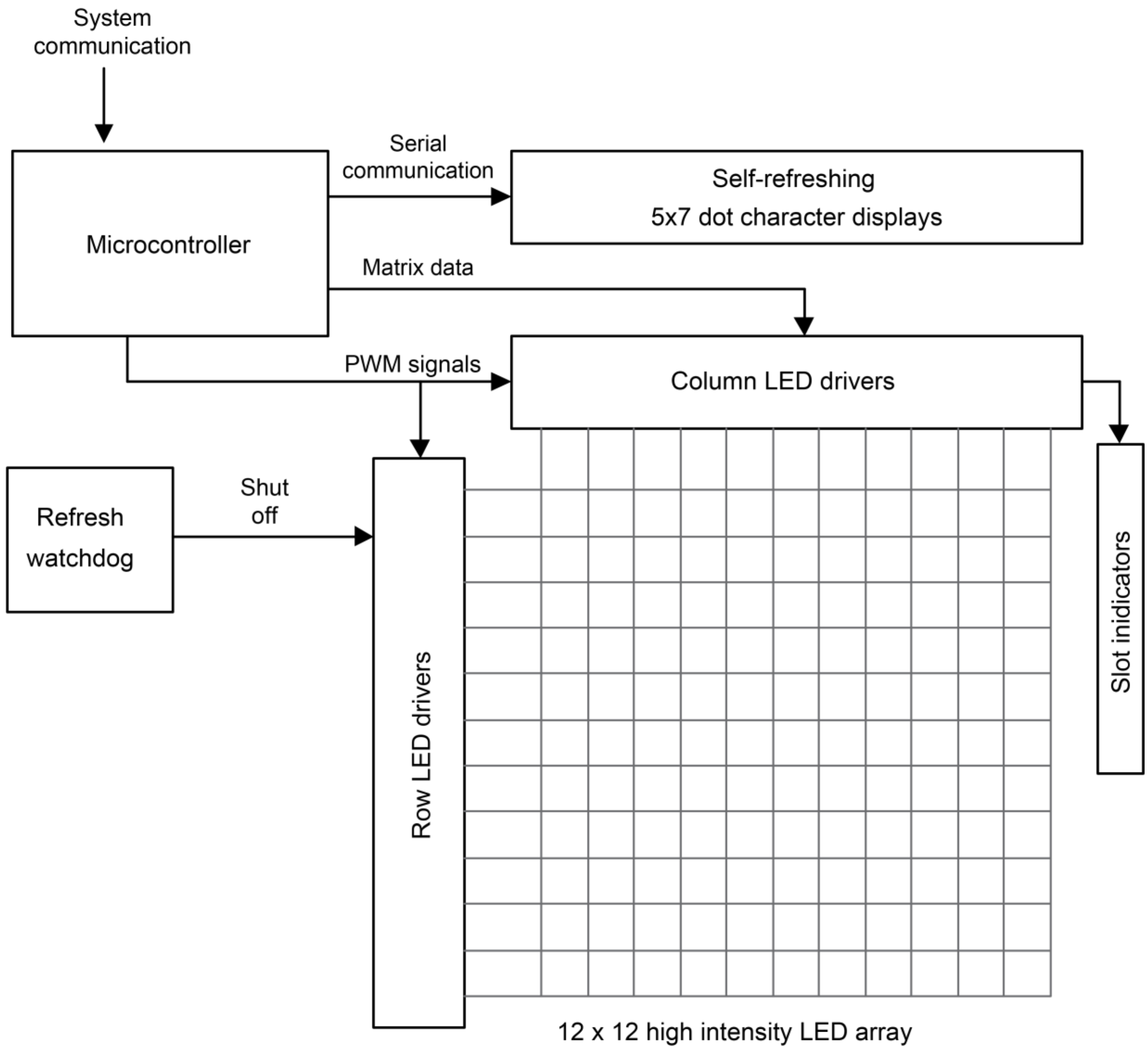
Upper crosspoint display (Model 707B only)

The larger upper display on the Model 707B mainframe provides a visual means of determining the relay status of each card slot in the instrument at any given time. It also shows the label names for each row and column of the given slot.

The crosspoint display may be operated from the navigation wheel to scroll through the slots displaying the status. You can use the navigation wheel to select channels for relay open or closures or other actions.

For details on indicators and additional options, see “Crosspoint display (Model 707B only)” in the *Models 707B and 708B User's Manual*.

Figure 23: Model 707B crosspoint display block diagram



Description of the crosspoint display components

The front panel of the Model 707B has two main display components, the vacuum fluorescent display (VFD) and a 12×12 LED matrix display. The matrix display, also called the crosspoint display, shows the relay status for each of the slots.

The previous block diagram shows only the matrix display. Its components are described in the following.

Microcontroller

The microcontroller receives commands from the main controller and updates the row and column labels. It also generates the pulse-width modulated (PWM) signals for operating the 12×12 LED matrix.

Row and column labels

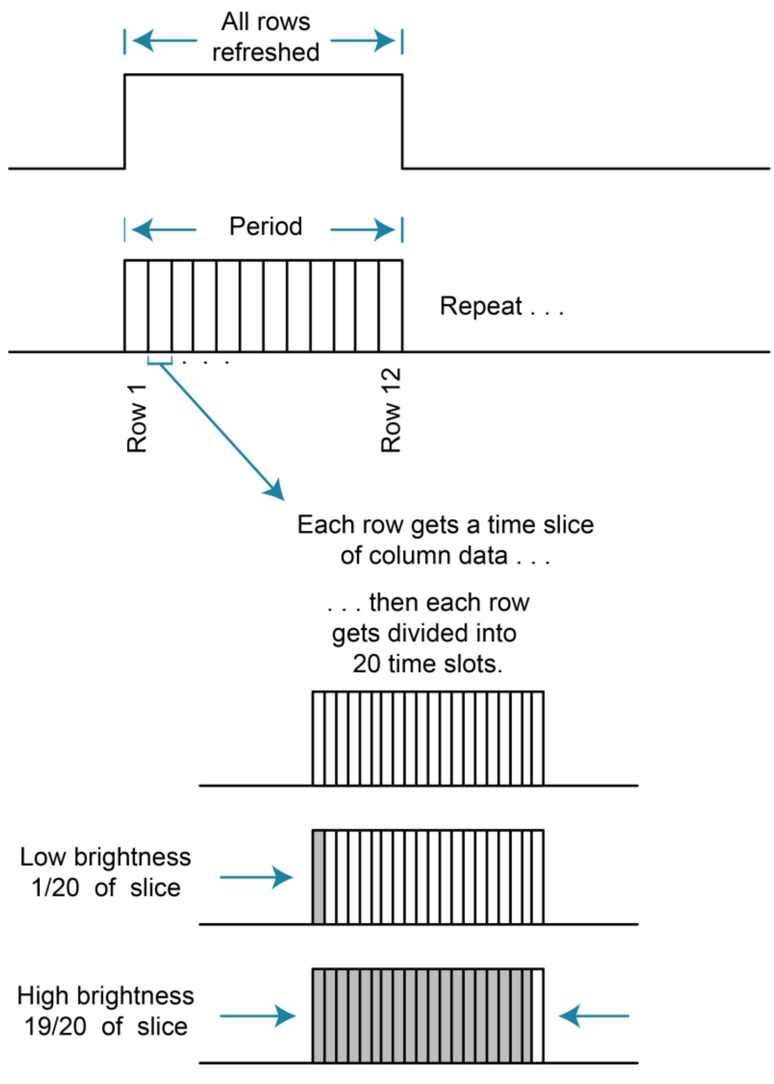
The row and column labels are updated by writing to the devices serially. They handle their own brightness and refresh controls, alleviating the main microcontroller from performing these tasks. As a result, these are only updated when the data changes and a command from the instrument is issued to update the label.

12×12 LED matrix

The 12×12 matrix array of LEDs is driven by PWM signals that cause the intensity of the LED to be bright or dim based on the appropriate setting from the instrument.

Each row and column is updated sequentially. The entire display has a refresh rate of approximately 60 Hz. The time domain details of display refreshing are shown in the following figure.

Figure 24: Model 707B display refresh



Slot indicators

The slot LEDs are treated the same as the matrix LEDs, as additional 13th and 14th[h] columns.

Display watchdog

Once the display tasks are started, if the microcontroller stops refreshing, the watchdog circuit shuts off the column drivers to preserve the life of the LEDs.

The display watchdog is part of the standard display test. Refer to Test the display, keys, and channel matrix (on page 11-9) for information.

If the watchdog shuts down the display, the first several row labels show “SLOT GRID LEDS OFF.”

Description of basic display operation

This display board is a subordinate device to the microprocessor board.

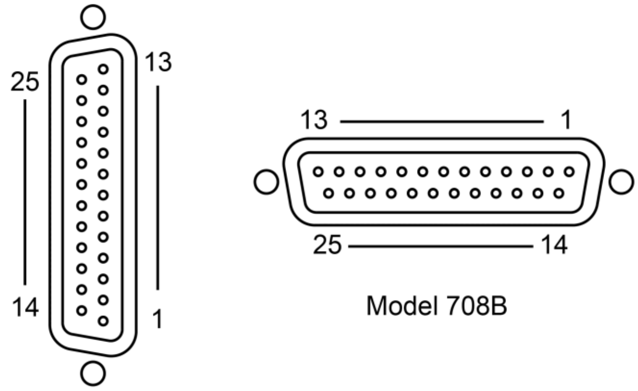
The microprocessor board communicates commands to change the state of the matrix or update the row and column labels.

The display board processes these commands independently. It also refreshes the display and performs other tasks. When a display command is issued, the microcontroller processes it to update the display. The task of refreshing rows and PWM generation is done in an interrupt-driven routine to ensure predictable timing.

Digital I/O port

The 707B and 708B have a digital input/output port that can be used to control external digital circuitry. For example, a handler that is used to perform binning operations can be used with a digital I/O port. The digital I/O port is a standard female DB-25 connector, as shown in the following figure.

Figure 25: Models 707B and 708B digital I/O ports



Pin	Description
1 ... 14	Digital I/O #1 ... Digital I/O #14
15 to 21	Ground
22	+5 VDC, limited to 600 mA, solid-state fuse protected
23	Not connected; pin reserved for future use
24	Not connected; pin reserved for future use
25	+5 VDC, limited to 600 mA, solid-state fuse protected

NOTE

If you are moving from a Model 707A or 708A, the digital I/O lines were fixed as follows:

- **Digital input:** Digital I/O lines 1 to 6
- **Digital output:** Digital I/O lines 7 to 12
- **External trigger:** Digital I/O line 13
- **Matrix ready:** Digital I/O line 14

Connect cables

Use a cable equipped with a standard male DB-25 connector (L-com part number CSMN25MF-5).

Digital I/O lines

The port provides 14 digital I/O lines (pins 1 through 14). Each output is set high (+5 V) or low (0 V) and can read high or low logic levels.

+5 V output

The digital I/O port provides three +5 VDC output lines that you can use to drive external logic circuitry. The maximum combined current output for all lines is 250 mA. These lines are protected by a self-resetting fuse with a one-hour recovery time.

Control digital I/O lines

Although the digital I/O lines are primarily intended for use with a device handler for limit testing, you can also use them for other purposes, such as controlling external logic circuits. You can control lines either from the front panel or over a remote interface.

You must write a `1` to all digital I/O lines that are to be used as inputs. The trigger mode for the line must be set to `digio.TRIG_BYPASS` to use the line for digital I/O. Refer to Trigger model (on page 2-21) for more information.

The digital I/O lines are not affected by any reset. However, they are affected by a power cycle.

NOTE

For the decimal values of the digital I/O lines, refer to the table in Digital I/O bit weighting (on page 3-12).

To set digital I/O values from the front panel:

1. Press the **MENU** key, select **DIGIO**, and then press the **ENTER** key or press the navigation wheel ◔.
2. Select **DIG-IO-OUTPUT**, and then press the navigation wheel ◔.
3. Set the decimal value as required to set digital I/O lines in the range of 0 to 16,383. For example, to set digital I/O lines 3 and 8, set the value to `132`.
4. Press the navigation wheel ◔.
5. Press the **EXIT (LOCAL)** key as needed to return to the main menu.

To write-protect specific digital I/O lines to prevent their values from being changed:

1. Press the **MENU** key, then select **DIGIO**, and then press the navigation wheel ◔.
2. Select **WRITE-PROTECT**, and then press the navigation wheel ◔.
3. Set the decimal value as required to write-protect digital I/O lines within the range of 0 to 16,383. For example, to write-protect digital I/O lines 4 and 10, set the value to `520`.
4. Press the navigation wheel ◔.
5. Press the **EXIT (LOCAL)** key as needed to return to the main menu.

To remove write protection, reset the decimal value to include only the lines that you want to write protect. To remove write protection from all lines, set the value to `0`.

Digital I/O bit weighting

Bit weighting for the digital I/O lines is shown in the following table.

Digital bit weight

Line #	Bit	Decimal weighting	Hexadecimal weighting
1	B1	1	0x0001
2	B2	2	0x0002
3	B3	4	0x0004
4	B4	8	0x0008
5	B5	16	0x0010
6	B6	32	0x0020
7	B7	64	0x0040
8	B8	128	0x0080
9	B9	256	0x0100
10	B10	512	0x0200
11	B11	1,024	0x0400
12	B12	2,048	0x0800
13	B13	4,096	0x1000
14	B14	8,192	0x2000

Section 4

Remote operations

In this section:

Remote communications interfaces

You can choose one of several communications interfaces to send commands to and receive responses from the Models 707B and 708B. This section shows you how to connect instruments to the following remote communications interfaces:

- Local area network (LAN)
- General purpose interface bus (GPIB or IEEE-488)
- Universal serial bus (USB)

This section describes how to configure and troubleshoot these interfaces on computers with Microsoft™ Windows™ operating systems.

It also describes the I/O software, drivers, and application software that can be used with these instruments.

You can control the Models 707B and 708B from only one communications interface at a time. The first interface on which the instrument receives a message takes control of the instrument. If another interface sends a message, that interface can take control of the instrument.

LAN communications

When you connect using a LAN, you can use a web browser to communicate with the instrument through the internal web page of the instrument and other web apps.

707B and 708B are class C LXI compliant. They are scalable test instruments with direct connections to host computers. They can also interact with a DHCP or DNS server and other LXI compliant instruments on a LAN.

The 707B and 708B are compliant with IEEE standard 802.3 (ethernet) and support full connectivity on a 10 Mbps or 100 Mbps network.

NOTE

Contact your network administrator to confirm your specific network requirements before setting up a LAN connection.

Connect the LAN cable

Connect the LAN connector between the rear panel of the instrument and the host computer or network router. You can use a LAN crossover cable (RJ-45, male/male) or straight-through cable. The instrument automatically senses which cable you have connected.

The location of the LAN connector on the instrument is shown in the following figure.

NOTE

The TSP-Link connectors accept a LAN connection, but are not identified as a LAN and do not connect properly. Be sure to connect the LAN connector correctly.

Figure 26: Model 707B rear-panel LAN connection

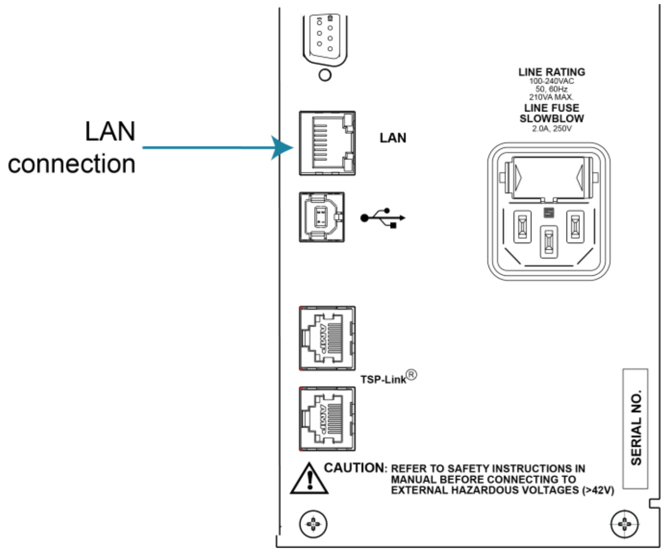
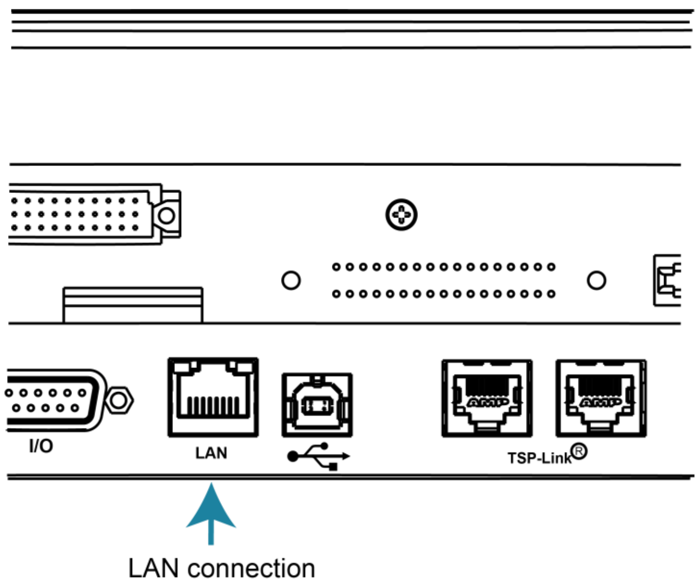


Figure 27: Model 708B rear-panel LAN connection



Configure the network interface card to obtain an IP address automatically

NOTE

Do not change your IP address without consulting your system administrator. Entering an incorrect IP address can prevent your computer from connecting to your corporate network.

Identify and record the existing IP configuration

CAUTION

You are responsible for returning settings to their original configuration before reconnecting the computer to a corporate network. Failure to do this could result in damage to your equipment or loss of data. These settings include, but are not limited to, the IP address, DHCP enabled mode, and the subnet mask.

Record the existing IP configuration information for the computer in the following table so that you can return all settings to their original configuration before reconnecting your computer to a corporate network.

DHCP Enabled	
IP Address	
Subnet Mask	
Default Gateway	
DNS Servers	

Set up automatic IP address selection

If you are connecting to a LAN that has a DHCP server or if you have a direct connection between the instrument and a host computer, you can use automatic IP address selection.

If you select Auto, the instrument attempts to get an IP address from a DHCP server. If this fails, it reverts to an IP address in the range of 169.254.1.0 through 169.254.254.255.

NOTE

Both the host computer and the instrument should be set to use automatic LAN configuration. Though it is possible to have one set to manual configuration, it is more complicated to set up.

To set up automatic IP address selection:

1. Select **Obtain an IP address automatically**.
2. Select **OK**. Close the network settings dialogs.

Configure the instrument to obtain an IP address automatically

1. From the front panel of the instrument, press the **MENU** key.
2. Use the navigation wheel to select **LAN > CONFIG > METHOD**. Press the navigation wheel ◷ to display the METHOD menu.
3. Select **AUTO**. Press the navigation wheel to select AUTO and return to the LAN CONFIG menu.
4. Press the **EXIT** key once to return to the LAN MENU.
5. Select **APPLY**. Press the navigation wheel to apply the setting. The Main Menu is displayed.

Use the LXI Discovery Tool

To find the IP address of the Models 707B and 708B from a computer, use the LXI Discovery Tool, a utility that is available from the Resources tab of the website for the LXI Consortium (lxistandard.org).

LAN reference

When ethernet ports became standard on computers, it was logical that instrumentation would follow. The VXI-11 protocol, which was standardized on in the early 1990s, is the standard used to emulate GPIB over ethernet.

Even though ethernet became the standard LAN technology on instruments, LAN instruments from different vendors differed in the approach they took. Some vendors only supported static IP, whereas others had DHCP, DLLA (Auto-IP), and static addressing. The LXI consortium was started to standardize what should be in all instruments that conform to LXI.

An instrument that conforms to LXI version 1.3 must have the following:

- All three IP addressing modes: DHCP, Auto-IP, and static IP.
- A web server that has some standard ethernet configuration parameters:
  - IP configuration: IP address, subnet mask, gateway.
  - Password protection on anything that might change the instrument state.
  - A control on the web page that flashes an LED or some form of indicator on the front panel of the instrument. LXI calls this the Device Identification Functionality. This allows you to identify the web page you are currently looking at with the instrument. This helps you identify a specific instrument in a rack of similar model instruments.

- A reserved URL in the instrument that provides an xml document that has standard configuration information. This can be useful for software tools that need to identify the instruments and their capabilities. The URL is `http://<host>:<port>/lxi/identification`.
- An IVI driver for the instrument.
- A LAN Status (fault) indicator.
- VXI-11 discovery protocol.
- LAN reset button or menu option. LXI calls this the LAN Configuration Initialize (LCI). When the LXI-defined LAN reset is selected, the instrument reverts its LAN settings to a known set of defaults. The default LAN settings for LXI instruments are:
  - DHCP and Auto-IP enabled. LXI refers to this as the Auto IP address mode (compared to the manual address mode, which is fixed or static IP addressing).
  - Web password is reset to the factory default.
  - Ping responder enabled.
  - Dynamic DNS and mDNS enabled.

Instrument LAN protocols

Raw socket communications

All Keithley instruments with LAN connections support raw socket communications. This means that you can connect to the TCP/IP port on the instrument and send and receive commands. There is no extra protocol overhead beyond what TCP provides. A programmer can easily communicate with the instrument using Winsock on Windows computers or Berkley sockets on Linux® or Apple® computers.

The port number to use for connections for the 707B and 708B is port 5025.

Dead socket connections

If a computer is connected to an instrument through TCP and the computer application is terminated without releasing the socket, it can leave the port on the instrument hanging. You cannot reconnect to it without switching the power to the instrument off and then back on.

To avoid cycling power when this occurs, some instruments have a dead socket port (sometimes known as a backdoor). The dead socket termination port is used to terminate all existing LAN connections. This port cannot be used for command and control functions.

Use the dead socket termination port to manually disconnect a dead session on any open socket. All existing LAN connections are terminated and closed when the connection to the dead socket termination port is closed.

The dead socket termination port for Model 707B or 708B is 5030. When the instrument is connected to this port, when it is disconnected, the dead port is released.

VXI-11

VXI-11 is a LAN protocol that emulates GPIB over ethernet. It uses remote procedure calls to call functions in the instrument for creating a link, sending data, reading data, and so on.

VXI-11 is slower than raw socket communications because there is a small header that indicates how much data is being sent. However, with VXI-11 the programmer or driver writer does not have to confirm that the correct number of bytes have been sent and received.

VXI-11 supports an out-of-band channel, which allows the instrument to signal to the computer that an event, such as an SRQ, occurred.

VXI-11 has a limitation in that it uses broadcast packets to locate the instrument when it wants to make a connection.

707B and 708B use port 1024 for VXI-11 communications. You do not have to know this port number to connect through VXI-11. The discovery portion of the protocol will negotiate the port number for you. If you are trying to configure a firewall, this port number might be useful.

LAN network types

Ethernet is a type of Local Area Network (LAN) that works with a variety of transmission media. The Models 707B and 708B works with a 10/100 BaseT network and use a standard RJ-45 connector. This is an eight-wire connector, but only four wires are used, one pair to transmit and one pair to receive data. A 10BaseT network can accommodate transmission speeds up to 10 Mbits/second. 100BaseT operates at up to 100 Mbits/second. Both types of networks usually require ethernet hubs to make connections. The exception is a one-to-one connection using a crossover cable.

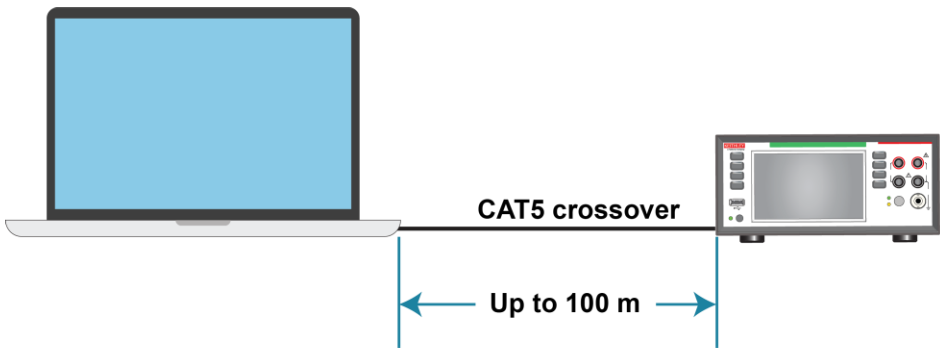
The LAN connector on an instrument gives you more flexibility than GPIB and RS-232 interface controller-subordinate configurations. Rather than connecting the instrument directly to a computer controller in a closed loop, a LAN instrument can be connected to a TCP/IP network using its own subnetwork, or it can be connected directly to an existing network, including a corporate intranet.

One-to-one connection

With most instruments, a one-to-one connection is done only when you are connecting a single instrument to a single network interface card.

A one-to-one connection using a network crossover cable connection is similar to a typical RS-232 system using a null modem cable. The crossover cable has its receive (RX) and transmit (TX) lines crossed to allow the receive line input to be connected to the transmit line output on the network interfaces.

Figure 28: One-to-one connection with a crossover cable



NOTE

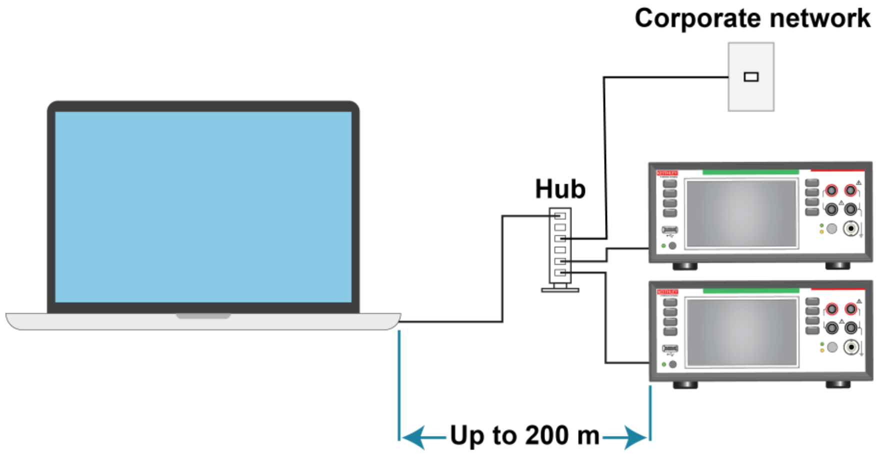
The 707B and 708B supports Auto-MDIX and can use either normal LAN CAT-5 cables (patch) or crossover cables. The instrument automatically adjusts to support either cable.

One-to-many connection

With a LAN hub, a single network interface card can be connected to as many instruments as the hub can support. This requires straight-through network (not crossover) cables for hub connections.

The advantage of this method is easy expansion of measurement channels when the test requirements exceed the capacity of a single instrument. With only the instruments connected to the hub, this is an isolated instrumentation network. However, with a corporate network attached to the hub, the instruments become part of the larger network.

Figure 29: One-to-many connection using a network hub or switch

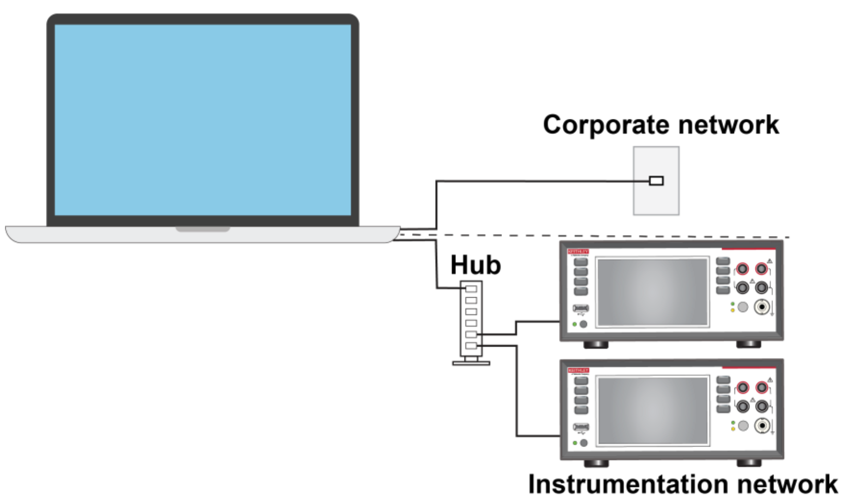


Two network card connection

If you need to connect independent corporate and instrumentation networks, two network interface cards are required in the computer controller. Though the two networks are independent, stations on the corporate network can access the instruments and the instruments can access the corporate network using the same computer.

This configuration resembles a GPIB setup in which the computer is connected to a corporate network, but also has a GPIB card in the computer to communicate with instruments.

Figure 30: Two network card connection

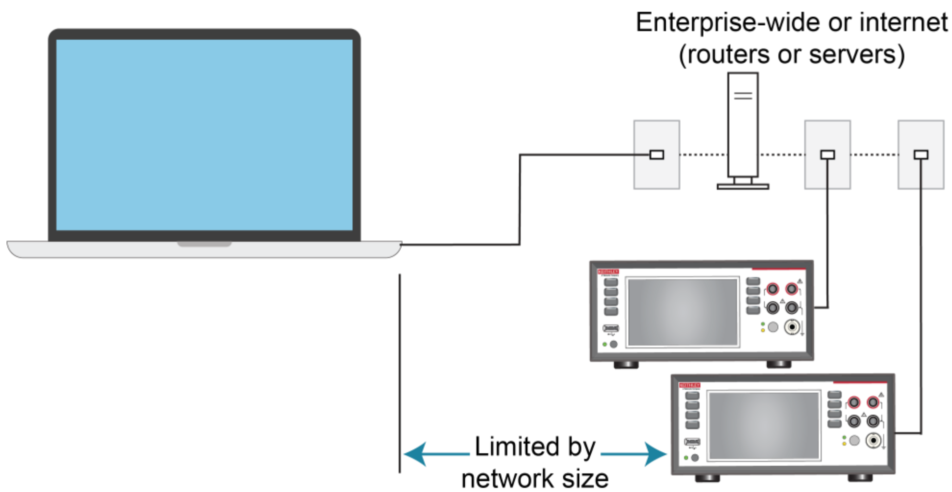


Instrumentation connection to enterprise routers or servers

This connection uses an existing network infrastructure to connect instruments to the computer controller. In this case, you must get the network resources from the network administrator.

Usually, the instruments are kept inside the corporate firewall, but the network administrator can assign resources that allow them to be outside the firewall. This allows instruments to be connected to the internet using appropriate security methods. Data collection and distribution can be controlled from virtually any location.

Figure 31: Instrumentation connection to enterprise routers or servers



Set up an isolated instrument network

The following describes how to set up a simple isolated Class C network for communicating with two LAN instruments using static IP addressing. This network example is similar to the network shown in One-to-one connection (on page 4-8), but without the corporate network connection to the hub.

The standard ethernet hub basically repeats anything it receives from one port, making that data available to all its other ports. Hub connections are made with straight-through cables. The hub is connected to the network interface card in the computer. The network interface card and its driver must be properly installed on the computer according to the manufacturer's instructions. You can also use a switch. The benefit of a switch is that it does not forward network packets out all of the ports. It only forwards those that are being addressed by the packet.

To set up the network:

1. Create IP addresses for the three hosts (the network interface card and two instruments) on the network. This is a Class C network, so the subnet mask is 255.255.255.0. From the table in IPv4 address syntax and subnets (on page 4-14), note that the first three parts of the IP address make up the network ID. For purposes of this example, a network ID of 192.68.1 is used, which is the default network ID of most Keithley instruments.

NOTE

If a corporate network is also connected to the same computer using dual network interface cards, the instrumentation network ID must be different than the corporate network ID.

2. Assign the host ID portions of the three IP addresses. In this example, a host number of 1 is assigned to the network interface card. The first instrument is assigned a host number of 10 and the second instrument becomes host number 20. The complete IP addresses are listed in the following table.

Example host IP address

Card or instrument	IP address
Network interface card	192.68.1.1
First instrument	192.68.1.10
Second instrument	192.68.1.20

3. On the computer, assign the network interface card IP address.
4. Using the front-panel menus, assign a unique IP address to each of the other two instruments. See Set the IP address on the instrument (on page 4-12).

NOTE

It is a good idea to record IP addresses for reference. This is especially important when changing the existing network settings on the computer.

5. Verify that the instruments and the network have been set up and are working properly. You can try logging into the web interface of the instrument to test the connection. See Connect to the instrument web interface (on page 4-40).
6. If you are unable to establish communications, double-check the network settings and try again.

To set the IP address:

1. From the front panel, press the **MENU** key.
2. Use the navigation wheel ◔ to select **LAN > CONFIG > METHOD**. Press the navigation wheel ◔ to display the METHOD menu.
3. Select **MANUAL**. Press the navigation wheel ◔ to select your choice and return to the LAN CONFIG menu.
4. Select **IP_ADDRESS**. Press the navigation wheel ◔ to display the current IP address.
5. If you are:
   - **Connecting directly to a host computer:** Set all but the last three digits to match the IP address of the host computer. Change the last three numbers (after the last decimal point) to a number that is unique on the LAN. The last three digits may be anything from 1 to 254 for a subnet mask of 255.255.255.0.
   - **Connecting to a network:** Enter the address provided by your system administrator.

NOTE

To set the address, turn the navigation wheel to go to the number that needs to change, then press the navigation wheel. Turn the navigation wheel to change the number, then press the navigation wheel to set that number. Repeat this for each number that needs to change.

6. Record the IP address of the instrument.
7. Press the **ENTER** key when the IP address is complete. The LAN CONFIG menu is displayed.
8. Press the **EXIT (LOCAL)** key once to return to the LAN menu.
9. Select **APPLY**. Press the navigation wheel to save the change. The Main Menu is displayed.

TCP/IP network basics

Regardless of the type of network connection used, there must be a way to identify each network device on a network. A software driver installed in the computer provides the means of controlling the instrument. A data communications protocol defines the method of exchanging instructions and data between the computer and each instrument.

CAUTION

When connecting to a corporate network, the network administrator MUST provide all of the network settings for the LAN instrument. Failure to use settings provided by the network administrator could result in failures at other locations on the corporate network. Failure to work through the network administrator could also be considered a breach of company policy. Always consult with the network administrator before attempting to connect instrumentation to the network.

LAN and LXI instruments use the TCP/IP protocol to communicate with other hosts on the network. A host is defined as any device on the network that can transmit and receive IP packets. In addition to the instrumentation, this includes workstations, servers, and routers. Each host on a TCP/IP network is assigned an IP address that is unique to that host.

IPv4 addressing

707B and 708B support IPv4 addressing. There are three ways of assigning an IP address to a host: DCHP, DLLA, and static.

DHCP

The Dynamic Host Configuration Protocol (DHCP) protocol is a way for network devices to request internet protocol (IP) parameters such as the IP address, gateway, subnet mask, and DNS server addresses. Each time a network device connects, it leases the IP address for an amount of time set by the DHCP server. Generally, the time is 24 hours, but it could be shorter or longer depending on how the DHCP server was configured.

Typically, DHCP IP addressing is used for corporate networks. It is also commonly the default on home wireless routers.

One of the benefits of DHCP is that a system administrator does not have to give you an IP address each time you want to connect something to the network. However, one of the drawbacks is that you do not know the IP address that may be assigned to your network device and it can change from connection to connection. There are ways to configure DHCP servers to always give the same network device the same IP address each time. If you need this option, contact your system administrator.

Dynamic Link-local Addressing (DLLA)

DLLA, also called Auto-IP, allows all the network devices to automatically allocate their own unique IP addresses. An Auto-IP address is in the range 169.254.0.0 to 169.254.255.255. A network device randomly picks an address in this range and sends out an ARP packet (on IPv4 only) to see if any other device is using it. If another device responds that it is using it, the network device generates another address and sees if it is in use and so on.

Static or fixed IP address

Static IP addressing means that the person setting up and configuring the network has assigned a fixed, unique IP address to each network device. This requires enforcement to make sure everyone has a unique IP address, but it guarantees that the address will not change.

IPv4 address syntax and subnets

For IPv4, the IP address is 32 bits wide and is divided into two main parts: A network ID number and a host ID number. The address is expressed as four decimal numbers separated by three periods. Valid addresses range from 0.0.0.0 to 255.255.255.255. Each of the four numbers represents the decimal value of the numbers' 8-bit bytes. The way these four numbers are assigned for host ID and network ID depends on the class of network being used.

The network ID must be unique among all network subnets that connect to the internet or corporate intranet. If the subnet will be connected to the public internet, the network ID must be obtained from the Network Information Center, which assigns and preserves unique IDs. In any case, each host ID must be unique among all the hosts on the same network (which presumably has a unique network ID number).

In the TCP/IP protocol, a subnet mask separates the network ID from the host ID. The subnet mask looks like an IP address, but sets a data bit high for each position of the IP address that makes up the network ID. Three different classes of network are defined with the IP address and subnet mask, as shown in the following table.

Network classes defined by IP address and subnet mask combinations

Network class	IP address	Subnet mask	Available subnets	Available hosts
A	*nnn.hhh.hhh.hhh*	255.0.0.0	126	16777214
B	*nnn.nnn.hhh.hhh*	255.255.0.0	16384	65534
C	*nnn.nnn.nnn.hhh*	255.255.255.0	2097151	254

NOTE

In the IP address format, *n* is a network ID position and *h* is a host ID position. For simplicity, the first byte definition has been omitted from the table.

Class C networks are the most common and use the subnet mask 255.255.255.0. The first three bytes are the network ID number and the last byte is the host ID on the network. Host ID numbers 1 through 254 are available for assignment. All hosts on the same isolated network must have the same subnet mask. As a general rule, the top and bottom host numbers are reserved. The top one (*nnn.nnn.nnn.*255) is the broadcast address and the bottom one (*nnn.nnn.nnn.*0) is shorthand for the whole subnet.

DNS

The Domain Name System (DNS) is a protocol that provides a way to associate a user-friendly name to an IP address. For example, while few people know the IP address of the Google website, most people know `www.google.com`. When you enter this URL into an internet browser, the DNS on the network looks up the URL and translates it to the IP address for the Google website.

For DNS to work, there must be a DNS server on the network and the correct IP address for that server must be configured in the computer. Some LAN instruments support DNS. If so, the IP address for both the DNS server and instrument must be configured. Instruments, especially LXI instruments, also have to show a valid hostname on their LXI LAN Welcome and IP Configuration pages, and they need to use the DNS to validate that any hostname they display is valid. If they fail to validate a hostname, they must display the IP address for the instrument or a blank hostname in the hostname field on the web page.

DNS requires a network administrator that can update the database in the DNS server with any host name and IP address combinations, so it is not usually suitable for instrumentation setups.

Dynamic DNS

DNS is a rigid and inflexible system, because you must have a system administrator add the DNS entry to the DNS server. Dynamic DNS tries to address this inflexibility. It addresses the needs of network devices that are powered up and down with IP addresses that can change several times a day.

Dynamic DNS is generally used to refer to a system where there is a DHCP server on the network that allocates IP addresses to the network devices, and therefore there is a method to register a hostname with the DHCP server. The DHCP server assigns the IP address and tracks the hostnames at the same time.

Multicast DNS

Multicast DNS (mDNS) is a protocol that is more suitable than DNS for localized ad hoc networks.

mDNS uses multicast packets for network devices to inform each other of their IP addresses, hostnames, and advertise what services might be available on that device. The packets are usually blocked from going any further than the nearest router. This limits the scope of mDNS, but multicast packets are more network-friendly than broadcast packets. For example, every network device in your subnet receives broadcast packets regardless of need, while for multicast packets, a network device must register for multicast addresses to that will be received.

Multicast packets have IPv4 addresses in the range 224.0.0.0 through 239.255.255.255.

Configure LAN settings through the front panel

All instruments need to be configured before they can be used on a network. The main parameters that need to be configured for IPv4 are:

- IP Addressing mode: 707B and 708B have a choice of Auto or Manual. Auto mode means the instrument tries to get and address through a DHCP server first and if this fails, it reverts to Auto-IP mode. Manual is static IP addressing.
- Subnet Mask.
- Gateway.
- DNS Server address if you are using DNS.

Instruments with front-panel displays and menus allow you to configure the instrument LAN settings through the front panel.

Web connection

You can enable or disable access to the web interface of the instrument.

To enable or disable a web connection:

1. From the front panel, press the **MENU** key, and then select **LAN > ENABLE > WEB**.
2. Select either **ON** or **OFF**. After the power cycle reminder, you return to the LAN CONFIG menu.
3. Press the **EXIT (LOCAL)** key to return to the LAN MENU.
4. Turn the instrument off and on again to finalize the changes.

Telnet connection

The telnet protocol is similar to raw socket and can be used when you need to interact directly with the instrument. Telnet is often used for debugging and troubleshooting. You need a separate telnet program to use this protocol.

The 707B and 708B supports the telnet protocol, which you can use over a TCP/IP connection to send commands to the instrument. You can use a telnet connection to interact with scripts or send real-time commands.

To enable or disable a telnet connection:

1. From the front panel, press the **MENU** key, and then select **LAN > ENABLE > TELNET**.
2. Select either **ON** or **OFF**. After the power cycle reminder, you return to the LAN CONFIG menu.
3. Press the **EXIT (LOCAL)** key to return to the LAN MENU.
4. Turn the instrument off and on again to finalize the changes.

VXI-11 connection

This remote interface is similar to GPIB and supports message boundaries, serial poll, and service requests (SRQs). A VXI-11 driver or NI-VISA software is required. Test Script Builder (TSB) uses NI-VISA and can be used with the VXI-11 interface. You can expect a slower connection with this protocol.

To enable or disable a VXI-11 connection:

1. From the front panel, press the **MENU** key, and then select **LAN > ENABLE > VXI11**.
2. Select either **ON** or **OFF**. After the power cycle reminder, the LAN CONFIG menu is displayed.
3. Press the **EXIT (LOCAL)** key to return to the LAN MENU.
4. Turn the instrument off and on again to finalize the changes.

Raw socket connection

Raw socket is a basic ethernet connection that communicates in a manner similar to RS-232 without explicit message boundaries. The instrument always terminates messages with a line feed, but because binary data may include bytes that resemble line-feed characters, it may be difficult to distinguish between data and line-feed characters.

Use raw socket as an alternative to VXI-11. Raw socket offers a faster connection than VXI-11. However, raw socket does not support explicit message boundaries, serial poll, and service requests.

To enable or disable a raw socket connection:

1. From the front panel, press the **MENU** key, and then select **LAN > ENABLE > RAW**.
2. Select either **ON** or **OFF**. After the power cycle reminder, you return to the LAN CONFIG menu.
3. Press the **EXIT (LOCAL)** key to return to the LAN MENU.
4. Turn the instrument off and on again to finalize the changes.

Check the LAN network settings

You can check the network settings for the instrument without making changes.

To check the network settings:

1. From the instrument front panel, select **MENU > LAN > STATUS**.
2. Use the navigation wheel ◷ to select the following network settings:
   - **IP_ADDRESS:** The IP address that the instrument is using to communicate over the LAN.
   - **GATEWAY:** The gateway address that the instrument is using to communicate over the LAN.
   - **SUBNET:** The subnet mask that the instrument is using to communicate over the LAN.
   - **METHOD:** Automatic or Manual. When this is automatic, the instrument assigns LAN settings automatically. When this is manual, you need to set the LAN settings.
   - **DNS:** The DNS information.
   - **MAC-ADDRESS:** The Media Access Control address of the network interface card.
   - **SPEED:** The instrument automatically detects the speed of the LAN and adjusts its own settings to match.
   - **DUPLEX:** The instrument automatically detects the duplex setting of the LAN and adjusts its own settings to match.
   - **Port:** RAW-SOCKET, TELNET, VXI-11, or DST. Select the port type to see the assigned port number.
   - **Password:** The present password.
3. Press the **ENTER** key to view the setting.
4. Press the **EXIT** key once to return to the STATUS menu.

Set up automatic IP address selection

If you are connecting to a LAN that has a DHCP server or if you have a direct connection between the instrument and a host computer, you can use automatic IP address selection.

If you select Auto, the instrument attempts to get an IP address from a DHCP server. If this fails, it reverts to an IP address in the range of 169.254.1.0 through 169.254.254.255.

NOTE

Both the host computer and the instrument should be set to use automatic LAN configuration. Though it is possible to have one set to manual configuration, it is more complicated to set up.

To set up automatic IP address selection:

1. On the host computer, in the Internet Protocol Properties dialog, select **Obtain an IP address automatically**. This enables DHCP mode. If this fails, the computer will automatically try Auto-IP addressing (DLLA).
2. Select **OK**. Close the network settings dialogs.
3. From the front panel of the instrument, press the **MENU** key.
4. Use the navigation wheel ◷ to select **LAN > CONFIG > METHOD**. Press the navigation wheel to display the CONFIG menu.
5. Select **AUTO**. Press the navigation wheel to select AUTO and return to the LAN CONFIG menu.
6. Press the **EXIT (LOCAL)** key once to return to the LAN MENU.
7. Select **APPLY**. Press the navigation wheel to apply the setting. The Main Menu is displayed.

Set up instrument for manual LAN configuration

After setting up your computer for connection to the instrument, you configure the LAN settings for the instrument through the instrument front panel. Settings include the IP address, subnet mask, and the default gateway.

To set up the LAN on the instrument:

1. From the front panel, press the **MENU** key.
2. Use the navigation wheel ◷ to select **LAN > CONFIG > METHOD**. Press the navigation wheel to display the METHOD menu.
3. Select **MANUAL**. Press the navigation wheel to make a selection and return to the LAN CONFIG menu.
4. Select **IP_ADDRESS**.

5. If you are:
   - Connecting directly to a host computer, refer to the recorded IP address of the computer. Set the IP address to match the IP address of the host computer. Change the last three numbers (after the last decimal point) to a number that is unique on the LAN. The last three digits may be anything from 1 to 254 for a subnet mask of 255.255.255.0.
   - Connecting to a network: Enter the address provided by your system administrator.
6. Record the IP address of the instrument.
7. Press the **ENTER** key when the IP address is complete. The LAN CONFIG menu is displayed.
8. Select **GATEWAY**.
9. Set the gateway value to match the gateway of the host computer.
10. Press the **ENTER** key when the gateway is complete. The LAN CONFIG menu is displayed.
11. Select **SUBNET**.
12. Set the subnet value to match the settings of the host computer or use the value supplied by your system administrator.
13. Press the **ENTER** key. The LAN CONFIG menu is displayed again.
14. Press the **EXIT (LOCAL)** key once to return to the LAN menu.
15. Select **APPLY**. Press the navigation wheel to apply the change. The Main Menu is displayed.

Change DNS settings

On the instrument, you can enable or disable the DNS settings and assign a host name to the DNS server.

When verify is enabled, the instrument performs a DNS lookup to verify that the DNS host name matches the value specified in the DNS server.

When dynamic is enabled, DNS registration works with the DHCP server to register the host name specified with the DNS server.

You can also define additional DNS addresses.

To change DNS settings:

1. From the front panel, press the **MENU** key, and then select **LAN > CONFIG > DNS > VERIFY**.
2. Select either **ENABLE** or **DISABLE**. You return to the DNS menu.
3. Select **DYNAMIC**.
4. Select either **ENABLE** or **DISABLE**. You return to the DNS menu.
5. To set an additional DNS addresses, select **DNS-ADDRESS1** or **DNS-ADDRESS2**.
6. Enter the address.
7. Press the **ENTER** key.
8. Press the **EXIT** key twice to return to the LAN MENU.
9. Select **APPLY**.

Change the IP configuration through the web interface

The LAN settings, such as IP address, subnet mask, gateway, and DNS address, can be changed through the web page of the instrument.

If you change the IP address through the web page, the web page attempts to redirect to the new IP address. In some cases, this may fail. This generally happens if you switch from static IP address assignment to IP address assignment using a DHCP server. If this happens, you need to revert to either using the front panel to set the IP address or use an automatic discovery tool to determine the new IP address.

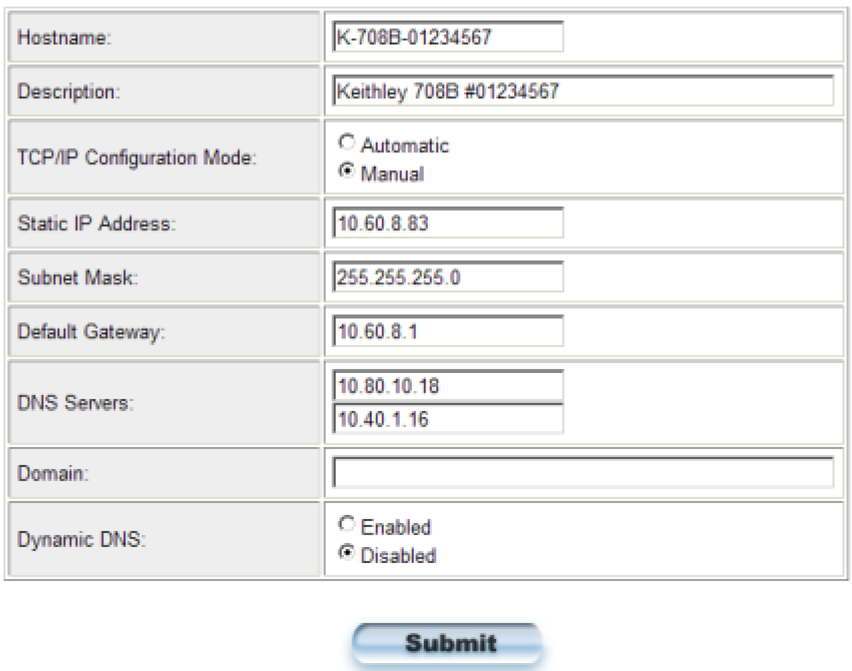
NOTE

You can also change the IP configuration using TSP commands. See LAN and LXI (on page 5-10) for more information.

To change the IP configuration using the instrument web page:

1. Access the internal web page of the instrument (see Using the web interface (on page 4-39)).
2. From the navigation bar on the left, in the LXI Home menu, select **IP Config**.
3. Select **Modify**.
4. You are prompted for a password. The default is `admin`.

Figure 32: Modify IP Configuration page



5. Change the values.
6. Select **Submit**. The instrument reconfigures its settings, which may take a few moments.

NOTE

You may lose your connection with the embedded web interface after selecting **Submit**. This is normal and does not indicate an error or failure of the operation. If this occurs, re-open the web page of the instrument.

GPIB setup

This section contains information about GPIB standards, connections, and address selection.

GPIB standards

The GPIB is the IEEE-488 instrumentation data bus, which uses hardware and programming standards originally adopted by the Institute of Electrical and Electronic Engineers (IEEE) in 1975. The instrument is IEEE Std 488.1 compliant and supports IEEE Std 488.2 common commands and status model topology.

Install the GPIB driver software

Check the documentation for your GPIB controller for information about where to acquire drivers. Keithley also recommends that you check with the manufacturer of the GPIB controller for the latest version of drivers or software.

It is important that you install the drivers before you connect the hardware. This prevents associating the incorrect driver to the hardware.

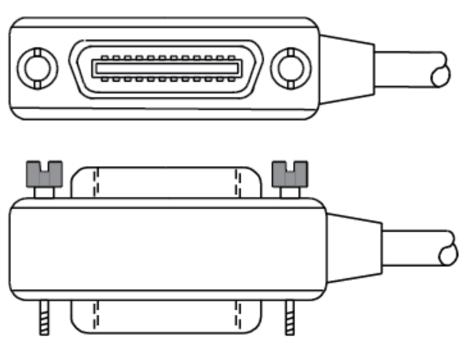
Install the GPIB cards in your computer

Refer to the documentation from the GPIB controller manufacturer for information about installing the GPIB controllers.

Connect the GPIB cable

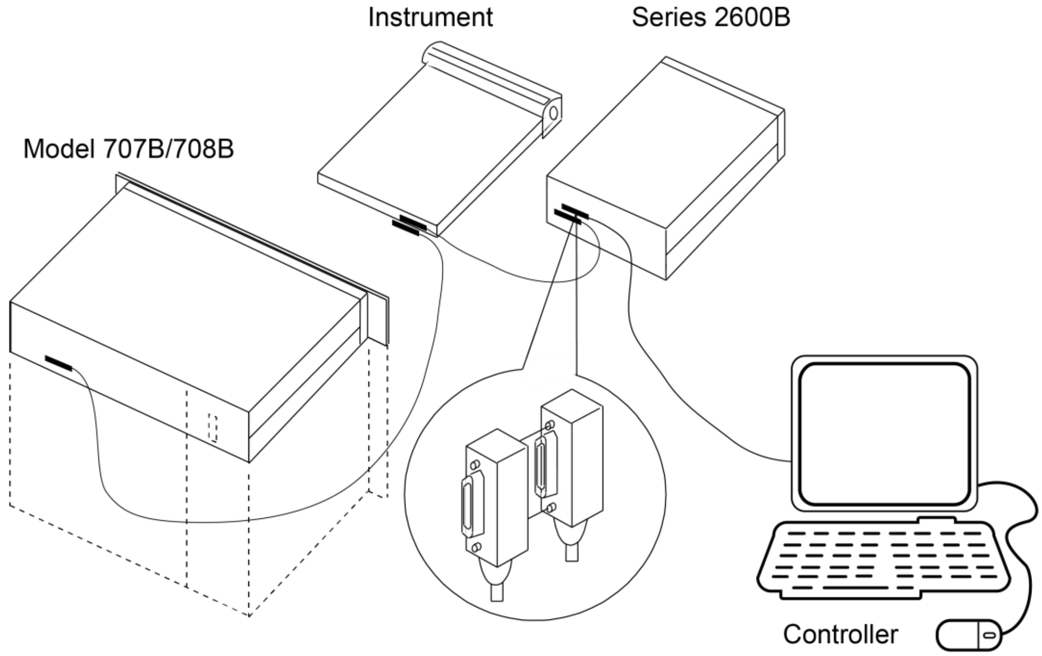
To connect an instrument to the GPIB bus, use a cable equipped with standard IEEE-488 connectors, as shown in the following figure.

Figure 33: GPIB connector



To allow many parallel connections to one instrument, stack the connectors. Each connector has two screws on it to ensure that connections remain secure. The following figure shows a typical connection diagram for a test system with multiple instruments.

Figure 34: IEEE-488 connections



CAUTION

To avoid possible mechanical damage, stack no more than three connectors on any one instrument. To minimize interference caused by electromagnetic radiation, use only shielded IEEE-488 cables. Contact Keithley for shielded cables.

To connect the instrument to the IEEE-488 bus, line up the cable connector with the connector on the rear panel. Install and tighten the screws securely, making sure not to overtighten them. The following figure shows the location of the connector.

Figure 35: Model 707B rear-panel IEEE-488 connection

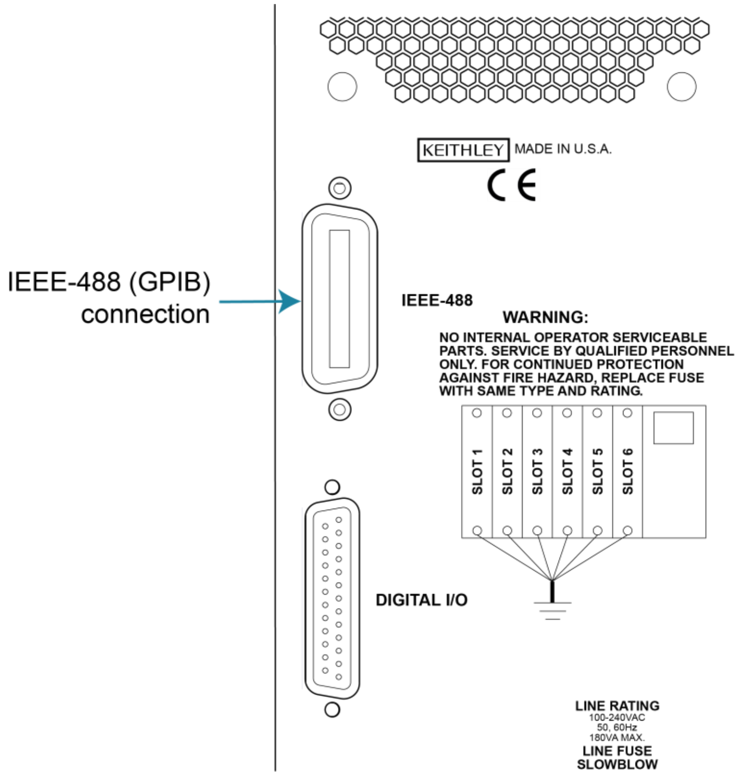
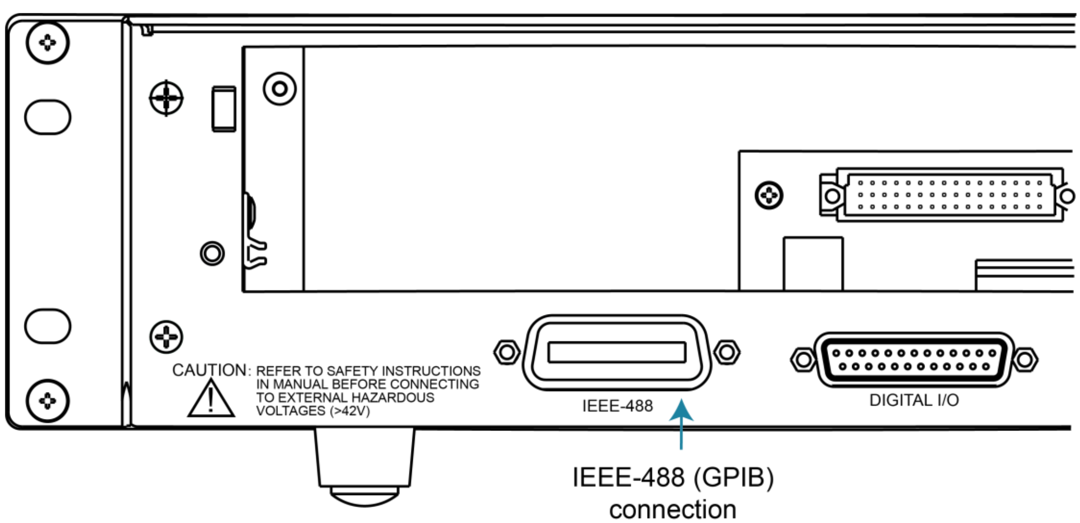


Figure 36: Model 708B rear-panel IEEE-488 connection



Connect any additional connectors from other instruments as required for your application. Make sure the other end of the cable is properly connected to the controller. You can have up to 15 devices connected to a GPIB interface, including the controller. The maximum cable length is the lesser of either:

- The number of devices multiplied by 2 m (6.5 ft)
- 20 m (65.6 ft)

You may see erratic bus operation if you ignore these limits.

Set the GPIB address

The GPIB address is set to 16 at the factory. The address can be set to any address value from 0 to 30. However, the address must be unique in the system. It cannot conflict with an address that is assigned to another instrument or to the GPIB controller.

To change the GPIB address:

1. Press the **MENU** key.
2. Select **GPIB > ADDRESS**. Press the navigation wheel ◷ to display the current address.
3. Choose the appropriate GPIB address.
4. Press **ENTER** to save the address.

The address value is saved in nonvolatile memory and will not change when a reset() (on page 7-142) command is sent or when the power is turned off and then turned on again.

When the GPIB bus is operating, you can use the gpib.address (on page 7-98) attribute to change the GPIB address remotely.

Enable GPIB

By default, the instrument is set to GPIB enabled. You only need to enable it if GPIB control was disabled.

To enable control through the GPIB:

1. Press the **MENU** key.
2. Select **GPIB**. Press the navigation wheel ◷ to display the GPIB MENU.
3. Select **ENABLE**. Press the navigation wheel.
4. To enable GPIB, select **ON**. To disable it, select **OFF**.
5. Press **ENTER** to save the setting.

You must turn the instrument off and on before the setting takes effect.

Communicate with instruments

The GPIB driver software you installed installs an interactive dumb terminal program that allows you to send commands to the instrument. They directly call the GPIB driver support libraries.

For NI™ GPIB controllers, you can use NI-MAX. Start NI-MAX. If your hardware is installed correctly, you should see the controller in the GPIB section of the tree control on the left side. Select it and right-click to see an option to communicate with the instrument.

NOTE

If you want to use the GPIB controller with instrument driver (such as VXIPnP or IVI) or high-level software, you must also install I/O software, which installs the VISA layer. Refer to Install the Keithley I/O Layer.

Address instruments through GPIB

There are two different resource classes in VISA for the GPIB bus.

INSTR is the basic class that everyone uses. It allows application software to send and receive data and commands without dealing with some low-level GPIB nuances. This class is recommended for typical GPIB communications.

The **INTFC** class allows finer control over the GPIB controller card in the computer. You must comply with the IEEE-488.1 protocol and tell the instrument to listen and the controller to talk before sending a message to the instrument. This class allows you to communicate to the instrument using low-level GPIB commands. Refer to your VISA documentation for more details on how to use this class.

The GPIB INSTR resource class format is:

```
GPIB[board]::primary address[::secondary address][::INSTR]
```

Where:

- *board* is the number of the GPIB card, if there is more than one in the computer. If there is only one GPIB card, do not include *board*, but do not leave a space.
- *primary address* is the main GPIB address of the instrument, which can be changed, if necessary, through the front panel of the instrument.
- *secondary address* is for secondary addressing in GPIB. Some instruments have subinstruments or cards inside the main instrument or backplane. The primary address identifies the main instrument. The secondary address identifies subinstruments. Refer to the instrument user manual for the secondary address, if applicable.

Terminator

When receiving data over the GPIB, the instrument terminates messages on any line feed character or any data byte with EOI asserted (line feed with EOI asserted is also valid). When sending data, it appends a line feed character to all outgoing messages. The EOI line is asserted with the terminating line feed character. However, if you want your program to communicate with all I/O buses on the instrument (GPIB, USB, LAN (VXI-11 and raw socket)), it is good practice to add a line feed to the end of the outgoing command. If you use VISA, the same program works with all the I/O buses by changing the resource string in the VISA Open method.

Configure the GPIB controllers

Each instrument on a GPIB bus must have a unique address. You can assign an address from a range of 0 to 30. To avoid address conflicts with GPIB host controllers, do not configure any of the instruments for 0 or 21.

If you need to change the host controller address, consult the controller documentation.

If you are using the KPCI-488LPA and KUSB-488B GPIB controller from Keithley, the configuration utility is called the KI-488 Diagnostic Tool. It is available from the Microsoft™ Windows™ Start menu at **Keithley Instruments > KI-488 > KI-488 Diagnostic Tool**.

If you are using the KUSB-488A GPIB controller from Keithley, the configuration utility is called GPIB Configuration. It is available from the Windows Start Menu at **Keithley Instruments > GPIB-488 > GPIB Configuration**.

USB communications

To use the rear-panel USB port, you need a driver that communicates using the USBTMC protocol, such as NI-VISA, on the host computer.

When installed, the USBTMC protocol allows the Microsoft™ Windows™ operating system to recognize the instrument.

When you connect a USB device that implements the USBTMC or USBTMC-USB488 protocol to the computer, the driver automatically detects the device. Note that the driver does not recognize other USB devices, such as printers, scanners, and storage devices.

The USB can be used for single ASCII-based commands.

In this section, "USB instruments" refers to devices that implement the USBTMC or USBTMC-USB488 protocol.

See Software components for applications and instrument drivers (on page 4-36) for more information on drivers.

Connect the USB cable

To communicate from a computer to the instrument, you need a USB cable with a USB Type B connector end and a USB Type A connector end. You need a separate USB cable for each instrument you plan to connect to the computer at the same time using the USB interface.

To connect an instrument to a computer using USB:

1. Connect the Type A end of the cable to the computer.
2. Connect the Type B end of the cable to the instrument. The location of the USB connector is shown in the following figures.

Figure 37: Model 707B rear-panel USB connection

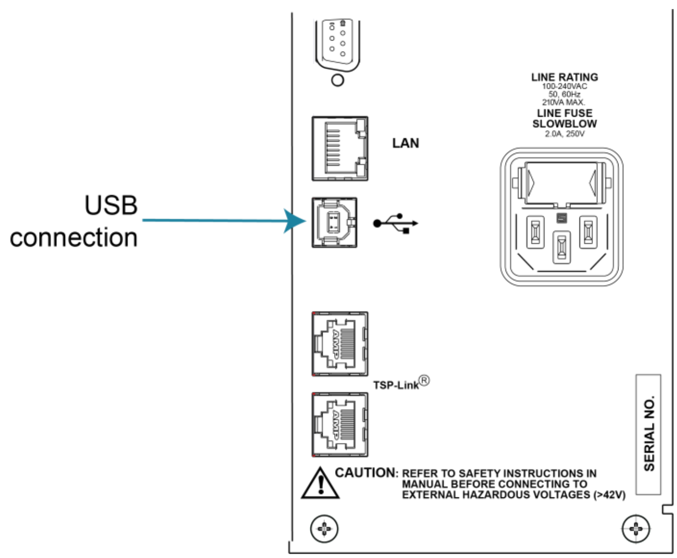
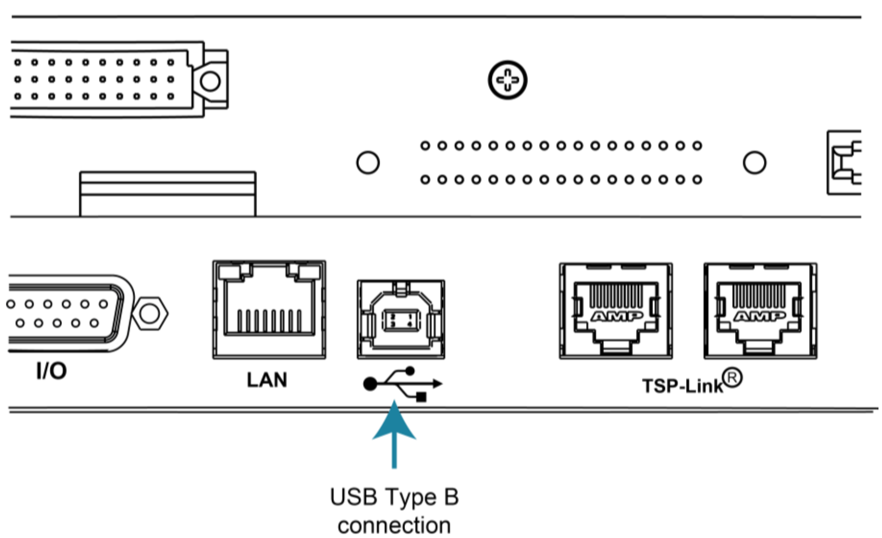


Figure 38: Model 708B rear-panel USB connection



3. Turn on instrument power. When the computer detects the new USB connection, the Found New Hardware Wizard starts.
4. If the "Can Windows connect to Windows Update to search for software?" dialog opens, select **No**, and then select **Next**.
5. On the "USB Test and Measurement device" dialog, select **Next**, and then select **Finish**.

Communicate with the instrument

When using Virtual Instrument Software Architecture (VISA) to communicate with the USB device, you need to use a VISA communications driver. VISA requires a resource string in the following format to connect to the correct USB instrument:

`USB[`*`board`*`]::`*`manufacturer ID`*`::`*`model code`*`::`*`serial number`*`[::`*`USB interface number`*`][::INSTR]`

Where:

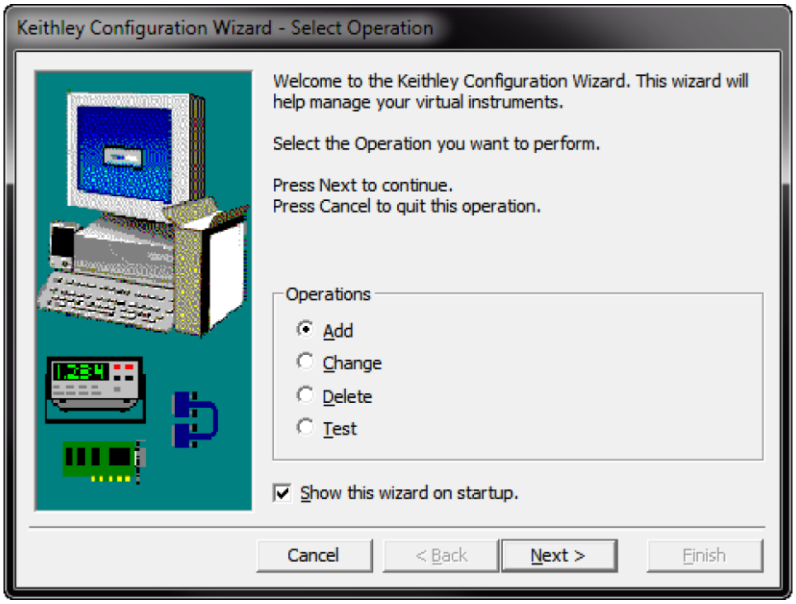
- *`board`* is not used (`0`).
- *`manufacturer ID`* is the USB.org reserved four-digit hexadecimal code for the instrument vendor company. The Keithley hexadecimal code is `05E6`.
- *`model code`* is the model number of the instrument. For example, when addressing a Model 707B, use `707B`.
- *`serial number`* is the serial number of the instrument.
- *`USB interface number`* identifies which USBTMC interface on the instrument to address (usually `0`).

This requires you to determine the parameters. You can gather this information by running a utility that automatically detects all instruments connected to the computer. If you installed the Keithley I/O Layer, the Keithley Configuration Panel is available from the Microsoft™ Windows™ Start menu in the Keithley Instruments menu.

To use the Keithley Configuration Panel to determine the VISA resource string:

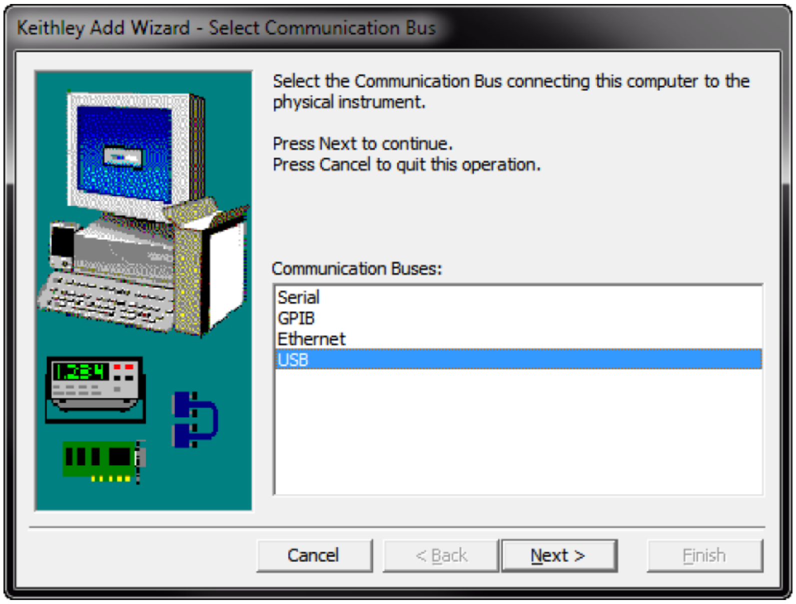
1. Start the Keithley Configuration Panel. The Select Operation dialog is displayed.
2. Select **Add**.

Figure 39: Select Operation dialog



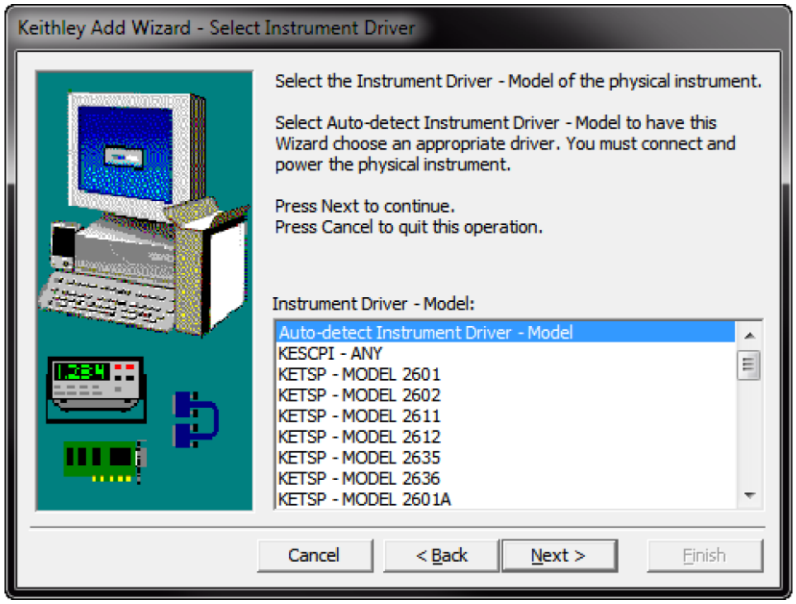
3. Select **Next**. The Select Communication Bus dialog is displayed.

Figure 40: Select Communication Bus dialog



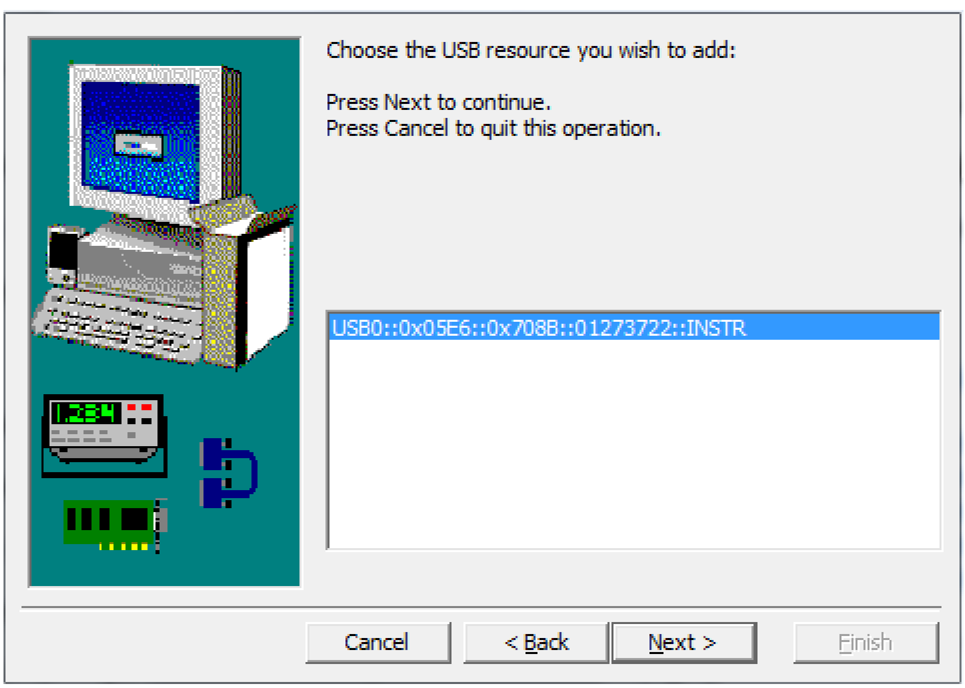
4. Select **USB**.
5. Select **Next**. The Select Instrument Driver dialog is displayed.

Figure 41: Select Instrument Driver dialog



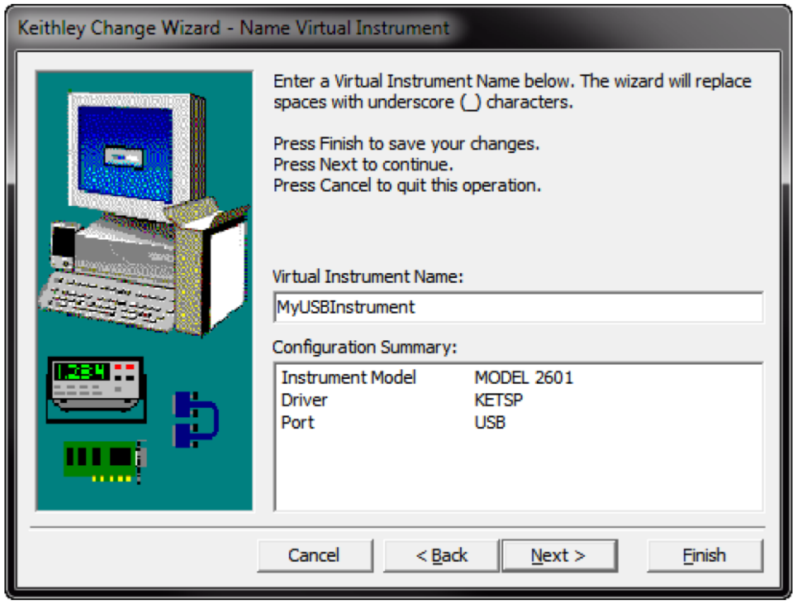
6. Select **Auto-detect Instrument Driver - Model**.
7. Select **Next**. The Configure USB Instrument dialog is displayed with the detected instrument VISA resource string highlighted.

Figure 42: Configure USB Instrument dialog



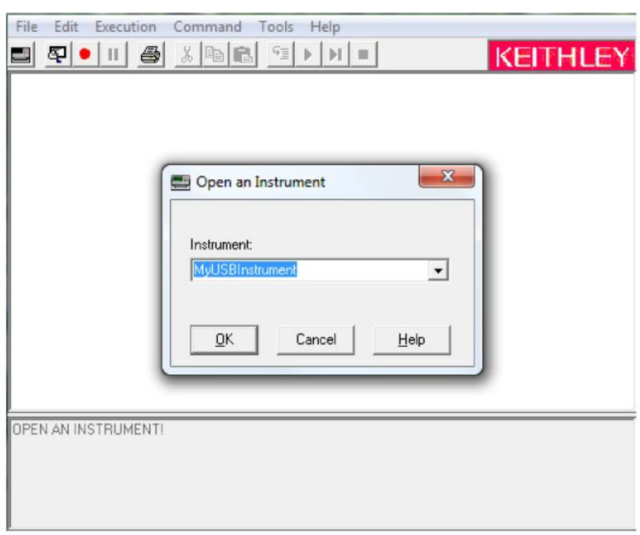
8. Select **Next**. The Name Virtual Instrument dialog is displayed.

Figure 43: Name Virtual Instrument dialog



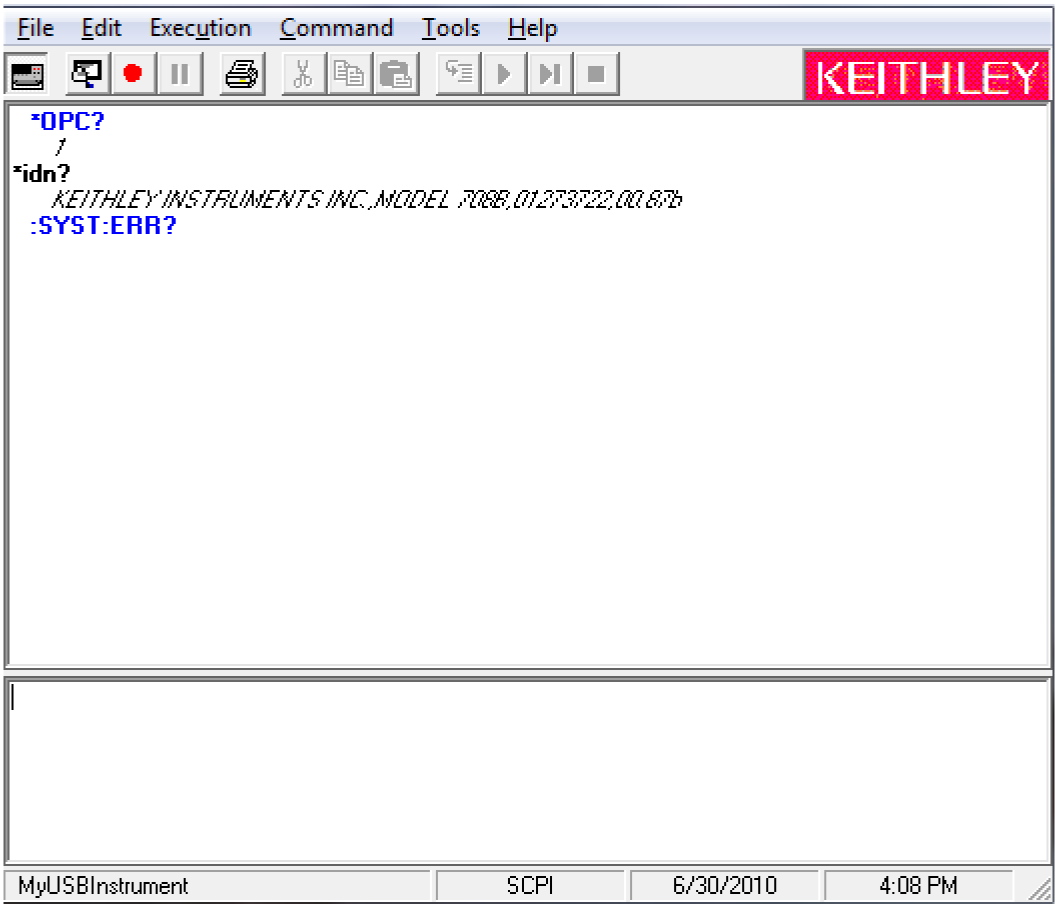
9. For **Virtual Instrument Name**, enter a name that you want to use to refer to the instrument.
10. Select **Finish**.
11. Select **Cancel** to close the wizard.
12. Save the configuration. From the Configuration Utility, select **File > Save**.
13. In the Keithley Communicator, select **File > Open Instrument** to open the instrument you just named.

Figure 44: Keithley Communicator Open an Instrument



14. Select **OK**.
15. Send a command to the instrument and see if it responds.

Figure 45: Send a command to the instrument



NOTE

If you have a full version of NI-VISA on your system, you can run NI-MAX or the VISA Interactive Utility. See their documentation for information.

If you have the Keysight IO Libraries on your system, you can run Keysight Connection Expert to check out your USB instruments. See their documentation for information.

Additional USB information

This section provides further details and more advanced information about the USB bus and test-and-measurement instruments.

Connect multiple USB instruments to the computer

The most convenient way to connect USB instrumentation to the computer is to plug a USB cable directly from the instrument to the computer. If you have more than one USB instrument or have other USB devices, such as printers, keyboards, and mice, you might not have enough USB connectors on the computer.

To gain more ports, you can use a USB hub or add more USB controller cards if you have available PCI or PCI Express slots.

The types of USB hubs that you can use with Model 707B or 708B:

- **Bus powered**: This type of hub draws its power from the USB bus and can only supply 100 mA (USB 2.0) per port.
- **Self powered**: This type of hub has an external power supply and can supply up to 500 mA per port (USB 2.0).

USB VISA identifiers

The USB identifiers to communicate with your Keithley instrument using VISA are:

- **707B:** `USB0::0x05E6::0x707B::[`*`serial number`*`]::INSTR`
- **708B:** `USB0::0x05E6::0x708B::[`*`serial number`*`]::INSTR`

Where:

- `USB0`: USB interface
- `0x05e6`: The Keithley vendor ID (assigned to Keithley by the USB Implementers Forum, Inc.)
- `707B` or `708B`: Instrument model number
- `[`*`serial number`*`]`: The serial number of the instrument. The serial number is also on the rear panel.
- `INSTR`: Use the USBTMC protocol

Software components for applications and instrument drivers

The majority of software applications and all instrument drivers from Keithley depend on the following software components:

- NI-VISA
- VISA shared components
- IVI shared components
- NI CVI runtime engine
- NI IVI compliance package
- Keithley SCPI-based instrument driver

These software components are available for download from the Product Support and Downloads web page (tek.com/support/product-support).

VISA

The Virtual Instrument Software Architecture (VISA) is a standards body that maintains the specifications for a whole series of software components related to instrument connectivity (I/O). The VISA specifications are maintained by the IVI Foundation (ivifoundation.org).

The VISA library (standard VPP-4.3) is a standard for an API to communicate with instruments connected to the computer communications buses, including ethernet, USB, RS-232, and GPIB. VPP-4.3 is a software API standard.

There are three types of programming interfaces to VISA: VISA-C, VISA-COM, and VISA .NET.

- VISA-C is a DLL that has a flat API. The five main message-based functions are viOpenDFLT, viOpen, viWrite, viRead and viClose.
- VISA-COM is an ActiveX type interface to VISA that is more suited for the VB6 and .NET environments.
- VISA .NET is a .NET interface for VISA that the IVI Foundation is presently (July 2010) standardizing on.

VISA shared components

The IVI Foundation provides the VISA shared components, which contain the VISA-COM components that VISA vendors install with their VISA installations. There are separate entries in the Add/Remove Programs dialog for the VISA shared components and the VISA from the vendor.

IVI shared components

The IVI shared components are similar in concept to the VISA shared components. The IVI Foundation provides class drivers for:

- All the supported instruments (DMM, Scope, Fgen, and so on)
- The configuration store

The IVI shared components also create the installation folders and registry keys that all IVI drivers and support files use for installation.

NI CVI runtime engine

IVI-C drivers that are created using the NI™ LabWindows™/CVI environment depend on either the CVI runtime DLL (`cvirte.dll`) or the instrument support runtime DLL (`instrsup.dll`). These DLLs must be present on the system for them to run.

NI IVI Compliance Package

The NI™ IVI Compliance Package (ICP) is a software package that contains IVI class drivers and support libraries that are needed for the development and use of applications that leverage IVI instrument interchangeability. The IVI Compliance Package also is based on and is compliant with the latest version of the instrument programming specifications defined by the IVI Foundation.

The NI ICP installer installs the IVI shared components, CVI runtime engine, and the instrument support runtime engine.

Keithley SCPI-based instrument driver

The Keithley SCPI-based Instrument IVI-C Driver is used to support the Keithley Configuration Panel Wizard and Keithley Communicator functionality. It contains simple functions for opening, configuring, taking measurements from, and closing the instrument.

Keithley I/O layer

The Keithley I/O Layer is a software package that contains several utilities and drivers. It is mainly used as a supplement to IVI drivers.

For additional detail on the Keithley I/O layer, including computer requirements and installation instructions, see tek.com/keithley.

NI-VISA Runtime

NI-VISA is the NI™ implementation of the VISA standard. The Keithley I/O Layer contains a licensed version of the NI-VISA Runtime Engine that contains only the binary files (DLLs) that allow the NI-VISA drivers to operate.

If you already have NI software (such as LabVIEW™ or LabWindows™) installed, you have a valid license that can be used with Keithley drivers and application software.

If you do not have NI software installed, you must install the Keithley I/O Layer to install the drivers.

Keithley Configuration Panel

The Keithley Configuration Panel is a configuration utility for IVI drivers, similar to NI-MAX. It can also autodetect USBTMC instruments and LAN instruments that support the VXI-11 protocol.

Keithley Communicator

The Keithley Communicator is a dumb terminal program that uses VISA to communicate with the instrument.

Address instruments with VISA

VISA allows you to communicate with the instrument on different communications buses by changing a resource string that gets passed in with the viOpen function, in VISA-C, or with the Open method on the VISA-COM resource manager object.

For detailed information about the format of the resource string, refer to the VISA specification VPP4.3 at the IVI Foundation website, or refer to the help file provided by the vendor of the VISA implementation you are using.

The following sections describe the resource strings for some of the communications types that Keithley supports. Any field that has `[ ]` (square brackets) around it is optional and will revert to a default value.

Address instruments through the LAN

VISA supports two different LAN protocols, each of which has a different resource string.

VXI-11 is a protocol that emulates GPIB over the LAN. 707B and 708B supports this protocol. The resource string is:

```
TCPIP[board]::host address[::LAN device name][::INSTR]
```

`board` is the network interface card in the computer. This value is usually skipped and VISA determines the correct network interface card (if you have more than one) by looking at the IP address.

`host address` can be either a valid DNS hostname, mDNS hostname, or the IPv4 IP (only) address of the instrument.

`LAN device name` is a method of addressing secondary instruments at the main IP address, similar to secondary addressing on the GPIB bus. The default is `inst0`.

A **raw socket** connection requires more work by the driver or application program to make sure the correct amount of data has been sent or received correctly. All Keithley instruments support the raw socket connection.

```
TCPIP[board]::host address::port::SOCKET
```

The `board` and the `host address` are the same as for the VXI-11 protocol.

`port` is the port to which to connect on the instrument. For the 707B and 708B, the port is 5025. See Instrument LAN protocols (on page 4-6) for a complete list of port numbers.

Use the web interface

The Model 707B or 708B web interface allows you to review instrument status, control the instrument, and upgrade the instrument over a LAN connection.

The instrument web page resides in the firmware of the instrument. Changes you make through the web interface are immediately made in the instrument.

The Model 707B or 708B web interface can be used with your choice of web browsers.

All examples in this manual can be run through the TSB Embedded (on page 4-53) web application that is available on the instrument web interface.

Connect to the instrument web interface

To connect to the instrument web interface, you must have an LAN connection from the computer to the instrument.

The web interface requires the web browser plug-in Sun Java™ Runtime Environment Version 6 or higher. Installation files are available from Java download site (java.com/download/manual.jsp).

The ActiveX control and Java applets are installed from the instrument but, depending on the browser security settings, they may require the users permission to be downloaded and installed.

To find the IP address of the Models 707B and 708B from a computer, use the LXI Discovery Tool, a utility that is available from the Resources tab of the website for the LXI Consortium (lxistandard.org).

After the instrument is connected and Java is installed, to connect to the instrument:

1. Connect the 707B and 708B to the LAN and confirm that the LAN light on the instrument is illuminated.
2. To find the IP address, from the front panel of the instrument, select **MENU** > **LAN** > **STATUS** > **IP-ADDRESS**.
3. Open an internet browser.
4. In the Address box of the browser, enter the IP address of the instrument.
5. Press **Enter**.

The Home page of the instrument web interface is displayed.

Identify the instrument

If you have a bank of instruments, you can select the ID button to determine which one you are communicating with.

To identify the instrument:

1. In the middle of the left side of the Home page, select the **ID** button. The button turns green and the LAN status indicator on the instrument blinks.
2. Select **ID** again to return the button to its original color and return the LAN status indicator to steady on.

Web interface home page

The home page of the web interface gives you basic information about the instrument, including:

- The instrument model, serial number, firmware revision, and LXI information
- A list of slots and matrix cards that are installed in each slot
- An **ID** button to help you locate the instrument
- Links to the instrument web options, including TSB Embedded

Log in to the instrument

The web interface has both interactive and read-only pages. These pages are always listed in the navigation panel on the left side of the web interface. You can review information on any of the pages without logging in, but to change information, you must log in.

Pages that contain information you can change include a **Login** button. Once you have logged in to one page of the web interface, you do not need to log in again unless you reload the page.

To log into the instrument:

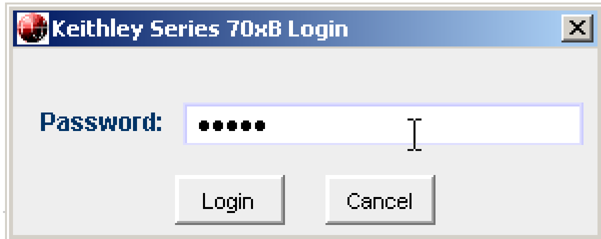
1. Open a page that contains a Login button, such as one of the Cards pages, Scan Builder, or TSB Embedded.

Figure 46: Log in

2. Select **Login**. The login dialog is displayed.

   Enter the password (the default is **admin**).

Figure 47: Enter password



3. Select **Login**.

NOTE

The default password is **admin**. If the password has been changed, it is available from the front panel of the instrument. Press **MENU** > **LAN** > **STATUS** > **PASSWORD**.

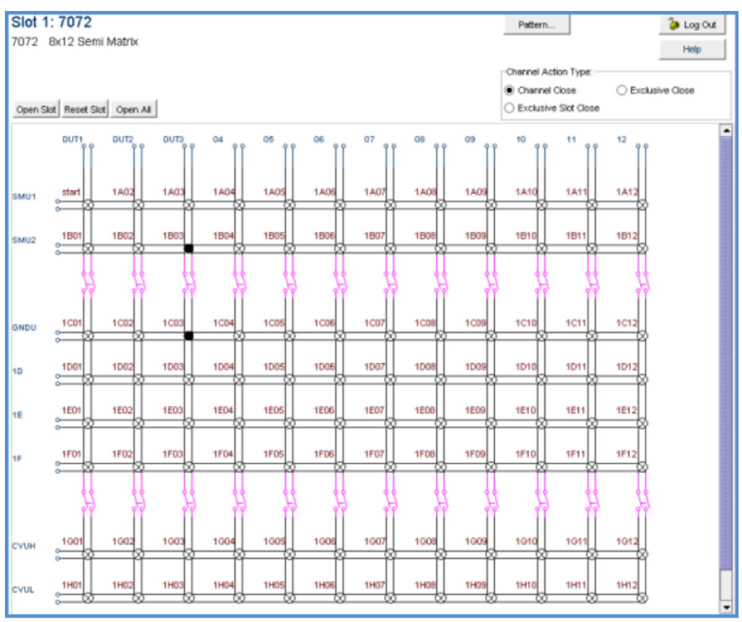
Card pages

The card pages are interactive pages where you can work with channels in each slot.

To open a card page, on the left navigation, select the slot number.

There is a specific page for each card installed in the mainframe. The page displays a grid that shows the relay configuration of the matrix card, as shown in the following figure.

Figure 48: Web interface cards page



Open and close slots from the card pages

You can open and close channels from the card pages in several ways.

The simplest method is to select a connection. The channel changes state to open or closed. When the channel is open, the connection looks similar to one of the following graphics (the actual item on the web interface depends on the installed card):

Figure 49: Web interface open channel

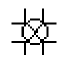
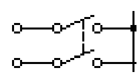


Figure 50: Web interface relay open



When the channel is closed, the connection will look similar to one of the following:

Figure 51: Web interface closed channel

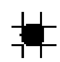
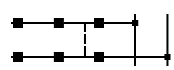


Figure 52: Web interface relay closed



To specify the type of close, select a Channel Action Type from the box in the upper right before closing a channel. The options are:

- **Channel Close**: Close the selected channel without affecting the state of any other channels.
- **Exclusive Slot Close**: Close the selected channel and open any closed channels in the same slot.
- **Exclusive Close**: Close the selected channel and open any closed channels in the instrument (the only closed channel is the selected channel).

You can open all channels in a slot by selecting **Open Slot**.

You can open all channels in the instrument by selecting **Open All**.

For more information on opening and closing channels, see Work with channels
(on page 2-1).

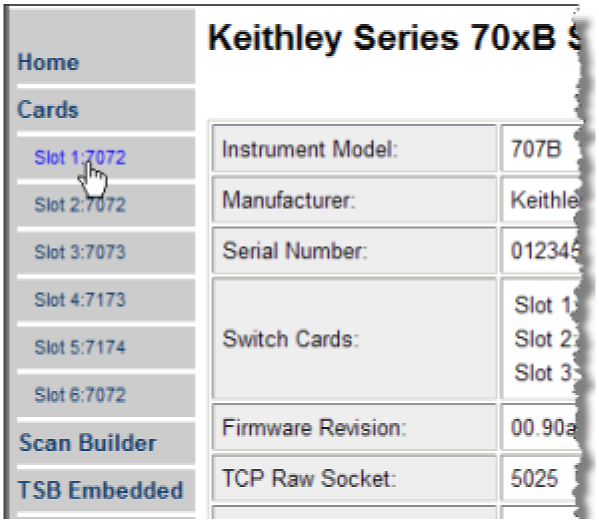
Exercise: Close multiple channels using the web interface

This exercise describes how to close channels and channel patterns using the web interface.

To close multiple channels using the web interface:

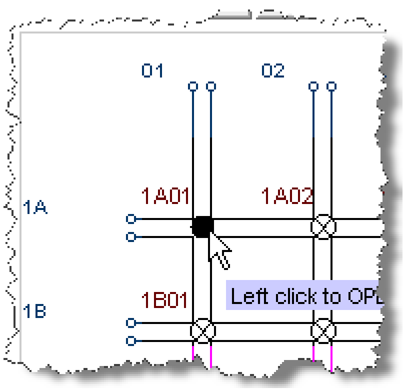
1. The instrument home page is displayed. From the navigation on the left, select the slot that contains the channels you want to close.

Figure 53: Select the slot



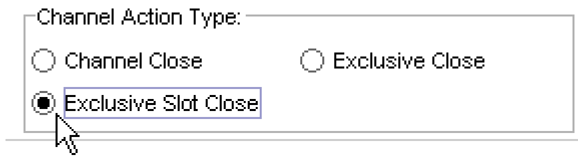
2. Select **Login**. The login dialog is displayed.
3. Enter the password (the default is **admin**).
4. Select **Login**.
5. To close channel `1A01`, select the channel.

Figure 54: Close a channel



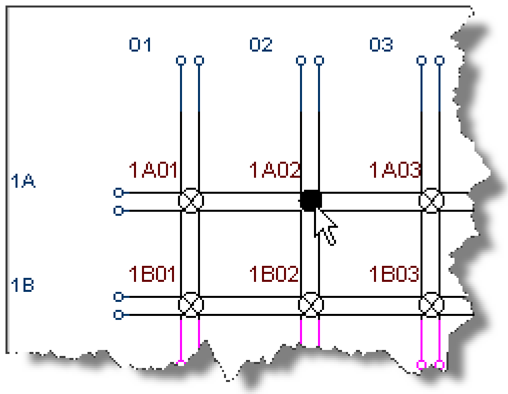
6. To perform an exclusive close on channel `1A02`, select **Exclusive Slot Close** in the Channel Action Type box.

Figure 55: Select exclusive slot close



7. Select `1A02` to close that channel and open all other channels.

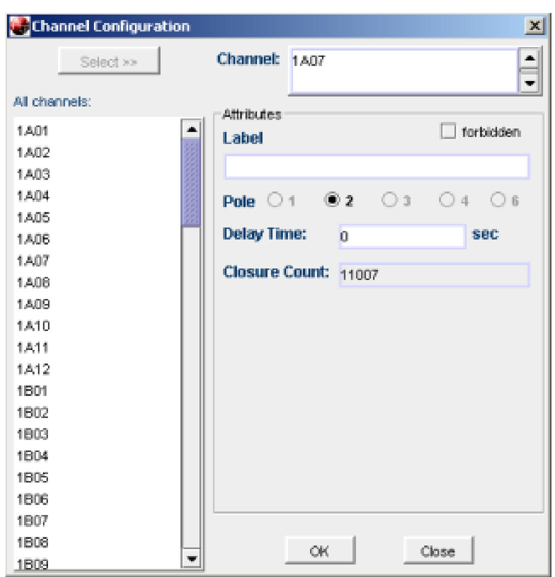
Figure 56: Close a channel with exclusive close



Configure channels from the web interface

To configure channels from the web interface, right-click the channel. The Channel Configuration dialog is displayed.

Figure 57: Channel configuration dialog



In this dialog, you can set:

- **Label:** The label for the channel. This is the same as the command `channel.setlabel()`.
- **Forbidden:** Select to set the channel to forbidden. This prevents the channel from being closed from any interface. If the channel is used in a channel pattern, the pattern is deleted when you set the channel to forbidden to close.
- **Delay Time**: The additional delay to incur after the relay settles when closing the channel. Enter the value for the delay in seconds. The total delay for channel close is this delay plus the relay settling time.

This dialog also displays the closure count. See Determining the number of relay closures (on page 2-6) for information.

Set up channel patterns from the web interface

You can use channel patterns as a convenient way to refer to a group of switching channels with a single alphanumeric name. When you perform close or open operations on a channel pattern, only the channels that are in the channel pattern are affected.

There is no speed advantage in performing close and open operations on channel patterns compared to performing the same operations on individual channels or a list of channels.

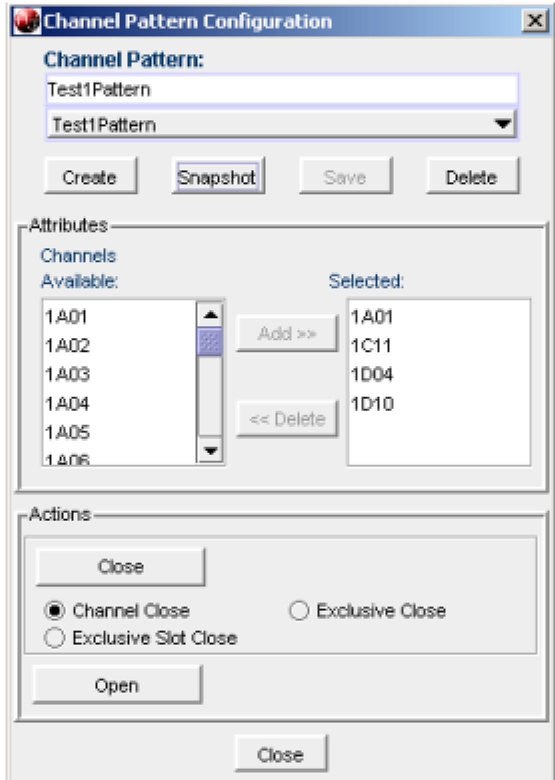
Exercise: Create channel pattern "Test1Pattern" using the Snapshot feature

To create a channel pattern "Test1Pattern" from the web interface using the Snapshot feature:

1. From the left navigation, select **Slot 1**.
2. Close the channels that you want to include in the pattern. For this example, close channels **1D12**, **1C11**, **1B10**, and **1A09**.
3. Select **Pattern** (above Channel Action Type). The Channel Pattern Configuration dialog is displayed, as shown in the following figure.
4. Type **Test1Pattern** for the pattern name in the box at the top of the dialog.

   Select **Snapshot**. A new pattern is created that contains the closed channels.

Figure 58: Channel pattern configuration dialog



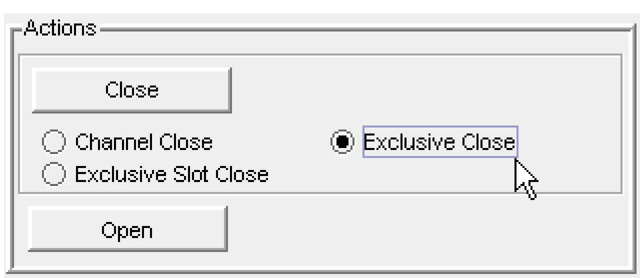
Exercise: Close channels in "Test1Pattern" from the web interface

To close channels in channel pattern "Test1Pattern" from the web interface:

1. Select **Pattern** (above Channel Action Type). The Channel Pattern Configuration dialog is displayed.
2. Select **Test1Pattern** from the list at the top of the dialog.

   In the Actions area, select the type of close you want to perform.

Figure 59: Close channels in a pattern



3. Select **Close** to close the channels.
4. When changes are complete, select **Close** at the bottom of the dialog to exit.

To delete a channel pattern from the web interface:

1. Select the name of the pattern that you want to delete in the Channel Pattern Configuration dialog.
2. Select **Delete**.
3. For more information about patterns, including opening and closing the channels that are in patterns, see Channel patterns (on page 2-14).

Reset a slot from the web interface

You can reset all the relays in the displayed slot by selecting **Reset Slot**.

When you reset the relays in a slot:

- Any closed channels open
- Labels return to default of slot, row, column
- Delays are set to zero
- If the channel is forbidden to close, it is cleared from being forbidden to close
- If any of the channels of the slot are in channel patterns, the patterns are deleted

The rest of the instrument settings are unaffected.

Scan Builder page

The Scan Builder page of the web interface allows you to set up and run scans and triggers.

A scan is a series of steps that opens and closes switches sequentially for a selected group of channels. During each step, actions occur, such as waiting for a trigger, taking a measurement on an external instrument, and completing a step count. Scans automate actions that you want to perform consistently and repeatedly on a set of channels.

Triggers are events that prompt the instrument to move from one step to another in a scan. Triggers can come from a variety of sources, such as a key press, digital input, or expiration of a timer. The sequence of actions and events that occur during the scan is called the trigger model.

Scanning and triggering allow you to synchronize actions across channels. You can set up a scan using the trigger model to precisely time and synchronize the Model 707B or 708B between channels and multiple instruments. You can also use triggers without the trigger model to set up a scan to meet the needs of a specific application that does not fit the trigger model.

NOTE

If you use Scan Builder to create a scan, use the options in the Scan Builder page to run the scan. Using the TSB Embedded page may not give you the expected results.

Create a scan list

Before you can run a scan, you must create a scan list. A scan list is a set of steps that runs in a specific order during a scan. Each step contains a channel, channels, or channel patterns that you want to measure in that step. Each step is acted on separately during the scan.

You can mix channel patterns and individual channels in a scan list. The steps are executed in the order in which they are added to the scan.

NOTE

Before setting up a scan list, make sure your channels and channel patterns are configured. Refer to Work with channels (on page 2-1) for detail.

If you change the channel configurations or channel patterns after the scan list has been set up, you may not see expected results. If the change prevents the scan from functioning properly (such as deleting something referenced by the scan), an error message is logged.

Exercise: Create a scan list from the web interface

In the following exercise, create a scan list containing channels **1A01**, **1B02**, **1C03**, **1D04**, **1E05**, and the pattern **Test1Pattern**.

To create the scan list from the web interface:

1. From the left navigation of the web interface home page, select **Scan Builder**.
2. In the Add Channel By list on the right, select **Number** to add the channels. Select the channel numbers from the list. You can use **Ctrl+click** to select multiple channels and **Shift+click** to select a range of channels. To remove your selections from the Add Channel By list, select **Clear Channel Selection**.
3. Select **Add Step**. The channels are added to the Steps list.
4. In the Add Channel By list on the right, select **Pattern**.
5. Select **Test1Pattern** from the Channel Pattern list.
6. Select **Add Step**. The pattern is added to the Steps list.
7. In the Scan Count box, enter the number of times you want to repeat the steps in the scan. For example, enter **5**.

Clear the scan list from the web interface

Clearing the scan list deletes all channels and channel patterns from the scan list.

To clear the scan list from the web interface:

1. From the left navigation area of the web interface home page, select **Scan Builder**.
2. Select **Scan Clear**.

Review the scan list

You can review the existing scan list to see which channels and channel patterns are listed, and in which order.

To review the scan list from the web interface:

1. From the left navigation of the web interface Home page, select **Scan Builder**.
2. Select the **Build & Run** tab. The scan list is shown in the Steps box.

Reset the scan list

You can clear the scan list and return scan settings to their factory defaults using scan reset. A scan reset does not affect any settings in the instrument except the scan list and trigger model.

The settings that are affected are:

- Channels and patterns are removed from the scan list
- **Bypass:** Returned to default setting of ON
- **Mode:** Returned to default setting of Open All
- **Scan count:** Returned to default setting of `1`
- **Trigger to start scan:** Set to Immediate
- **Trigger to continue channel action for each scan step:** Setting is cleared
- **Arm (Scan Start Stimulus):** Set to None
- **Channel Action Stimulus:** Set to Channel Ready Event
- **Channel Ready Event:** Set to None
- **Scan Complete Event:** Set to None

To reset the scan list from the web interface:

1. From the left navigation of the web interface Home page, select **Scan Builder**.
2. Select **Scan & Trigger Reset**.

Stop the scan

To stop the scan from the web interface:

On the Build & Run tab, select **Abort**.

Monitor the state of the scan

To monitor the state of the scan, you can select **Query State** on the Build & Run tab. **Query State** displays the present state of the scan, which can be:

- **Empty:** No scan defined
- **Building:** Scan list is being created
- **Running:** Scan in process
- **Success:** Scan completed successfully

Set up simple triggers

You can set up triggers to control your scan using the options on the Simple Trigger tab. You can set:

- The event that starts the scan
- The time interval event that controls the channel action for each step of the scan

To see these options, select the **Simple Trigger** tab from the top of the Scan Builder page.

Trigger to start scan

You can choose the triggers used to start the scan. The options to start the scan are:

Immediate: When Immediate is selected, the scan starts as soon as you select **Execute Background** on the Build & Run tab. Select Immediate when you do not have trigger requirements to start the scan. This is the default selection.

Digital Input: When selected, you select a digital line (1 to 14) that is used to start a scan. You can select falling or rising for the digital input. Falling selects the falling edge trigger. Rising selects the rising edge of the trigger.

NOTE

If Other is displayed in the mode list, a different mode (other than falling or rising) is already selected. Other is not a mode and cannot be selected. It is only an indicator that the digital triggering is already set up for a different mode. See Advanced triggering (on page 4-52) for other options.

Channel action trigger

You can select the trigger to continue channel action for each scan. The options to continue channel action are:

Immediate: When immediate is selected, the scan immediately steps to the next channel in the scan list. This is the default setting.

Digital Input: When selected, you select a digital line (1 to 14) that is used to trigger the instrument to step to the next channel. You can select falling or rising for the digital input. Falling selects the falling edge trigger. Rising selects the rising edge of the trigger.

NOTE

If Other is displayed in the mode list, a different mode (other than falling or rising) is already selected. Other is not a mode and cannot be selected. It is only an indicator that the digital triggering is already set up for a different mode. See Advanced triggering (on page 4-52) for other options.

Every *N* seconds: This parameter adds a fixed delay between each channel. The delay occurs before the next channel in the scan list is closed.

Advanced triggering

The Advanced Trigger tab of the Scan Builder allows you to set the options that are available from the Simple Trigger tab and more sophisticated options to control scan triggering.

The Advanced Trigger tab uses the trigger model flowchart to help you visualize and define the input and output triggers to the scan.

For more information on the trigger model, see Trigger model (on page 2-21).

The options on the Advanced Trigger tab include:

- **Mode**: Select Open All to open all slots before the scan starts. Select Open Selective to open only channels that are involved in scanning; closed channels that are not involved in scanning remain closed. Select Fixed ABR to open all channels involved in the scan, but close all required backplane relays before the scan.
- **Arm (Scan Start) Stimulus**: Select the event that causes the arm event detector trigger to be set to the detected state (the scan can begin).
- **Channel Action Stimulus**: Select the event that causes the channel event detector to be set to the detected state (the step can begin).
- **Channel Ready Event To**: Select the recipient of the Channel Ready Event.
- **Scan Complete Event To**: Select the recipient of the Scan Complete Event.

There is also a **Config** button available for each of the options except Mode. When you select **Config**, a dialog with additional options for the selection is displayed.

Set the scan mode

The scan mode determines how channels are opened before the start of the scan.

You can select:

- **Open all:** All slots are opened.
- **Open select:** All channels selected in the scan list are opened; any closed channels remain closed if they are not in the scan list.
- **Fixed ABR**: All necessary backplane relays are closed before the scan.

To set the scan mode from the web interface:

1. Select the **Advanced Trigger** tab.
2. Select **Mode**.
3. Select **Open All** or **Open Selective**.

TSB Embedded

TSB Embedded is an application that includes a command line interface that you can use to issue TSP commands. It also offers script-building functionality. TSB Embedded resides in the instrument.

Script management options

Existing scripts are listed in the User Scripts box on the left side of TSB Embedded. Script management options are:

- **Run a script:** Select the name of the script and then select **Run**.
- **Delete a script:** Delete the name of the script and select **Delete**. The script is deleted from the User Scripts list and from the nonvolatile memory of the instrument.
- **Stop operation of a script:** Select **Abort Script**.
- **Export the selected script to the computer:** Select **Export to PC**. Choose the directory in which to save the script and select **Save**. Scripts are saved to a file with the extension `tsp`. TSP files are native to Test Script Builder or TSB Embedded, but they can be opened and edited in any text editor.
- **Import scripts from the computer:** Select **Import from PC**. Select the directory that contains the file. You can only import files with the extension `tsp`.
- **Clear the name box and the box that contains the script:** Select **Clear**.

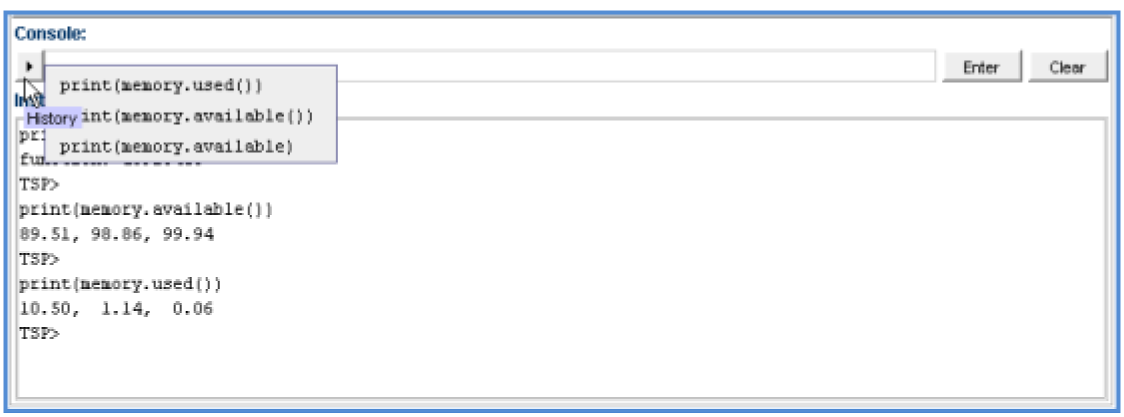
View the contents of a script: Type the name of a script in the TSP Script box and select **View Script**.

Command line interface

Console: Enter command line entries here to send commands to the instrument. Select **Enter** to send the command. The commands and output are shown in the Instrument Output box.

To resend a command, select the button at the left of the Console box.

Figure 60: Web interface console



Instrument control

To reset the entire TSP-enabled system, including the controlling node and all subordinate nodes, select **Reset**.

Create a script using TSB Embedded

NOTE

If you are using TSB Embedded to create scripts, you do not need to use the commands `loadscript` or `loadandrunscript` and `endscript`.

You can create a script from the instrument web interface with TSB Embedded. When you save the script, it is loaded into the runtime environment and saved in the nonvolatile memory of the instrument.

To create a script using TSB Embedded:

1. In the **TSP Script** box, enter a name for the script.
2. In the input area, enter the sequence of commands to be included in the script.
3. Select **Save Script**. The name is added to the User Scripts list on the left.

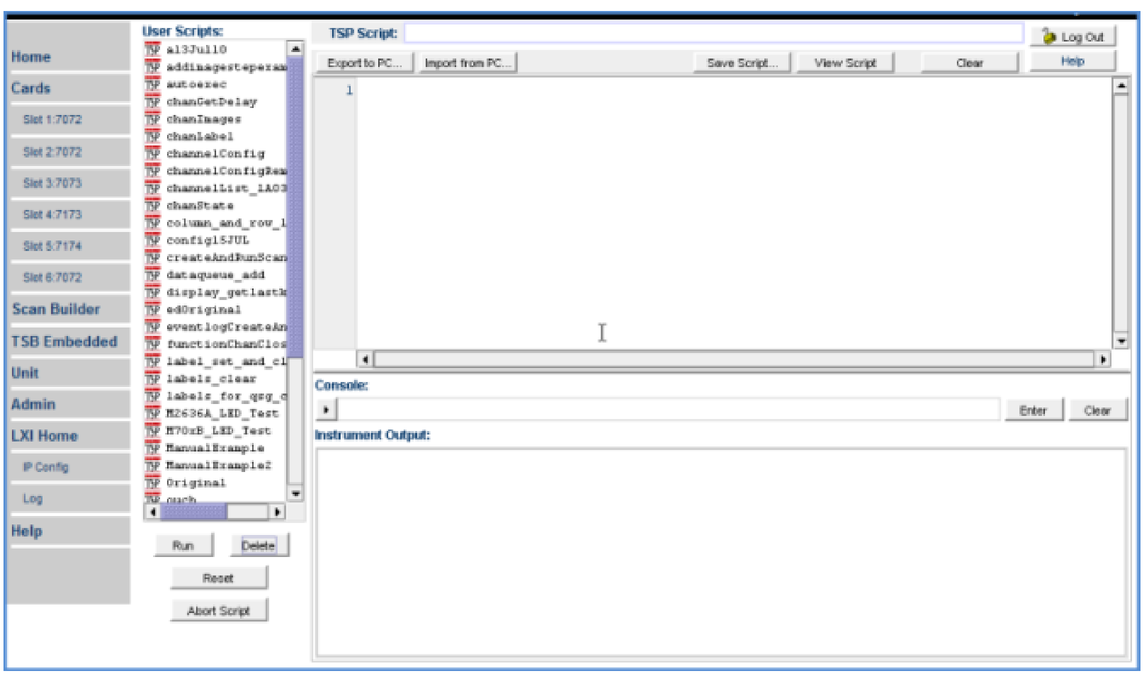
Exercise: Create and run a script with TSB Embedded

This exercise script assigns row and column labels for the card installed in slot 1. It then commands the 707B or 708B to close a crosspoint specified by the row and column labels.

To create and run a script with TSB Embedded:

1. From the navigation area on the left side of the web interface, select **TSB Embedded**. The TSB Embedded page is displayed.

Figure 61: Select TSB Embedded



2. You must log in to the instrument to use TSB Embedded. After logging in, you can access the options on the TSB Embedded page.
3. To create the example script, enter the name, **column_and_row_labels**, in the TSP Script box.

Figure 62: Enter script name

4. Enter the following code in the box below the buttons:

```
channel.setlabelrow("1A01", "SMU1")
channel.setlabelrow("1B01", "SMU2")
channel.setlabelrow("1C01", "GNDU")
channel.setlabelrow("1G01", "CVUH")
channel.setlabelrow("1H01", "CVUL")
channel.setlabelcolumn("1A01", "DUT1")
channel.setlabelcolumn("1B02", "DUT2")
channel.setlabelcolumn("1C03", "DUT3")
channel.open("allslots")
channel.close("SMU1+DUT3")
```

NOTE

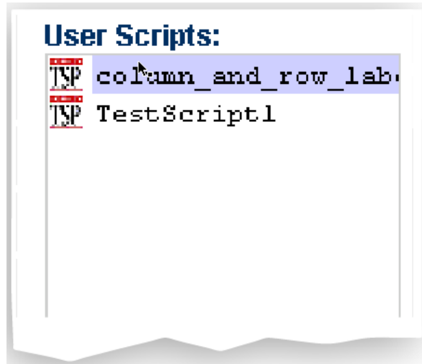
Commands and parameters for Models 707B and 708B are case-sensitive. It is important to type in the commands exactly as shown to avoid any syntax and execution errors.

5. Select **Save Script**. The script is added to the User Scripts list.

NOTE

Standard edit functions, such as Copy, Cut, and Paste, work within TSB Embedded.

Figure 63: Script added to User Scripts list



6. To run the script, select the script in the User Scripts list and select **Run**.
7. The Instrument Output box displays any error messages and output from the script.

Script management options

Existing scripts are listed in the User Scripts box on the left side of TSB Embedded. Script management options are:

- **Run a script:** Select the name of the script and then select **Run**.
- **Delete a script:** Delete the name of the script and select **Delete**. The script is deleted from the User Scripts list and from the nonvolatile memory of the instrument.
- **Stop operation of a script:** Select **Abort Script**.
- **Export the selected script to the computer:** Select **Export to PC**. Choose the directory in which to save the script and select **Save**. Scripts are saved to a file with the extension `tsp`. TSP files are native to Test Script Builder or TSB Embedded, but they can be opened and edited in any text editor.
- **Import scripts from the computer:** Select **Import from PC**. Select the directory that contains the file. You can only import files with the extension `tsp`.
- **Clear the name box and the box that contains the script:** Select **Clear**.
- **View the contents of a script:** Type the name of a script in the TSP Script box and select **View Script**.

Exercise: Use the command line to close and open channels

Use the command line interface to perform close and open operations on channels:

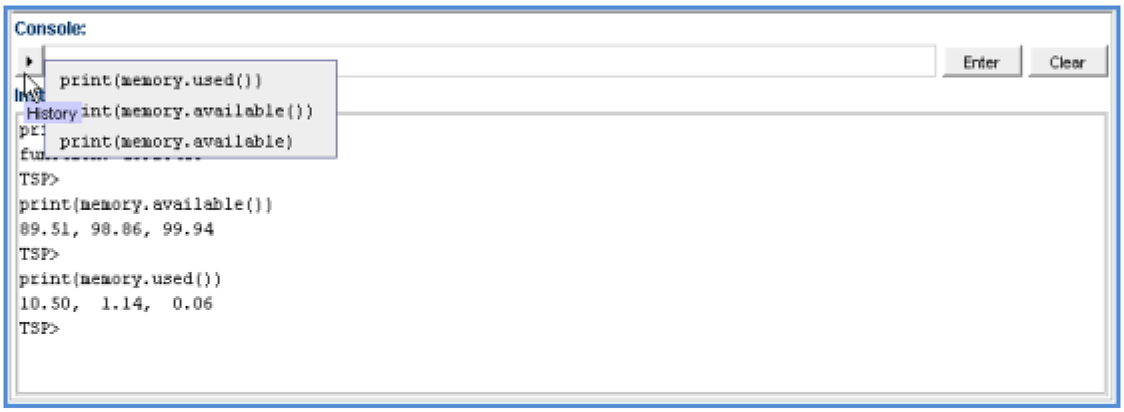
1. Enter `channel.close("1B02")` in the Console box to close channel 1B02.
2. Select **Enter** to send the command.

3. To open all channels, enter `channel.open("allslots")` in the Console box.
4. Select **Enter** to send the command.

Figure 64: Enter command in Console box

5. To resend a command, select the arrow on the left side of the Console box.

Figure 65: Web interface console



Admin page

Through the Admin page, you can change the instrument password and the instrument time.

Change the password

To change the password for the web interface:

1. In the web interface, from the left navigation, select **Admin**. A login page is displayed.
2. Enter the existing password in the Password box. The default is `admin`.
3. Click **Login**. The Administration page is displayed.
4. In the Current Password box, enter the existing password.
5. In the New Password box, enter the new password.
6. In the Confirm New Password box, enter the new password again.
7. Select **Submit**. The new password takes effect immediately.

Change the instrument date and time

To change the date and time of the instrument:

1. In the web interface, from the left navigation, select **Admin**. A login page is displayed.
2. Enter the password in the Password box. The default is `admin`.
3. Click **Login**. The Administration page is displayed.
4. Enter the Year.
5. Select the Month, Day, Hour, Minutes and Seconds from the lists.
6. Select **Submit**. The new time and date information takes effect immediately.

Unit page

You can use Create Config Script to save the setup of the instrument as a script.

To create the script:

1. Select **Create Config Script**. The Create Config Script dialog is displayed.
2. To create a script that runs automatically when the instrument is powered on, select **Auto-execute on powerup**. This overwrites the existing autoexec script.
3. To create a script with a new name, select **Name** and enter the name.
4. Select **OK**.

The other options on the Unit page include:

Reset: Resets all instruments in the TSP-enabled system. This is only available if the instrument is the master. Note that this does not reset the digital I/O lines shown in the Digital I/O Lines area of this page.

Open All: Opens all relays on all slots.

Upgrade Firmware: Use to select a firmware upgrade file to download to the instrument and begin the upgrade process.

Channel Connect Rule: Selects the channel connect rule. See Connection methods for close operations (on page 2-3) for detail.

Digital I/O Lines: Use to configure the 14 digital I/O lines of the instrument. Values can be read or written to the ports, or each individual bit can be toggled. You can set `Write Protect` individually for any I/O line.

Generate Report: Generates an instrument report you can use to:

- Review card or instrument information, including a basic description, the firmware revision, and the serial number.
- Review configuration information, including card configuration and number of poles.
- Review the number of closures for each channel on the selected slots. The counts reported for the following cards indicate the number of closures since the last power cycle of the card:
  - 7072
  - 7072-HV
  - 7173-50
  - 7174A
- The counts are the closures that have occurred over the lifetime of the card for all other cards.

To print the report, select **Print**.

To clear the report information from the screen, select **Clear**.

LXI page

The Model 707B or 708B is an LXI Class C instrument. The LXI page is a read-only page that displays the LXI information about the instrument.

IP Config

The IP Config allows you to review and change the LAN connection information.

See Configure channels from the web interface (on page 4-45) for more information.

Log page

The event log records all LAN[0-7] triggers generated and received and can be viewed over any command interface, including the web interface. The following figure shows the view of the LAN[0-7] event log from the embedded web interface.

Up to 32 LAN[0-7] events are logged and shown on this page. The event log is circular and rolls over after 32 events are captured. The LAN[0-7] events correspond to the lan.trigger[1-8] subsystem.

Figure 66: Event log

The timestamp, event identifier, the IP address, and the domain name identify the incoming and outgoing LXI trigger packets. The following table provides detailed descriptions for the columns in the event log.

Event log descriptions

Column title	**Description**	**Example**
Receive Time	Displays the date and time when the LAN trigger occurred in UTC, 24-hour time.	06:56:28.000 8 May 2008

EventID	Identifies the `lan.trigger[N]` that generates an event.	`LAN0 = lan.trigger[1]`
		`LAN1 = lan.trigger[2]`
		`LAN2 = lan.trigger[3]`
		`LAN3 = lan.trigger[4]`
		`LAN4 = lan.trigger[5]`
		`LAN5 = lan.trigger[6]`
		`LAN6 = lan.trigger[7]`
		`LAN7 = lan.trigger[8]`

From	Displays the IP address for the device that generates the LAN trigger.	localhost 192.168.5.20
System Timestamp	A timestamp that identifies the time the event occurred. The timestamp uses the following: ■ PTP timestamp ■ Seconds ■ Fractional seconds The Model 707B or 708B does not support the IEEE-1588 standard; the values in this field are always 0 (zero).	

Event log descriptions

Column title	Description	Example
HWDetect	Identifies a valid LXI trigger packet	LXI
Sequence	Each instrument maintains independent sequence counters: ▪ One for each combination of UDP multicast network interface and UDP multicast destination port. ▪ One for each TCP connection.	
Domain	Displays the LXI domain number (the default value is 0 (zero)).	0 1523
Flags	Contain data about the LXI trigger packet.	Values: 1 - Error 2 - Retransmission 4- Hardware 8 - Acknowledgments 16 - Stateless bit
Data	The Model 707B or 708B does not support the IEEE-1588 standard; the values in this field are always 0 (zero)	

Send raw commands to an instrument

The following sections show you how to use VISA-C and VISA-COM to send raw instrument commands without using the instrument drivers.

VISA-C sample code

The following is a C/C++ console application that returns the instrument identification string using VISA-C. You need to include `visa.h` and link with the `visa32.lib` file.

```
#include "stdafx.h"
#include <visa.h>

#define checkErr(fCall)      if (error = (fCall), (error = (error <
    0) ? error : VI_SUCCESS)) \
                                 {goto Error;}  else error = error

int _tmain(int argc, _TCHAR* argv[])
{
      ViSession defaultRM, vi;
      char buf [256] = {0};
      ViStatus  error = VI_SUCCESS;

      /* Open session to GPIB device at address 22. */
      checkErr(viOpenDefaultRM(&defaultRM));
      checkErr(viOpen(defaultRM, "GPIB0::14::INSTR", VI_NULL,VI_NULL, &vi));

      /* Initialize device. */
      checkErr(viPrintf(vi, "*RST\n"));

      /* Send an *IDN? string to the device. */
      checkErr(viPrintf(vi, "*IDN?\n"));
      ViUInt16 status = 0;
      do
      {
      checkErr(viReadSTB(vi, &status));
      printf("ReadSTB = %X\n", status);
      } while(status == 0);

      /* Read results. */
      checkErr(viScanf(vi, "%t", &buf));
      /* Print results. */
      printf ("Instrument identification string: %s\n", buf);

      /* Close session. */
      checkErr(viClose(vi));
      checkErr(viClose(defaultRM));

Error:

      if(error < VI_SUCCESS)
         printf("Visa Error Code: %X\n", error);
      printf("\nDone - Press Enter to Exit");
   getchar();

      return 0;
}
```

VISA-COM sample code

This example gets the instrument identification string using VISA-COM in C#.

To do this, you need to add a reference to the VISA-COM interop DLL, which is usually located at `C:\Program Files\IVI Foundation\VISA\VisaCom\Primary Interop Assemblies\Ivi.Visa.Interop.dll`.

```
using Ivi.Visa.Interop;

namespace WindowsApplication1
{

      public class IdnSample: System.Windows.Forms.Form
      {
private Ivi.Visa.Interop.FormattedIO488 ioDmm;
      //
   }
   }
      private void IdnSample_Load(object sender, System.EventArgs e)
      {
         ioDmm = new FormattedIO488Class();

         SetAccessForClosed();
      }

      private void btnInitIO_Click(object sender, System.EventArgs e)
      {

       try
          {
         ResourceManager grm = new ResourceManager();
               ioDmm.IO = (IMessage)grm.Open("GPIB::16::INSTR",
                  AccessMode.NO_LOCK, 2000, "");
               ioDmm.IO.TerminationCharacterEnabled = true;

          }
          catch (SystemException ex)
          {
               MessageBox.Show("Open failed on " + this.txtAddress.Text + " " +
   ex.Source + "  " + ex.Message, "IdnSample", MessageBoxButtons.OK,
   MessageBoxIcon.Error);
               ioDmm.IO = null;
          }
      }
```

Section 5

Introduction to TSP operation

In this section:

Introduction to TSP operation

Instruments that are enabled for Test Script Processor (TSP™) operate like conventional instruments by responding to a sequence of commands sent by the controller. You can send individual commands to the TSP-enabled instrument the same way you do when using any other instrument.

Unlike conventional instruments, TSP-enabled instruments can execute automated test sequences independently, without an external controller. You can load a series of TSP commands into the instrument. You can store these commands as a script that can be run later by sending a single command message to the instrument.

You do not have to choose between using conventional control or script control. You can combine these forms of instrument control in the way that works best for your test application.

Control the instrument by sending individual command messages

You can send a message that contains remote commands to control an instrument through the communications interface. You can use a test program that resides on a computer (the controller) to sequence the actions of the instrument.

TSP commands can be function-based or attribute-based. Function-based commands are commands that control actions or activities. Attribute-based commands define characteristics of an instrument feature or operation.

Constants represent fixed values.

Functions

Function-based commands control actions or activities. A function-based command performs an immediate action on the instrument.

Each function consists of a function name followed by a set of parentheses ( ). Only include information in the parentheses if the function takes a parameter. If the function takes one or more parameters, they are placed between the parentheses and separated by commas.

Example 1

`beeper.beep(0.5, 2400)`
`delay(0.250)`
`beeper.beep(0.5, 2400)`
	Emit a double beep at 2400 Hz. The sequence is 0.5 s on, 0.25 s off, 0.5 s on.

Example 2

You can use the results of a function-based command directly or assign the results to variables for later access. The following code defines `x` and prints it.

`x = math.abs(-100)` `print(x)`	Output: `100`

Function return values

Several of the channel functions return a value for specified channels and channel patterns.

The return value for these functions is a string containing a list of comma-delimited return items. The *channelList* argument of the remote command determines the number and order of these returned items.

When the *channelList* parameter for these functions is `"slotX"`, the response first lists the channels starting from lowest to highest in numeric order.

When the *channelList* parameter for these functions is `"allslots"`, the response starts with slot 1 and continues to slot 6 for the Model 707B. The Model 708B has only one slot, so `"allslots"` is the same as `"slot1"` for this model. Each slot is processed completely before going to the next. Therefore, all slot 1 channels are listed before slot 2 channels.

When the response is numeric, but in string format, use the `tonumber()` function to convert the string to a number. For example, sending these commands:

```
x = tonumber("1403")
print(x)
```

Results in:

```
1.403000000e+03
```

When the response is a comma-delimited string, the individual return items can be identified by iterating through the list using the comma delimiters. For example, the following Lua code starts at the beginning of a string and breaks the string into individual items at each comma. The `tonumber()` function is used on each item to determine if it is a number or not. In either case, the value is printed.

```
index1 = 1
index2 = 1
text = "123,abc,hello,4.56"
endIndex = string.len(text)
while index2 ~= endIndex do
   index2 = string.find(text, ",", index1)
   if not index2 then
      index2 = endIndex
   end

   subString = string.sub(text, index1, index2 - 1)
   if not number(subString) then
      print(subString)
   else
      print(tonumber(subString))
   end
   index1 = index2 + 1
end
```

Attributes

Attribute-based commands are commands that set the characteristics of an instrument feature or operation. For example, a characteristic of TSP-enabled instruments is the model number (`localnode.model`).

Attributes can be read-only, read-write, or write-only. They can be used as a parameter of a function or assigned to another variable.

To set the characteristics, attribute-based commands define a value. For many attributes, the value is in the form of a number or a predefined constant.

Example 1: Set an attribute using a number

`beeper.enable = 0`
This attribute controls the beeps that occur when front-panel controls are selected. Setting this attribute to `0` turns off the beeper.

Example 2: Set an attribute using a constant

`format.data = format.REAL64`
Using the constant `REAL64` sets the print format to double-precision floating-point format.

To read an attribute, you can use the attribute as the parameter of a function or assign it to another variable.

Example 3: Read an attribute using a function

`print(format.data)`
Reads the data format by passing the attribute to the print function. If the data format is set to 3, the output is: `3.0000000e+00` This shows that the data format is set to double precision floating point.

Example 4: Read an attribute using a variable

`fd = format.data`
This reads the data format by assigning the attribute to a variable named `fd`.

Queries

Instruments enabled for Test Script Processor (TSP™) do not have inherent query commands. Like other scripting environments, the `print()` and `printnumber()` commands generate output in the form of response messages. Each `print()` command creates one response message.

Example

```
x = 10
print(x)
```

Example of an output response message:

```
10
```

Your output may be different if you set your ASCII precision setting to a different value.

Data retrieval commands

You can send data retrieval commands that return a comma-delimited string. For example, `print(memory.available())` returns the amount of memory that is available in the instrument.

The comma-delimited string that is returned starts with the lowest channel and goes to the highest channel on Slot 1. It then lists each subsequent slot until the highest slot is reached.

Information on scripting and programming

If you need information about using scripts with your TSP-enabled instrument, see Fundamentals of scripting for TSP (on page 6-1).

If you need information about using the Lua programming language with the instrument, see Fundamentals of programming for TSP (on page 6-13).

About TSP commands

This section contains an overview of the TSP commands for the instrument. The commands are organized into groups, with a brief description of each group. Each section contains links to the detailed descriptions for each command in the TSP command reference section of this documentation.

Beeper control

The beeper commands allow you to sound, enable, or disable the instrument beeper.

beeper.beep() (on page 7-7)

beeper.enable (on page 7-8)

Bit manipulation and logic operations

The bit functions perform bitwise logic operations on two given numbers, and bit operations on one given number. Logic and bit operations truncate the fractional part of given numbers to make them integers.

Logic operations

The `bit.bitand()`, `bit.bitor()`, and `bit.bitxor()` functions in this group perform bitwise logic operations on two numbers. The Test Script Processor (TSP™) scripting engine performs the indicated logic operation on the binary equivalents of the two integers. This bitwise logic operation is performed on all corresponding bits of the two numbers. The result of a logic operation is returned as an integer.

Bit operations

The rest of the functions in this group are used for operations on the bits of a given number. You can use these functions to:

- Clear a bit
- Toggle a bit
- Test a bit
- Set a bit or bit field
- Retrieve the weighted value of a bit or field value

All these functions use an index parameter to specify the bit position of the given number. The least significant bit of a given number has an index of 1, and the most significant bit has an index of 32.

NOTE

The Test Script Processor (TSP) scripting engine stores all numbers internally as IEEE Std 754 double-precision floating-point values. The logical operations work on 32-bit integers. Any fractional bits are truncated. For numbers larger than 4294967295, only the lower 32 bits are used.

Channel functions and attributes

Channel functions and attributes allow you to adjust, select, open, and close channels. You can also set common channel attributes and set up channel patterns.

The channel functions and attributes are:

channel.reset() (on page 7-42)
channel.setdelay() (on page 7-43)
channel.setforbidden() (on page 7-44)
channel.setlabel() (on page 7-45)
channel.setlabelcolumn() (on page 7-46)
channel.setlabelrow() (on page 7-48)

Data queue

Use the data queue commands to:

- Share data between test scripts running in parallel
- Access data from a remote group or a local node on a TSP-Link™ network at any time

The data queue in the Test Script Processor (TSP™) scripting engine is first-in, first-out (FIFO).

You can access data from the data queue even if a remote group or a node has overlapped operations in process.

dataqueue.add() (on page 7-50)
dataqueue.CAPACITY (on page 7-51)
dataqueue.clear() (on page 7-52)
dataqueue.count (on page 7-53)
dataqueue.next() (on page 7-54)

Delay function

This function holds up instrument operation for a specified period of time. It is typically used to soak a device at a specific voltage or current for a specified period.

delay() (on page 7-55)

Digital I/O

The digital I/O port of the instrument can control external circuitry (such as a component handler for binning operations).

The I/O port has 14 lines. Each line can be at TTL logic state 1 (high) or 0 (low). See the pinout diagram in Digital I/O port (on page 3-9) for additional information.

There are commands to read and write to each individual bit, and commands to read and write to the entire port.

Display

Error queue

When errors and events occur, the error and status messages are placed in the error queue. Use the error queue commands to request error and status message information.

Event log functions and attributes

The functions and attributes in this group control the event log.

eventlog.all() (on page 7-89)
eventlog.clear() (on page 7-90)
eventlog.count (on page 7-90)
eventlog.enable (on page 7-91)
eventlog.next() (on page 7-92)
eventlog.overwritemethod (on page 7-93)

Exit function

This function terminates the script.

exit() (on page 7-94)

Queries and response messages

You can use the `print()` and `printnumber()` functions to query the instrument and generate response messages. The format attributes control how the data is formatted for the print functions used.

The `localnode` commands determine if generated errors are automatically sent and if prompts are generated.

format.asciiprecision (on page 7-94)
format.byteorder (on page 7-95)
format.data (on page 7-96)
localnode.prompts (on page 7-129)
localnode.prompts4882 (on page 7-130)
localnode.showerrors (on page 7-133)
print() (on page 7-140)
printnumber() (on page 7-141)

GPIB

These commands store the GPIB address and indicate whether GPIB communications is enabled.

gpib.address (on page 7-98)

LAN and LXI

The LAN commands have options that allow you to review and configure network settings.

The `lan.config.*` commands allow you to configure LAN settings over the remote interface.

NOTE

You must send `lan.applysettings()` for the configuration settings to take effect.

The `lan.status.*` commands help you determine the status of the LAN.

The `lan.trigger[N].*` commands allow you to set up and assert trigger events that are sent over the LAN.

Other LAN commands allow you to reset the LAN, restore defaults, check LXI domain information, and enable or disable the Nagle algorithm.

Local node

These commands allow you to set and read from the local node.

Get and set attribute functions

Use these functions to create functions to get and set attribute values.

Memory functions

Check the amount of memory that is available or used in the instrument.

Operation complete function

This function sets the operation complete status bit when all overlapped commands are completed.

Print functions

Output data from the instrument.

print() (on page 7-140)
printnumber() (on page 7-141)

Reset

Resets settings to their default settings.

digio.trigger[N].reset() (on page 7-62)
lan.reset() (on page 7-107)
localnode.reset() (on page 7-131)
reset() (on page 7-142)
timer.reset() (on page 7-207)
trigger.blender[N].reset() (on page 7-210)
trigger.timer[N].reset() (on page 7-219)
tsplink.trigger[N].reset() (on page 7-233)

Scan

The scan functions and attributes allow you to set up scanning over the remote interface.

scan.abort() (on page 7-143)
scan.add() (on page 7-144)
scan.addimagestep() (on page 7-145)
scan.background() (on page 7-146)
scan.bypass (on page 7-147)
scan.create() (on page 7-148)
scan.execute() (on page 7-149)
scan.list() (on page 7-149)
scan.mode (on page 7-151)
scan.reset() (on page 7-152)
scan.scancount (on page 7-153)
scan.state() (on page 7-154)
scan.stepcount (on page 7-155)
scan.trigger.arm.clear() (on page 7-155)
scan.trigger.arm.set() (on page 7-156)
scan.trigger.arm.stimulus (on page 7-156)
scan.trigger.channel.clear() (on page 7-157)
scan.trigger.channel.set() (on page 7-158)
scan.trigger.channel.stimulus (on page 7-158)
scan.trigger.clear() (on page 7-160)

Scripting

Scripting helps you combine commands into a block of code that the instrument can run. Scripts help you communicate with the instrument efficiently. These commands describe how to create, load, modify, run, and exit scripts.

For detail on using scripts, see Fundamentals of scripting for TSP (on page 6-1).

Slot

The slot attributes configure and read the settings of the cards in the slots. You can also set up pseudocards.

Status model functions

The status model is a set of status registers and queues. You can use the following commands to manipulate and monitor these registers and queues to view and control various instrument events.

status.condition (on page 7-175)
status.node_enable (on page 7-177)
status.node_event (on page 7-179)
status.operation.* (on page 7-181)
status.operation.user.* (on page 7-183)
status.questionable.* (on page 7-186)
status.request_enable (on page 7-188)
status.request_event (on page 7-190)
status.reset() (on page 7-192)
status.standard.* (on page 7-192)
status.system.* (on page 7-195)
status.system2.* (on page 7-197)
status.system3.* (on page 7-199)
status.system4.* (on page 7-201)
status.system5.* (on page 7-203)

Timer functions

Use the functions in this group to control the timer. The timer can be used to measure the time it takes to perform various operations. Use the `timer.reset()` function at the beginning of an operation to reset the timer to zero, and then use the `timer.measure.t()` at the end of the operation to measure the elapsed time.

timer.measure.t() (on page 7-206)
timer.reset() (on page 7-207)

Trigger functions and attributes

Use the trigger functions and attributes to control specific trigger objects.

trigger.blender[N].clear() (on page 7-207)
trigger.blender[N].EVENT_ID (on page 7-208)
trigger.blender[N].orenable (on page 7-208)
trigger.blender[N].overrun (on page 7-209)
trigger.blender[N].reset() (on page 7-210)
trigger.blender[N].stimulus[M] (on page 7-210)
trigger.blender[N].wait() (on page 7-212)
trigger.clear() (on page 7-213)
trigger.EVENT_ID (on page 7-213)
trigger.timer[N].clear() (on page 7-214)

TSP-Link

These functions and attributes allow you to set up and work with a system that is connected by a TSP-Link™ network.

TSP-Net

The TSP-Net module provides a simple socket-like programming interface to instruments that are enabled for Test Script Processor (TSP™).

tspnet.clear() (on page 7-238)
tspnet.connect() (on page 7-239)
tspnet.disconnect() (on page 7-240)
tspnet.execute() (on page 7-241)
tspnet.idn() (on page 7-242)
tspnet.read() (on page 7-243)
tspnet.readavailable() (on page 7-244)
tspnet.reset() (on page 7-244)
tspnet.termination() (on page 7-245)
tspnet.timeout (on page 7-246)
tspnet.tsp.abort() (on page 7-246)
tspnet.tsp.abortonconnect (on page 7-247)
tspnet.tsp.rbtablecopy() (on page 7-248)
tspnet.tsp.runscript() (on page 7-249)
tspnet.write() (on page 7-250)

User-defined strings

Use the functions in this group to store and retrieve user-defined strings in nonvolatile memory. These strings are stored as key-value pairs. Key-value pairs are associated arrays of data items, where the key is used to index into the array. The key is a unique identifier such as a part number or identification string. The value is a data item or a pointer to where that data item is stored.

You can use the `userstring` functions to store custom, instrument-specific information in the instrument, such as department number, asset number, or manufacturing plant location.

userstring.add() (on page 7-250)
userstring.catalog() (on page 7-251)
userstring.delete() (on page 7-252)
userstring.get() (on page 7-253)

Waitcomplete function

Allows you to send a command to wait for all overlapped operations in a group to complete.

waitcomplete() (on page 7-253)

Section 6

Instrument programming

In this section:

Fundamentals of scripting for TSP

NOTE

Though it can improve your process to use scripts, you do not have to create scripts to use the instrument. Most of the examples in the documentation can be run by sending individual command messages. The following few sections of the documentation describe scripting and programming features of the instrument. You only need to review this information if you are using scripting and programming.

Scripting helps you combine commands into a block of code that the instrument can run. Scripts help you communicate with the instrument more efficiently.

Scripts offer several advantages compared to sending individual commands from the host controller (computer):

- Scripts are easier to save, refine, and implement than individual commands.
- The instrument performs more quickly and efficiently when it processes scripts than it does when it processes individual commands.
- You can incorporate features such as looping and branching into scripts.
- Scripts allow the controller to perform other tasks while the instrument is running a script, enabling some parallel operation.
- Scripts eliminate repeated data transfer times from the controller.

In the instrument, the Test Script Processor (TSP™) scripting engine processes and runs scripts.

This section describes how to create, load, modify, and run scripts.

What is a script?

A script is a collection of instrument control commands and programming statements. Scripts that you create are referred to as **user scripts**.

Your scripts can be interactive. Interactive scripts display messages on the front panel of the instrument that prompt the operator to enter parameters.

Runtime and nonvolatile memory storage of scripts

Scripts are loaded into the runtime environment of the instrument. From there, they can be stored in nonvolatile memory in the instrument.

The runtime environment is a collection of global variables, which include scripts, that the user has defined. A global variable can be used to store a value while the instrument is turned on. When you create a script, the instrument creates a global variable with the same name so that you can reference the script more conveniently. After scripts are loaded into the runtime environment, you can run and manage them from the front panel of the instrument or from a computer. Information in the runtime environment is lost when the instrument is turned off.

Nonvolatile memory is where information is stored even when the instrument is turned off. Save scripts to nonvolatile memory to save them even if the power is cycled. The scripts that are in nonvolatile memory are loaded into the runtime environment when the instrument is turned on.

Scripts are placed in the runtime environment when:

- The instrument is turned on. All scripts that are saved to nonvolatile memory are copied to the runtime environment when the instrument is turned on.
- Loaded over a remote command interface.

For detail on the amount of memory available in the runtime environment, see Memory considerations for the runtime environment (on page 6-45).

NOTE

If you make changes to a script in the runtime environment, the changes are lost when the instrument is turned off. To save the changes, you must save them to nonvolatile memory. See Working with scripts in nonvolatile memory (on page 6-10).

What can be included in scripts?

Scripts can include combinations of Test Script Processor (TSP^{TM}) commands and Lua code. TSP commands instruct the instrument to do one thing and are described in the command reference (see TSP commands (on page 7-7)). Lua is a scripting language that is described in Fundamentals of programming for TSP (on page 6-13).

Commands that cannot be used in scripts

Though the instrument accepts the following commands, you cannot use these commands in scripts.

Commands that cannot be used in scripts

General commands	IEEE Std 488.2 common commands	
abort endflash endscript flash loadscript loadandrunscript password	*CLS *ESE *ESE? *ESR? *IDN? *OPC *OPC?	*RST *SRE *SRE? *STB? *TRG *TST? *WAI

Manage scripts

This section describes how to create scripts by sending commands over the remote interface.

Tools for managing scripts

To manage scripts, you can send messages to the instrument, use your own development tool or program, use Keithley Test Script Builder (TSB) software, or use TSB Embedded on the web interface of the instrument.

TSB software is a programming tool that you can download from the Product Support and Downloads web page (tek.com/support/product-support). You can use it to create, modify, debug, and store Test Script Processor (TSP^{TM}) scripting engine scripts. For more information about using the TSB software, see Using Test Script Builder (TSB) (on page 6-33).

TSB Embedded is a tool with a reduced set of features than the complete Keithley TSB software. TSB Embedded has both script-building functionality and console functionality (single-line commands). It is accessed from a web browser. Refer to Working with TSB Embedded for additional information.

NOTE

If you are using TSB or TSB Embedded to create scripts, you do not need to use the commands `loadscript` or `loadandrunscript` and `endscript`.

Create and load a script

You create scripts by loading them into the runtime environment of the instrument. You can load a script as a named script or as the anonymous script.

Once a script is loaded into the instrument, you can execute it remotely or from the front panel.

Anonymous scripts

If a script is created with the `loadscript` or `loadandrunscript` command with no name defined, it is called the anonymous script. There can only be one anonymous script in the runtime environment. If another anonymous script is loaded into the runtime environment, it replaces the existing anonymous script.

Named scripts

A named script is a script with a unique name. You can have as many named scripts as needed in the instrument (within the limits of the memory available to the runtime environment). When a named script is loaded into the runtime environment with the `loadscript` or `loadandrunscript` commands, a global variable with the same name is created to reference the script.

Key points regarding named scripts:

- If you load a new script with the same name as an existing script, the existing script becomes an unnamed script, which in effect removes the existing script if there are no variables that reference it.
- Sending revised scripts with different names does not remove previously loaded scripts.
- Named scripts can be saved to internal nonvolatile memory. Saving a named script to nonvolatile memory allows the instrument to be turned off without losing the script. See Working with scripts in nonvolatile memory (on page 6-10).

Load a script by sending commands over the remote interface

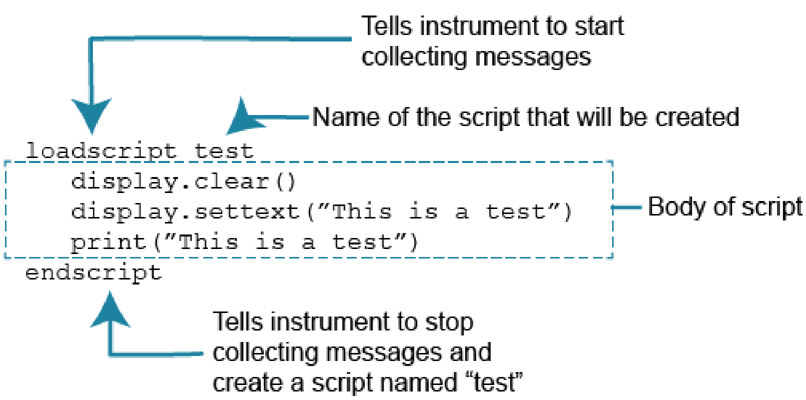
To load a script over the remote interface, you can use the `loadscript`, `loadandrunscript`, and `endscript` commands.

The `loadscript` and `loadandrunscript` commands start the collection of messages that make up the script. When the instrument receives either of these commands, it starts collecting all subsequent messages. Without these commands, the instrument runs them immediately as individual commands.

The `endscript` command tells the instrument to compile the collection of messages. It compiles the messages into one group of commands. This group of commands is loaded into the runtime environment.

The following figure shows an example of how to load a script named `test`. The first command tells the instrument to start collecting the messages for the script named `test`. The last command marks the end of the script. When this script is run, the message `This is a test` is displayed on the instrument and sent to the computer.

Figure 67: Loadscript and endscript example



To load a named script by sending commands:

1. Send the command `loadscript` *`scriptName`*, where *`scriptName`* is the name of the script. The name must be a legal Lua variable name.
2. Send the commands that need to be included in the script.
3. Send the command `endscript`.
4. You can now run the script. See Run scripts (on page 6-6).

NOTE

To run the script immediately, use `loadandrunscript` *`scriptName`* instead of `loadscript`.

Create a script using TSB Embedded

NOTE

If you are using TSB Embedded to create scripts, you do not need to use the commands `loadscript` or `loadandrunscript` and `endscript`.

You can create a script from the instrument web interface with TSB Embedded. When you save the script, it is loaded into the runtime environment and saved in the nonvolatile memory of the instrument.

To create a script using TSB Embedded:

1. In the **TSP Script** box, enter a name for the script.
2. In the input area, enter the sequence of commands to be included in the script.
3. Select **Save Script**. The name is added to the User Scripts list on the left.

Run scripts

This section describes how to run the anonymous and named scripts.

On the front panel, items are available through the USER menu if you explicitly add them to the menu. The items the menu selections represent can be scripts, function calls, or instrument commands. Items in the menus are referred to as scripts in this section.

The SCRIPTS menu lists the names of scripts in nonvolatile memory or scripts that have been added to the runtime environment. The anonymous script also appears in this menu.

NOTE

If the instrument is in local control when the script is started, it switches to remote control (`REM` is displayed) while the script is running. The instrument is returned to local control when the script completes. If you press the front-panel **EXIT (LOCAL)** key while the script is running, the script is stopped.

Run the anonymous script

The anonymous script can be run many times without reloading it. It remains in the runtime environment until a new anonymous script is created or until the instrument is turned off.

To run the anonymous script, use any one of these commands:

- `run()`
- `script.run()`
- `script.anonymous()`
- `script.anonymous.run()`

Run a named script

You can run any named script that is in the runtime environment using one of the following commands:

- *`scriptVar`*`()`
- *`scriptVar`*`.run()`

Where *`scriptVar`* is the user-defined name of the script.

When a script is named, it can be accessed using the global variable *`scriptVar`*.

To run a named script from TSB Embedded, select the script from the User Scripts list and select **Run**.

Example: Run a named script

`test3()`	If the script `test3` is loaded into the runtime environment, the instrument executes `test3`.

Run a user script from the instrument front panel

From the front panel, you can load and run a script that was previously added to the USER menu.

To run the code from the front panel and add it to the USER menu:

1. Select the **LOAD** key.
2. Select **USER**.
3. Select the script from list and press the **ENTER** key. The script is loaded into the runtime environment.
4. Press the **RUN** key to execute.

To run a script directly without adding it to the USER menu:

1. Select the **LOAD** key.
2. Select **SCRIPTS** and select the **ENTER** key. There may be a short pause before a menu is displayed that represents the scripts in the instrument.
3. Select the script from the list and select the **ENTER** key. The script is now loaded for front-panel execution.
4. Press the **RUN** key to execute.

Scripts that run automatically

You can set up scripts to run automatically when you power on the instrument. To do this, either set the `autorun` attribute for the script to `yes` (see Autorun scripts (on page 6-8)) or create a script with the script name `autoexec` (see Autoexec script (on page 6-9)).

Autorun scripts

Autorun scripts run automatically when the instrument is turned on. You can set any number of scripts to autorun. The run order for autorun scripts is arbitrary, so make sure the run order is not important.

You can set a script to run automatically by setting the `.autorun` attribute of the script to `"yes"` and then saving the script. The format is:

```
scriptVar.autorun = "yes"
scriptVar.save()
```

Where *`scriptVar`* is the user-defined name of the script.

To disable autorun, set the `autorun` attribute of the script to `no` and then save the script.

NOTE

The *scriptVar*.save() command saves the script to nonvolatile memory, which makes the change persistent through a power cycle. Refer to Save a user script to nonvolatile memory (on page 6-10) for more detail.

Example: Set a script to run automatically

```
test5.autorun = "yes"
test5.save()
```

Assume a script named test5 is in the runtime environment.
The next time the instrument is turned on, test5 script automatically loads and runs.

Autoexec script

The autoexec script runs automatically when the instrument is turned on. It runs after all the scripts have loaded and any scripts defined as autorun have run.

To create a script that executes automatically, create and load a new script and name it autoexec. See Create and load a script (on page 6-4).

NOTE

You must save the autoexec script to nonvolatile memory if you want to use it after instrument power has been turned off and then turned on again. See Save a user script to nonvolatile memory (on page 6-10) for more detail.

Example: Create an autoexec script with the loadscript command

```
loadscript autoexec
display.clear()
display.settext("Hello from autoexec")
endscript
autoexec.save()
```

Creates the script autoexec.
Saves the autoexec script to nonvolatile memory. The next time the instrument is turned on, Hello from autoexec is displayed.

Example: Create an autoexec script using TSB Embedded

```
display.clear()
display.settext("Hello from autoexec")
```

In the TSP Script box, enter autoexec.
Enter the code in the entry box.
Select **Save Script**.
Creates a new script that clears the display when the instrument is turned on and displays Hello from autoexec.

Work with scripts in nonvolatile memory

This section of the manual has primarily described working with scripts in the runtime environment. You can also work with scripts in nonvolatile memory.

The runtime environment and nonvolatile memory are separate storage areas in the instrument. The information in the runtime environment is lost when the instrument is turned off. The nonvolatile memory remains intact when the instrument is turned off. When the instrument is turned on, information in nonvolatile memory is loaded into the runtime environment.

Save a user script

You can save scripts to nonvolatile memory using commands or TSB Embedded.

Only named scripts can be saved to nonvolatile memory. The anonymous script must be named before it can be saved to nonvolatile memory.

To save a script to nonvolatile memory from a remote interface:

1. Create and load a named script (see Create and load a script (on page 6-4)).
2. Send the command `scriptVar.save()`, where `scriptVar` is the name of the script.

To save a script to nonvolatile memory using TSB Embedded:

Select **Save Script**.

Example: Save a user script to nonvolatile memory

`test1.save()`
Assume a script named `test1` has been loaded. `test1` is saved into nonvolatile memory.

To save a script to an external USB flash drive using a remote interface:

NOTE

When you save a script to a USB flash drive, you do not need to specify a file extension. The extension `.tsp` is automatically added. If you do specify a file extension, it must be `.tsp`. An error occurs if you use any other file extension.

1. Load a script (see Create and load a script (on page 6-4)).
2. Send the command `scriptVar.save("/usb1/filename.tsp")`, where `scriptVar` is the variable referencing the script and `filename.tsp` is the name of the file.

To save a script to an external USB flash drive or other accessible drive using TSB Embedded:

Load the script and select **Export to PC**.

Save the anonymous script as a named script

To save the anonymous script to nonvolatile memory, you must name it first.

To save the anonymous script as a named script:

1. To name the script, send the command `script.anonymous.name = "myTest"` (where *myTest* is the name of the script).
2. Send the `script.anonymous.save()` command to save *myTest* to nonvolatile memory.

Save a script from the instrument front panel

You can save scripts from the runtime environment to nonvolatile memory on the instrument front panel.

NOTE

If you want to save the anonymous script to nonvolatile memory, you must name it first. See Save the anonymous script as a named script (on page 6-11).

To save a script to nonvolatile memory from the front panel:

1. Select the **MENU** key.
2. Select the **SCRIPT** option.
3. Select the **SAVE** option.

   A list of the scripts available to save is displayed. It may take a few seconds to display. The displayed list is from the `script.user.scripts` table in the instrument.
4. Turn the navigation wheel to select the script that you want to save.
5. Select **INTERNAL**. Press the navigation wheel. The script is saved to nonvolatile memory using the name attribute of the script.
6. Press **EXIT (LOCAL)** several times to return to the Main Menu.

Delete user scripts

NOTE

These steps remove a script from nonvolatile memory. To completely remove a script from the instrument, there are additional steps you must take. See Delete user scripts from the instrument (on page 6-44).

To delete a script from nonvolatile memory using a remote interface, send either of the following commands:

- `script.delete("name")`
- `script.user.delete("name")`

Where: *`name`* is the user-defined name of the script.

Example: Delete a user script from nonvolatile memory

`script.delete("test8")`	Delete a user script named `test8` from nonvolatile memory.

To delete a script from nonvolatile memory using TSB Embedded:

1. In TSB Embedded, select the script from the **User Scripts** list.
2. Select **Delete**. There is no confirmation message.

Programming example: Interactive script

An interactive script prompts the operator to input values using the instrument front panel. The following example script uses display messages to prompt the operator to:

- Enter the digital I/O line on which to output a trigger
- Enter the output trigger pulsewidth

After the output trigger occurs, the front display displays a message to the operator.

When an input prompt is displayed, the script waits until the operator inputs the parameter or presses the **ENTER** key.

The example shown here assumes that you are using TSB. If you are using a remote interface, you need to add the `loadscript` and `endscript` commands to the example code. See Load a script by sending commands over the remote interface (on page 6-5) for details.

```
-- Clear the display.
display.clear()

-- Prompt user for digital I/O line on which to output trigger.
myDigioLine = display.menu("Select digio line", "1 2 3 4 5 6 7 8 9")

-- Convert user input to a number.
intMyDigioLine = tonumber(myDigioLine)

-- Prompt user for digital output trigger mode.
myDigioEdge = display.menu("Select digio mode", "Rising Falling")
if myDigioEdge == "Rising" then
   edgeMode = digio.TRIG_RISING
else
   edgeMode = digio.TRIG_FALLING
end

-- Prompt user for output trigger pulsewidth.
myPulseWidth = display.prompt(
   "000.0", "us", "Enter trigger pulsewidth", 10, 10, 100)

-- Scale the entered pulsewidth.
myPulseWidth = myPulseWidth * 1e-6

-- Generate the pulse.
digio.trigger[intMyDigioLine].mode = edgeMode
digio.trigger[intMyDigioLine].pulsewidth = myPulseWidth
digio.trigger[intMyDigioLine].assert()

-- Alert the user through the display that the
-- output trigger has occurred.
display.setcursor(1, 1)
display.settext("Trigger asserted $Non digital I/O line " .. intMyDigioLine)

-- Wait five seconds and then return to main screen.
delay(5)
display.screen = display.MAIN
```

Fundamentals of programming for TSP

To conduct a test, a computer (controller) is programmed to send sequences of commands to an instrument. The controller orchestrates the actions of the instrumentation. The controller is typically programmed to request measurement results from the instrumentation and make test sequence decisions based on those measurements.

To use the advanced features of the instrument, you can add programming commands to your scripts. Programming commands control script execution and provide tools such as variables, functions, branching, and loop control.

The Test Script Processor (TSP™) scripting engine is a Lua interpreter. In TSP-enabled instruments, the Lua programming language has been extended with control commands that are specific to Keithley instruments.

What is Lua?

Lua is a programming language that can be used with TSP-enabled instruments. Lua is an efficient language with simple syntax that is easy to learn.

Lua is also a scripting language, which means that scripts are compiled and run when they are sent to the instrument. You do not compile them before sending them to the instrument.

Lua basics

This section contains the basics about the Lua programming language to allow you to start adding Lua programming commands to your scripts quickly.

For more information about Lua, see the Lua website (lua.org). Another source of useful information is the Lua users group (lua-users.org), created for and by users of Lua programming language.

Comments

You can start a comment anywhere outside a string by typing a double hyphen (`--`). If the text immediately after `--` is anything other than double left brackets (`[[`), the comment is a short comment, which continues until the end of the line.

If `--` is followed by `[[`, the following characters are a long comment, which continues until double right brackets (`]]`) close the comment. Long comments may continue for several lines and may contain nested `[[ . . . ]]` pairs.

An example of a short comment is:

```
-- Turn off the front-panel display.
```

An example of a long comment is:

```
--[[Display a menu with three menu items. If the second menu item is selected,
the selection is given the value Test2.]]
```

Function and variable name restrictions

You cannot use Lua reserved words and top-level command names for function or variable names.

You cannot use the following Lua reserved words for function or variable names.

Lua reserved words

`and`	`for`	`or`
`break`	`function`	`repeat`
`do`	`if`	`return`
`else`	`in`	`then`
`elseif`	`local`	`true`
`end`	`nil`	`until`
`false`	`not`	`while`

You also cannot use top-level command names as variable names. If you use these names, it results in the loss of use of the commands. For example, if you send the command `digio = 5`, you cannot access the `digio.*` commands until you turn the instrument power off and then turn it on again. These groups include:

Top level command names

`beeper`	`localnode`
`bit`	`memory`
`channel`	`opc`
`dataqueue`	`reset`
`delay`	`scan`
`digio`	`slot`
`display`	`status`
`errorqueue`	`timer`
`eventlog`	`trigger`
`exit`	`tsplink`
`format`	`tspnet`
`gpib`	`userstring`
`lan`	`waitcomplete`

Values and variable types

In Lua, you use variables to store values in the runtime environment for later use.

Lua is a dynamically-typed language; the type of the variable is determined by the value that is assigned to the variable.

Variables in Lua are assumed to be global unless they are explicitly declared to be local. A global variable is accessible by all commands. Global variables do not exist until they have been assigned a value.

Variable types

Variables can be one of the following types.

Variable types and values

Variable type returned	Value	Notes
`"nil"`	not declared	The type of the value `nil`, whose main property is to be different from any other value; usually it represents the absence of a useful value.
`"boolean"`	true or false	Boolean is the type of the values `false` and `true`. In Lua, both `nil` and `false` make a condition `false`; any other value makes it `true`.
`"number"`	number	All numbers are real numbers; there is no distinction between integers and floating-point numbers.
`"string"`	sequence of words or characters	
`"function"`	a block of code	Functions perform a task or compute and return values.
`"table"`	an array	New tables are created with `{ }` braces. For example: `{1, 2, 3.00e0}`
`"userdata"`	variables	Allows arbitrary program data to be stored in Lua variables.
`"thread"`	line of execution	

To determine the type of a variable, you can call the `type()` function, as shown in the following examples.

NOTE

The output you get from these examples may vary depending on the data format that is set.

Example: Nil

`x = nil` `print(x, type(x))`	`nil      nil`

Example: Boolean

`y = false` `print(y, type(y))`	`false      boolean`

Example: String and number

`x = "123"` `print(x, type(x))`	`123      string`
`x = x + 7` `print(x, type(x))`	Adding a number to x forces its type to number. `130     number`

Example: Function

`function add_two(first_value,`
`   second_value)`
`   return first_value + second_value`
`end`
`print(add_two(3, 4), type(add_two))`
	`7      function`

Example: Table

`atable = {1, 2, 3, 4}`
`print(atable, type(atable))`
`print(atable[1])`
`print(atable[4])`
	Defines a table with four numeric elements.
	Note that the *`table`* value (shown here as `a096cd30`) varies.
	
	`table: a096cd30      table`
	`1`
	`4`

Delete a global variable

To delete a global variable, assign `nil` to the global variable. This removes the global variable from the runtime environment.

Functions

With Lua, you can group commands and statements using the `function` keyword. Functions can take zero, one, or multiple parameters, and they return zero, one, or multiple values.

You can use functions to form expressions that calculate and return a value. Functions can also act as statements that execute specific tasks.

Functions are first-class values in Lua. That means that functions can be stored in variables, passed as arguments to other functions, and returned as results. They can also be stored in tables.

When a function is defined, it is stored in the runtime environment. Like all data that is stored in the runtime environment, the function persists until it is removed from the runtime environment, is overwritten, or the instrument is turned off.

Create functions using the function keyword

Functions are created with a message or in Lua code in either of the following forms:

```
function myFunction(parameterX) functionBody end
myFunction = function (parameterX) functionBody end
```

Where:

- *myFunction*: The name of the function.
- *parameterX*: Parameter names. To use multiple parameters, separate the names with commas.
- *functionBody*: The code that is executed when the function is called.

To execute a function, substitute appropriate values for *parameterX* and insert them into a message formatted as:

```
myFunction(valueForParameterX, valueForParameterY)
```

Where *valueForParameterX* and *valueForParameterY* represent the values to be passed to the function call for the given parameters.

NOTE

The output you get from these examples may vary depending on the data format settings of the instrument.

Example 1

`function add_two(first_value,`
`  second_value)`
`  return first_value + second_value`
`end`
`print(add_two(3, 4))`
	Creates a variable named `add_two` that has a variable type of function.
	Output:
	`7`

Example 2

`add_three = function(first_value,`
`  second_value, third_value)`
`  return first_value + second_value +`
`    third_value`
`end`
`print(add_three(3, 4, 5))`
	Creates a variable named `add_three` that has a variable type of function.
	Output:
	`12`

Example 3

`function sum_diff_ratio(first_value,`
`   second_value)`
`   psum = first_value + second_value`
`   pdif = first_value - second_value`
`   prat = first_value / second_value`
`   return psum, pdif, prat`
`end`
`sum, diff, ratio = sum_diff_ratio(2, 3)`
`print(sum)`
`print(diff)`
`print(ratio)`
	Returns multiple parameters (sum, difference, and ratio of the two numbers passed to it).
	Output:
	`5`
	`-1`
	`0.66666666666667`

Create functions using scripts

You can use scripts to define functions. Scripts that define a function are like any other script: They do not cause any action to be performed on the instrument until they are executed. The global variable of the function does not exist until the script that created the function is executed.

A script can consist of one or more functions. Once a script has been run, the computer can call functions that are in the script directly.

NOTE

The following steps use TSB Embedded. You can also use the `loadscript` and `endscript` commands to create the script over the remote interface. See Load a script by sending commands over the remote interface (on page 6-5).

Steps to create a function using a script:

1. In TSB Embedded, enter a name into the TSP Script box. For example, type `MakeMyFunction`.
2. Enter the function as the body of the script. This example concatenates two strings:

```
MyFunction = function (who)
      print ("Hello".. who)
end
```

3. Select **Save Script**.
4. `MakeMyFunction` is now on the instrument in a global variable with the same name as the script (`MakeMyFunction`). However, the function defined in the script does not yet exist because the script has not been executed.

5. Run the script as a function. For this example, send:

   `MakeMyFunction()`

   This instructs the instrument to run the script, which creates the `MyFunction` global variable. This variable is of the type "`function`" (see Variable types (on page 6-16)).

6. Run the new function with a value.

   `MyFunction("world")`

   The response message is:

   `Hello world.`

Group commands using the function keyword

The following script contains instrument commands that display the name of the person that is using the script on the front panel of the instrument. It takes one parameter to represent this name. When this script is run, the function is loaded in memory. Once loaded into memory, you can call the function outside of the script to execute it.

When calling the function, you must specify a string for the *`name`* argument of the function. For example, to set the name to **John**, call the function as follows:

```
myDisplay("John")
```

Example: User script

<table>
<tr><th>User script created in Test Script Builder or TSB Embedded</th><th>User script created in a different program</th></tr>
<tr><td><pre>
function myDisplay(name)
   display.clear()
   display.settext(
      name .. "$N is here!")
end</pre></td><td><pre>loadscript
function myDisplay(name)
   display.clear()
   display.settext(
      name .. " $N is here!")
end
endscript</pre></td></tr>
</table>

Operators

You can compare and manipulate Lua variables and constants using operators.

Arithmetic operators

Operator	Description
+	addition
−	subtraction
*	multiplication
/	division
−	negation (for example, c = −a)
^	exponentiation

Relational operators

Operator	Description
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
~=	not equal
==	equal

Logical operators

The logical operators in Lua are `and`, `or`, and `not`. All logical operators consider both `false` and `nil` as false and anything else as true.

The operator `not` always returns `false` or `true`.

The conjunction operator `and` returns its first argument if the first argument is `false` or `nil`; otherwise, `and` returns its second argument. The disjunction operator `or` returns its first argument if this value is different from `nil` and `false`; otherwise, `or` returns its second argument. Both `and` and `or` use shortcut evaluation, that is, the second operand is evaluated only if necessary.

NOTE

The example output you get may vary depending on the data format settings of the instrument.

Example

`print(10 or errorqueue.next())`	`1.0000000e+01`
`print(nil or "a")`	`a`
`print(nil and 10)`	`nil`
`print(false and errorqueue.next())`	`false`
`print(false and nil)`	`false`
`print(false or nil)`	`nil`
`print(10 and 20)`	`2.0000000e+01`

String concatenation

String operators

Operator	Description
`..`	Concatenates two strings. If either argument is a number, it is coerced to a string (in a reasonable format) before concatenation.

Example: Concatenation

```
print(2 .. 3)
print("Hello " .. "World")
```

Output:

```
23
Hello World
```

Operator precedence

Operator precedence in Lua follows the order below (from higher to lower priority):

- `^` (exponentiation)
- `and`
- `or`

You can use parentheses to change the precedences in an expression. The concatenation ("`..`") and exponentiation ("`^`") operators are right associative. All other binary operators are left associative. The following examples show equivalent expressions.

Equivalent expressions

`reading + offset < testValue/2+0.5`	=	`(reading + offset) < ((testValue/2)+0.5)`
`3+reading^2*4`	=	`3+((reading^2)*4)`
`Rdg < maxRdg and lastRdg <= expectedRdg`	=	`(Rdg < maxRdg) and (lastRdg <= expectedRdg)`
`-reading^2`	=	`-(reading^2)`
`reading^testAdjustment^2`	=	`reading^(testAdjustment^2)`

Conditional branching

Lua uses the `if`, `else`, `elseif`, `then`, and `end` keywords to do conditional branching.

In Lua, `nil` and `false` are `false` and everything else is `true`. Zero (`0`) is `true` in Lua.

The syntax of a conditional block is as follows:

```
if expression then
   block
elseif expression then
   block
else
   block
end
```

Where:

- *`expression`* is Lua code that evaluates to either `true` or `false`
- *`block`* consists of one or more Lua statements

Example: If

`if 0 then`
`   print("Zero is true!")`
`else`
`   print("Zero is false.")`
`end`
	Output:
	`Zero is true!`

Example: Comparison

`x = 1`
`y = 2`
`if x and y then`
`   print("Both x and y are true")`
`end`
	Output:
	`Both x and y are true`

Example: If and else

`x = 2`
`if not x then`
`   print("This is from the if block")`
`else`
`   print("This is from the else block")`
`end`
	Output:
	`This is from the else block`

Example: Else and elseif

```
x = 1
y = 2
if x and y then
   print("'if' expression 2 was not false.")
end

if x or y then
   print("'if' expression 3 was not false.")
end

if not x then
   print("'if' expression 4 was not false.")
else
   print("'if' expression 4 was false.")
end

if x == 10 then
   print("x = 10")
elseif y > 2 then
   print("y > 2")
else
   print("x is not equal to 10, and y is not greater than 2.")
end
```

Output:

```
'if' expression 2 was not false.
'if' expression 3 was not false.
'if' expression 4 was false.
x is not equal to 10, and y is not greater than 2.
```

Loop control

If you need to repeat code execution, you can use the Lua `while`, `repeat`, and `for` control structures. To exit a loop, you can use the `break` keyword.

While loops

To use conditional expressions to determine whether to execute or end a loop, you use `while` loops. These loops are similar to Conditional branching (on page 6-22) statements.

```
while expression do
   block
end
```

Where:

- *`expression`* is Lua code that evaluates to either `true` or `false`
- *`block`* consists of one or more Lua statements

NOTE

The output you get from this example may vary depending on the data format settings of the instrument.

Example: While

`list = {`
`   "One", "Two", "Three", "Four", "Five", "Six"}`
`print("Count list elements on numeric index:")`
`element = 1`
`while list[element] do`
`   print(element, list[element])`
`   element = element + 1`
`end`
	This loop exits when `list[element] = nil`.
	Output:
	`Count list elements on`
	`   numeric index:`
	`1   One`
	`2   Two`
	`3   Three`
	`4   Four`
	`5   Five`
	`6   Six`

Repeat until loops

To repeat a command, you use the `repeat` ... `until` statement. The body of a `repeat` statement always executes at least once. It stops repeating when the conditions of the `until` clause are met.

```
repeat
   block
until expression
```

Where:

- *`block`* consists of one or more Lua statements
- *`expression`* is Lua code that evaluates to either `true` or `false`

NOTE

The output you get from this example may vary depending on the data format settings of the instrument.

Example: Repeat until

```
list = {"One", "Two", "Three", "Four", "Five", "Six"}
print("Count elements in list using repeat:")
element = 1
repeat
   print(element, list[element])
   element = element + 1
until not list[element]
```

Output:

```
Count elements in list
   using repeat:
1  One
2  Two
3  Three
4  Four
5  Five
6  Six
```

For loops

There are two variations of `for` statements supported in Lua: Numeric and generic.

NOTE

In a `for` loop, the loop expressions are evaluated once, before the loop starts.

The output you get from these examples may vary depending on the data format settings of the instrument.

Example: Numeric for

```
list = {"One", "Two", "Three", "Four", "Five", "Six"}
---------- For loop -----------
print("Counting from one to three:")
for element = 1, 3 do
   print(element, list[element])
end
print("Counting from one to four, in steps of two:")
for element = 1, 4, 2 do
   print(element, list[element])
end
```

The numeric `for` loop repeats a block of code while a control variable runs through an arithmetic progression.

Output:

```
Counting from one to three:
1   One
2   Two
3   Three
Counting from one to four, in steps of two:
1   One
3   Three
```

Example: Generic for

```
days = {"Sunday",
   "Monday",     "Tuesday",
   "Wednesday",  "Thursday",
   "Friday",     "Saturday"}

for i, v in ipairs(days) do
   print(days[i], i, v)
end
```

The generic `for` statement works by using functions called iterators. On each iteration, the iterator function is called to produce a new value, stopping when this new value is nil.

Output:

```
Sunday      1    Sunday
Monday      2    Monday
Tuesday     3    Tuesday
Wednesday   4    Wednesday
Thursday    5    Thursday
Friday      6    Friday
Saturday    7    Saturday
```

Break

The `break` statement terminates the execution of a `while`, `repeat`, or `for` loop, skipping to the next statement after the loop. A break ends the innermost enclosing loop.

Return and break statements can only be written as the last statement of a block. If it is necessary to return or break in the middle of a block, an explicit inner block can be used.

NOTE

The output you get from these examples may vary depending on the data format settings of the instrument.

Example: Break with while statement

<table>
<tr><td>

```
local numTable = {5, 4, 3, 2, 1}
local k = table.getn(numTable)
local breakValue = 3
while k > 0 do
   if numTable[k] == breakValue then
      print("Going to break and k = ", k)
      break
   end
   k = k - 1
end
if k == 0 then
   print("Break value not found")
end
```

</td><td>

This example defines a `break` value (`breakValue`) so that the `break` statement is used to exit the `while` loop before the value of `k` reaches `0`.

Output:

`Going to break and k = 3`

</td></tr>
</table>

Example: Break with while statement enclosed by comment delimiters

<table>
<tr><td>

```
local numTable = {5, 4, 3, 2, 1}
local k = table.getn(numTable)
-- local breakValue = 3
while k > 0 do
   if numTable[k] == breakValue then
      print("Going to break and k = ", k)
      break
   end
   k = k - 1
end
if k == 0 then
   print("Break value not found")
end
```

</td><td>

This example defines a break value (`breakValue`), but the break value line is preceded by comment delimiters so that the break value is not assigned and the code reaches the value `0` to exit the `while` loop.

Output:

`Break value not found`

</td></tr>
</table>

Example: Break with infinite loop

<table>
<tr><td>

```
a, b = 0, 1
while true do
   print(a, b)
   a, b = b, a + b
   if a > 500 then
      break
   end
end
```

</td><td>

This example uses a `break` statement that causes the while loop to exit if the value of `a` becomes greater than 500.

Output:

```
0     1
1     1
1     2
2     3
3     5
5     8
8     13
13    21
21    34
34    55
55    89
89    144
144   233
233   377
377   610
```

</td></tr>
</table>

Tables and arrays

Lua makes extensive use of the data type table, which is a flexible array-like data type. Table indices start with 1. Tables can be indexed not only with numbers, but with any value except `nil`. Tables can be heterogeneous, which means that they can contain values of all types except `nil`.

Tables are the sole data structuring mechanism in Lua. They may be used to represent ordinary arrays, symbol tables, sets, records, graphs, trees, and so on. To represent records, Lua uses the field `name` as an index. The language supports this representation by providing `a.name` as an easier way to express `a["name"]`.

NOTE

The output you get from this example may vary depending on the data format settings of the instrument.

Example: Loop array

`atable = {1, 2, 3, 4}`
`i = 1`
`while atable[i] do`
`   print(atable[i])`
`   i = i + 1`
`end`
	Defines a table with four numeric elements.
	Loops through the array and prints each element. The Boolean value of `atable[index]` evaluates to `true` if there is an element at that index. If there is no element at that index, `nil` is returned (`nil` is considered to be `false`).
	Output:
	`1`
	`2`
	`3`
	`4`

Standard libraries

In addition to the standard programming constructs described in this document, Lua includes standard libraries that contain useful functions for string manipulation, mathematics, and related functions. Test Script Processor (TSP™) scripting engine instruments also include instrument control extension libraries, which provide programming interfaces to the instrumentation that can be accessed by the TSP scripting engine. These libraries are automatically loaded when the TSP scripting engine starts and do not need to be managed by the programmer.

The following topics provide information on some of the basic Lua standard libraries. For additional information, see the Lua website (lua.org). The TSP scripting engine uses Lua 5.0.2.

Base library functions

Base library functions

Function	Description
`collectgarbage()` `collectgarbage(`*`limit`*`)`	Sets the garbage-collection threshold to the given limit (in kilobytes) and checks it against the byte counter. If the new threshold is smaller than the byte counter, Lua immediately runs the garbage collector. If there is no limit parameter, it defaults to zero (0), which forces a garbage-collection cycle. See Lua memory management (on page 6-29) for more information.
`gcinfo()`	Returns the number of kilobytes of dynamic memory that the Test Script Processor (TSP™) scripting engine is using and returns the present garbage collector threshold (also in kilobytes). See Lua memory management (on page 6-29) for more information.
`tonumber(`*`x`*`)` `tonumber(`*`x, base`*`)`	Returns *x* converted to a number. If *x* is already a number, or a convertible string, the number is returned; otherwise, it returns `nil`. An optional argument specifies the base to use when interpreting the numeral. The base may be any integer from 2 to 36, inclusive. In bases above 10, the letter `A` (in either upper or lower case) represents 10, `B` represents 11, and so forth, with `Z` representing 35. In base 10, the default, the number may have a decimal part and an optional exponent. In other bases, only unsigned integers are accepted.
`tostring(`*`x`*`)`	Receives an argument of any type and converts it to a string in a reasonable format.
`type(`*`v`*`)`	Returns the type of its only argument as a string. The possible results of this function are `"nil"` (a string, not the value `nil`), `"number"`, `"string"`, `"boolean"`, `"table"`, `"function"`, `"thread"`, and `"userdata"`.

Lua memory management

Lua automatically manages memory, which means you do not have to allocate memory for new objects and free it when the objects are no longer needed. Lua occasionally runs a garbage collector to collect all objects that are no longer accessible from Lua. All objects in Lua are subject to automatic management, including tables, variables, functions, threads, and strings.

Lua uses two numbers to control its garbage-collection cycles. One number counts how many bytes of dynamic memory Lua is using; the other is a threshold. When the number of bytes crosses the threshold, Lua runs the garbage collector, which reclaims the memory of all inaccessible objects. The byte counter is adjusted, and the threshold is reset to twice the new value of the byte counter.

String library functions

This library provides generic functions for string manipulation, such as finding and extracting substrings. When indexing a string in Lua, the first character is at position 1 (not 0, as in ANSI C). Indices may be negative and are interpreted as indexing backward from the end of the string. Thus, the last character is at position –1.

String library functions

Function	Description

`string.byte(s)`
`string.byte(s, i)`
`string.byte(s, i, j)`
	Returns the internal numeric codes of the characters `s[i], s[i+1], ···, s[j]`. The default value for *i* is `1`; the default value for *j* is *i*.

`string.char(···)`	Receives zero or more integers separated by commas. Returns a string with length equal to the number of arguments, in which each character has the internal numeric code equal to its corresponding argument.

`string.format(`
`formatstring, ...)`
	Returns a formatted version of its variable number of arguments following the description given in its first argument, which must be a string. The format string follows the same rules as the `printf` family of standard C functions. The only differences are that the modifiers `*`, `l`, `L`, `n`, `p`, and `h` are not supported and there is an extra option, `q`. The `q` option formats a string in a form suitable to be safely read back by the Lua interpreter; the string is written between double quotes, and all double quotes, newlines, embedded zeros, and backslashes in the string are correctly escaped when written.
	For example, the call:
	`string.format('%q', 'a string with "quotes" and \n newline')`
	produces the string:
	`"a string with \"quotes\" and \ newline"`
	The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument. `q` and `s` expect a string. This function does not accept string values containing embedded zeros, except as arguments to the `q` option.

`string.len(s)`	Receives a string and returns its length. The empty string `""` has length `0`. Embedded zeros are counted, so `"a\000bc\000"` has length `5`.
`string.lower(s)`	Receives a string and returns a copy of this string with all uppercase letters changed to lowercase. All other characters are left unchanged.
`string.rep(s, n)`	Returns a string that is the concatenation of *n* copies of the string *s*.
`string.sub(s, i)` `string.sub(s, i, j)`	Returns the substring of *s* that starts at *i* and continues until *j*; *i* and *j* can be negative. If *j* is absent, it is assumed to be equal to `-1` (which is the same as the string length). In particular, the call `string.sub(s, 1, j)` returns a prefix of *s* with length *j*, and `string.sub(s, -i)` returns a suffix of *s* with length *i*.
`string.upper(s)`	Receives a string and returns a copy of this string with all lowercase letters changed to uppercase. All other characters are left unchanged.

Math library functions

This library is an interface to most of the functions of the ANSI C math library. All trigonometric functions work in radians. The functions `math.deg()` and `math.rad()` convert between radians and degrees.

Math library functions

Function	Description
`math.abs(x)`	Returns the absolute value of *x*.
`math.acos(x)`	Returns the arc cosine of *x*.
`math.asin(x)`	Returns the arc sine of *x*.
`math.atan(x)`	Returns the arc tangent of *x*.
`math.atan2(y, x)`	Returns the arc tangent of `y/x` but uses the signs of both parameters to find the quadrant of the result (it also correctly handles the case of *x* being zero).
`math.ceil(x)`	Returns the smallest integer larger than or equal to *x*.
`math.cos(x)`	Returns the cosine of *x*.
`math.deg(x)`	Returns the angle *x* (given in radians) in degrees.
`math.exp(x)`	Returns the value e^x.
`math.floor(x)`	Returns the largest integer smaller than or equal to *x*.
`math.frexp(x)`	Returns `m` and `e` such that $x = m2^e$, where `e` is an integer and the absolute value of `m` is in the range `[0.5, 1]` (or zero when *x* is zero).
`math.ldexp(m, e)`	Returns $m2^e$ (`e` should be an integer).
`math.log(x)`	Returns the natural logarithm of *x*.
`math.log10(x)`	Returns the base-10 logarithm of *x*.
`math.max(x, ...)`	Returns the maximum value among its arguments.
`math.min(x, ...)`	Returns the minimum value among its arguments.
`math.pi`	The value of π (3.141592654).
`math.pow(x, y)`	Returns x^y (you can also use the expression `x^y` to compute this value).
`math.rad(x)`	Returns the angle *x* (given in degrees) in radians.

`math.random()`
`math.random(m)`
`math.random(m, n)`
	This function is an interface to the simple pseudorandom generator function `rand` provided by ANSI C.
	When called without arguments, returns a uniform pseudorandom real number in the range `[0,1]`. When called with an integer number *m*, `math.random()` returns a uniform pseudorandom integer in the range `[1, m]`. When called with two integer numbers *m* and *n*, `math.random()` returns a uniform pseudorandom integer in the range `[m, n]`.

`math.randomseed(x)`	Sets *x* as the seed for the pseudorandom generator; equal seeds produce equal sequences of numbers.
`math.sin(x)`	Returns the sine of *x*.
`math.sqrt(x)`	Returns the square root of *x*. You can also use the expression `x^0.5` to compute this value.
`math.tan(x)`	Returns the tangent of *x*.

Programming example: Script with a for loop

The following script puts a message on the front-panel display one character at a time. This example demonstrates:

- The use of a `for` loop
- Simple display remote commands
- Simple Lua string manipulation

NOTE

When creating a script using the TSB Embedded, you do not need the shell commands `loadscript` and `endscript`, as shown in the following examples.

Example: User script

<table>
<tr><th>User script created in TSB Embedded</th><th>User script created in user's program</th></tr>
<tr><td></td><td><pre>loadscript</pre></td></tr>
<tr><td><pre>display.clear()
myMessage = "Hello World!"
for k = 1, string.len(myMessage) do
   x = string.sub(myMessage, k, k)
   display.settext(x)
   print(x)
   delay(1)
end</pre></td><td><pre>display.clear()
myMessage = "Hello World!"
for k = 1, string.len(myMessage) do
   x = string.sub(myMessage, k, k)
   display.settext(x)
   print(x)
   delay(1)
end</pre></td></tr>
<tr><td></td><td><pre>endscript</pre></td></tr>
</table>

Example: Create channels with a for loop

<table>
<tr><th>User script created in TSB Embedded</th><th>User script created in another program</th></tr>
<tr><td><pre>

-- Pseudocard assignment necessary
-- only if slot is empty.
-- slot[1].pseudocard = 7072
for k = 1, 8, 2 do
   chan1 =
      channel.createspecifier(1, k, 1)
   chan2 =
      channel.createspecifier(1, k, 2)
   print(chan1 .. "," .. chan2)
   scan.addimagestep(chan1..","..chan2)
end</pre></td><td><pre>loadscript

-- Pseudocard assignment necessary
-- only if slot is empty.
-- slot[1].pseudocard = 7072
for k = 1,8,2 do
   chan1 =
      channel.createspecifier(1, k, 1)
   chan2 =
      channel.createspecifier(1, k, 2)
   print(chan1 .. "," .. chan2)
   scan.addimagestep(chan1 .. "," ..
    chan2)
end
endscript</pre></td></tr>
<tr><th colspan="2">Output</th></tr>
<tr><td colspan="2"><pre>1A01,1A02
1C01,1C02
1E01,1E02
1G01,1G02</pre></td></tr>
</table>

Using Test Script Builder (TSB)

Keithley Test Script Builder (TSB) is a software tool that simplifies building test scripts. You can use TSB to perform the following operations:

- Send remote commands and Lua statements
- Receive responses (data) from commands and scripts
- Upgrade instrument firmware
- Create, manage, and run user scripts
- Debug scripts
- Import factory scripts to view or edit and convert to user scripts

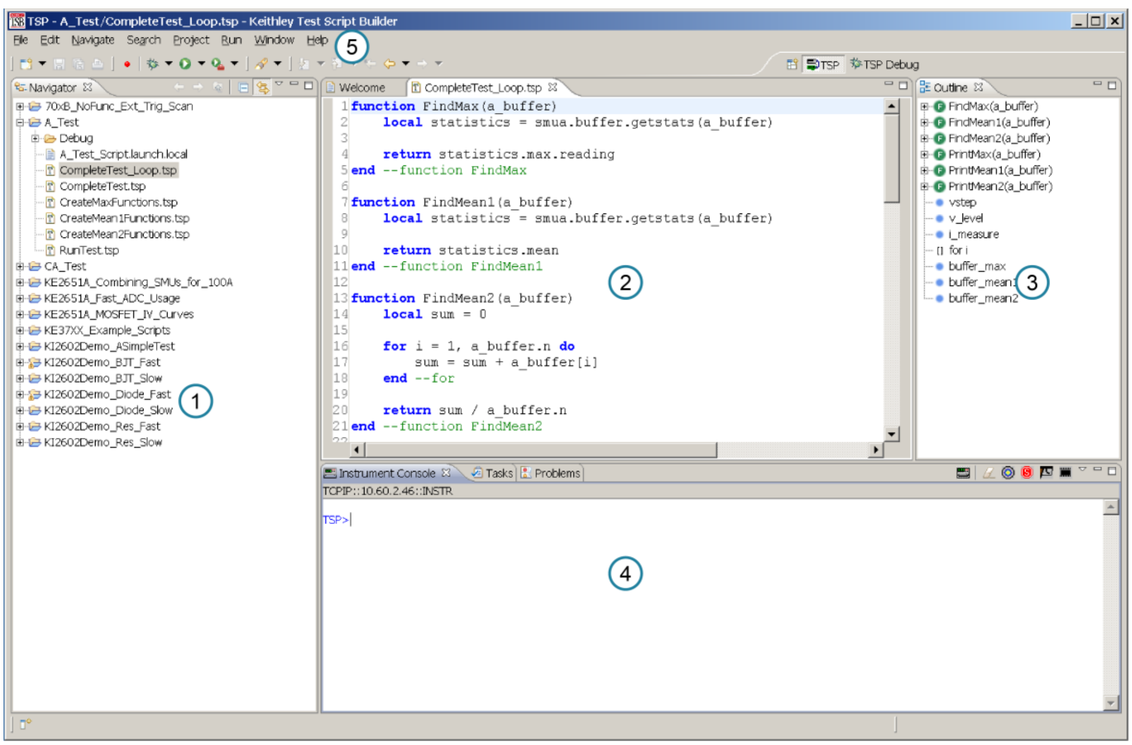
The Keithley Test Script Processor (TSP^{TM}) scripting engine is a Lua interpreter. In TSP-enabled instruments, the Lua programming language has been extended with control commands that are specific to Keithley instruments. For more information about using the Lua scripting language with Keithley TSP-enabled instruments, refer to the Fundamentals of programming for TSP (on page 6-13).

Keithley has created a collection of remote commands specifically for use with Keithley TSP-enabled instruments; for detailed information about those commands, refer to the "Command reference" section of the documentation for your specific instrument. You can build scripts from a combination of these commands and Lua programming statements. Scripts that you create are referred to as "user scripts." Also, some TSP-enabled instruments include built-in factory scripts.

The following figure shows an example of the Test Script Builder. As shown, the workspace is divided into these areas:

- Project navigator
- Script editor
- Outline view
- Programming interaction
- Help files

Figure 68: Example of the Test Script Builder workspace



Item	Description
1	Project navigator
2	Script editor; right-click to run the script that is displayed
3	Outline view
4	Programming interaction
5	Help; includes detailed information on using Test Script Builder

Install the TSB software

The installation files for the TSB software are available at tek.com/keithley.

To install the TSB software:

1. Close all programs.
2. Download the installer to your computer and double-click the `.exe` file to start the installation.
3. Follow the on-screen instructions.

Project navigator

The project navigator consists of project folders and the script files (.tsp) created for each project. Each project folder can have one or more script files.

To view the script files in a project folder, select the plus (+) symbol next to the project folder. To hide the folder contents, select the minus (−) symbol next to the project folder.

You can download a TSP project to the instrument and run it, or you can run it from the TSB interface.

Script editor

The script editor is where you write, modify, and debug scripts.

To open and display a script file, double-click the file name in the project navigator. You can have multiple script files open in the script editor at the same time. Each open script file is displayed on a separate tab.

To display another script file that is already open, select the tab that contains the script in the script editor area.

Programming interaction

This part of the workspace is where you interact with the scripts that you are building in Test Script Builder (TSB). The actual contents of the programming interaction area of the workspace can vary.

You can send commands from the Instrument Console command line, retrieve data, view variables and errors, and view and set breakpoints when using the debug feature. For additional information, refer to the online help that is accessible from Test Script Builder (TSB).

Advanced scripting for TSP

The following topics describe advanced information that can help you understand how the Test Script Processor (TSP™) scripting engine works.

Global variables and the script.user.scripts table

When working with script commands, it is helpful to understand how scripts are handled in the instrument.

Scripts are loaded into the runtime environment from nonvolatile memory when you turn the instrument on. They are also added to the runtime environment when you load them into the instrument.

A script in the runtime environment can be:

- A named script
- An unnamed script
- The anonymous script (which is a special unnamed script)

Script names can be assigned by using the loadscript command or by defining the `scriptVar` parameter of the `script.new()` function. When a named script is loaded into the runtime environment:

- A global variable with the same name is created so that you can reference the script more conveniently.
- An entry for the script is added to the `script.user.scripts` table.

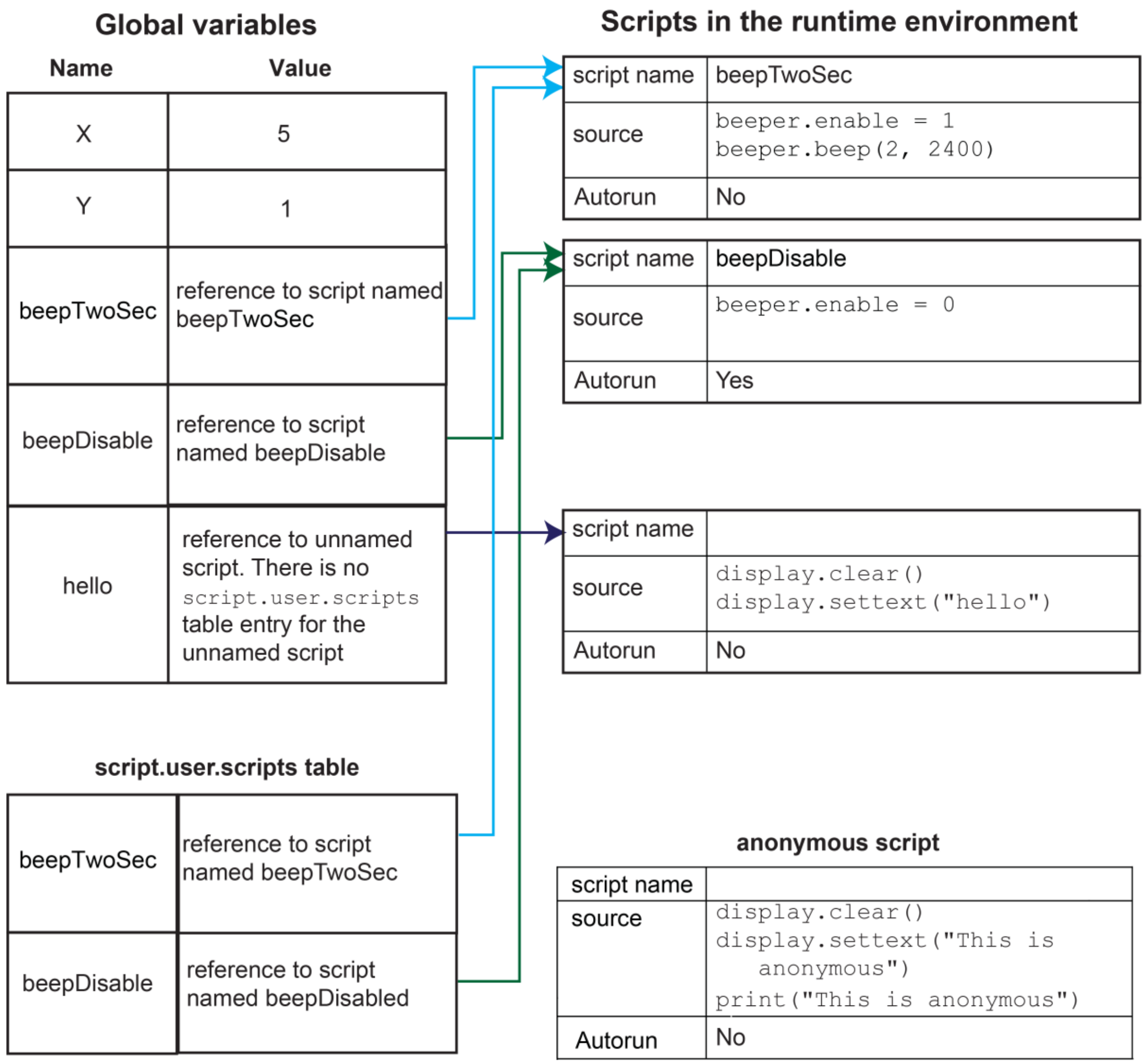
When you create a script using the `script.new()` function without providing a name, the script is added to the runtime environment as an unnamed script. The `script.new()` function returns the script, but the script is not added to the `script.user.scripts` table.

When the anonymous script is loaded, it does not have a global variable or an entry in the `script.user.scripts` table. If there is an existing anonymous script, it is replaced by the new one.

When the instrument is turned off, everything in the runtime environment is deleted, including the scripts and global variables.

See the following figure to see how the scripts, global variables, and `script.user.scripts` table interrelate.

Figure 69: Global variables and scripts in the runtime environment



Create a script using the script.new() command

Use the `script.new()` function to copy an existing script from the local node to a remote node. This enables parallel script execution.

You can create a script with the `script.new()` function using the command:

```
scriptVar = script.new(code, name)
```

Where:

scriptVar	=	Name of the variable created when the script is loaded into the runtime environment
code	=	Content of the script
name	=	Name that is added to the `script.user.scripts` table

For example, to set up a two-second beep, you can send the command:

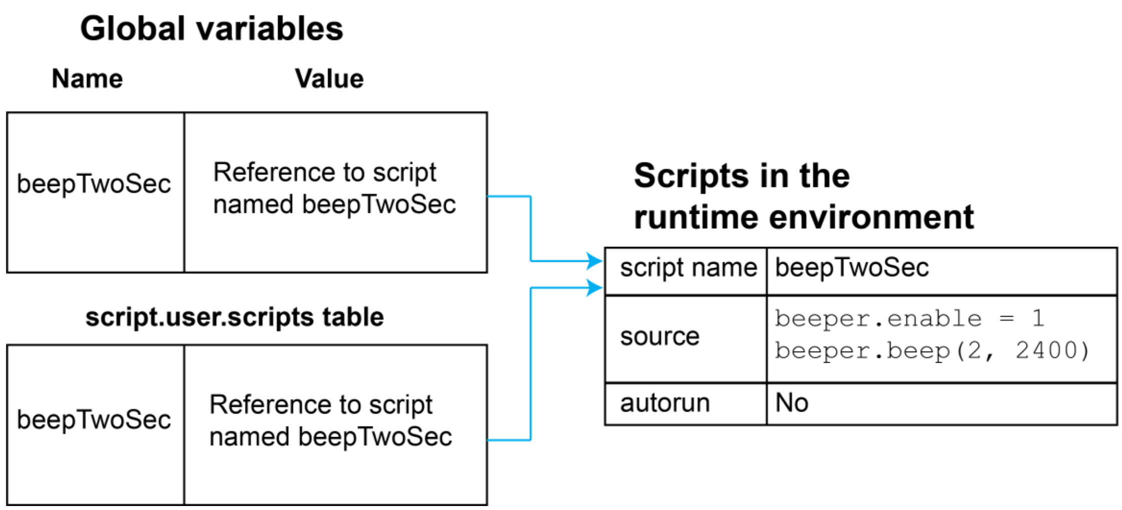
```
beepTwoSec = script.new("beeper.enable = 1 beeper.beep(2, 2400)", "beepTwoSec")
```

To run the new script, send the command:

```
beepTwoSec()
```

When you add `beepTwoSec`, the global variable and `script.user.scripts` table entries are made to the runtime environment, as shown in the following figure.

Figure 70: Runtime environment after creating a script



Create an unnamed script using script.new()

NOTE

Unnamed scripts are not available from the front-panel display of the instrument. Only the anonymous script and named scripts are available from the front-panel display.

When you create a script using `script.new()`, if you do not include *name*, the script is added to the runtime environment as an unnamed script. The `script.new()` function returns the script. You can assign it to a global variable, a local variable, or ignore the return value. A global variable is not automatically created.

For example, send the following command:

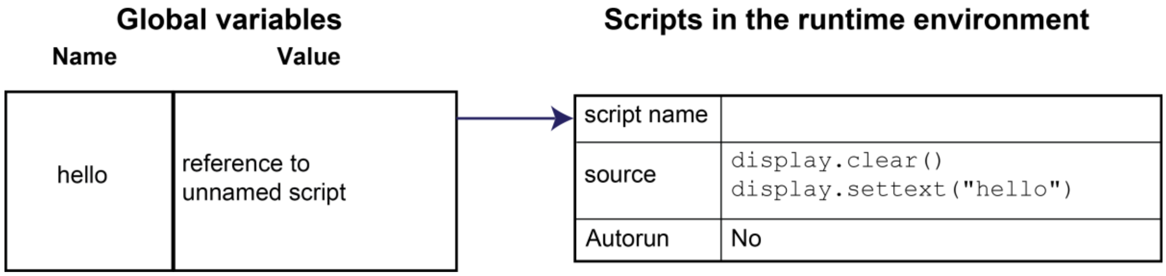
```
hello = script.new('display.clear() display.settext("hello")')
```

A script is created in the runtime environment and a global variable is created that references the script.

To run the script, send the command:

```
hello()
```

Figure 71: Create an unnamed script



A script becomes unnamed if you create a new script with the same name. In this circumstance, the name of the script in the `script.user.scripts` table is set to an empty string before it is replaced by the new script.

For example, if `beepTwoSec` already exists in the `script.user.scripts` table and you sent:

```
beepTwoSec1200 = script.new("beeper.enable = 1 beeper.beep(2, 1200)", "beepTwoSec")
```

The following actions occur:

- `beepTwoSec1200` is added as a global variable.
- The script that was in the runtime environment as `beepTwoSec` is changed to an unnamed script (the name attribute is set to an empty string).
- The global variable `beepTwoSec` remains in the runtime environment unchanged (it points to the now unnamed script).
- A new script named `beepTwoSec` is added to the runtime environment.

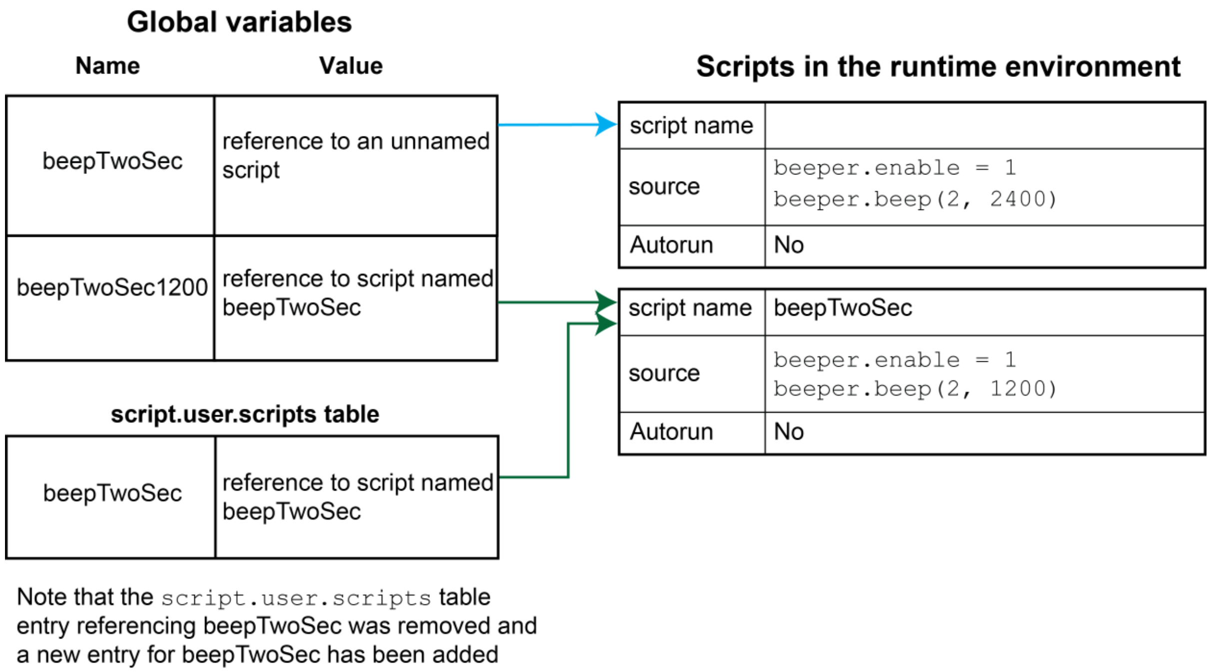
In this example, you can access the new script by sending either of the following commands:

```
beepTwoSec1200()
script.user.scripts.beepTwoSec()
```

To access the unnamed script, you can send the command:

```
beepTwoSec()
```

Figure 72: Create a new script with the name of an existing script



Rename a script

You can rename a script to a new name or be the autoexec script.

To change the name of a script, use the command:

```
scriptVar.name = "renamedScript"
```

Where:

scriptVar	=	The global variable name
"*renamedScript*"	=	The new name of the user script that was referenced by the *scriptVar* global variable

After changing the name, you need to save the original script to save the change to the name attribute.

For example:

```
beepTwoSec.name = "beep2sec"
beepTwoSec.save()
```

Run the `beep2sec` script using the following command:

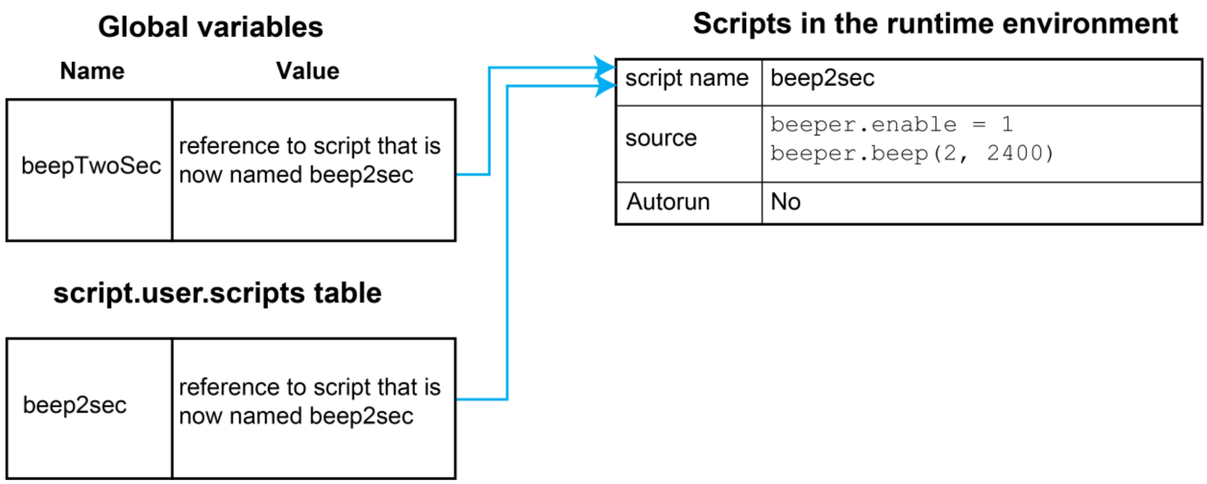
```
script.user.scripts.beep2sec()
```

NOTE

If the new name is the same as a name that is already used for a script, the name of the existing script is removed and that script becomes unnamed. This removes the existing script if there are no other variables that reference the previous script. If variables do reference the existing script, the references remain intact.

Changing the name of a script does not change the name of any variables that reference that script. After changing the name, the script is in the `script.user.scripts` table under its new name.

Figure 73: Rename script



For example, to change the name of the script named `test2` to be `autoexec`:

```
test2.name = "autoexec"
test2.save()
```

The `autoexec` script runs automatically when the instrument is turned on. It runs after all the scripts have loaded and any scripts marked as autorun have run.

NOTE

You can also use the `script.new()` and the *`scriptVar`*`.source` attribute commands to create a script with a new name. For example, if you had an existing script named `test1`, you could create a new script named `test2` by sending the command:

`test2 = script.new(test1.source, "test2")`

Refer to script.new() (on page 7-161).

Retrieve a user script

There are several ways to retrieve the source code of a user script:

- **One line at a time:** Use `scriptVar.list()` to retrieve the source code one line at a time.
- **Entire script:** Use the `print(scriptVar.source)` command to retrieve the script source code as a single string.
- **Use TSB Embedded:** Refer to TSB Embedded (on page 4-53) for more information.

See Create and load a script (on page 6-4) for information about recreating the script and loading it back into the instrument.

NOTE

To get a list of scripts that are in nonvolatile memory, use the script.user.catalog() (on page 7-164) function.

Retrieve source code one line at a time

To retrieve the source code one line at a time, send the `scriptVar.list()` command. When this command is received, the instrument sends the entire script. Each line of the script is sent as a separate response message. The output includes the `loadscript` or `loadandrunscript` and `endscript` keywords.

After retrieving the source code, you can modify and save the command lines as a user script under the same name or a new name.

To retrieve the source code of a script one line at a time, send the command:

```
scriptVar.list()
```

Where *scriptVar* is the name of the script.

NOTE

To retrieve the commands in the anonymous script, use `script.anonymous.list()`.

Example: Retrieve source code one line at a time

`test.list()`	Retrieve the source of a script named "`test`".
	The output looks similar to:
	`loadscript test`
	`display.clear()`
	`display.settext("This is a test")`
	`print("This is a test")`
	`endscript`

Retrieve a script as a single string

To retrieve the entire user script source code as a single string, use the *scriptVar*.source attribute. The loadscript or loadandrunscript and endscript keywords are not included.

To retrieve the source code as a single string, send the command:

```
print(scriptVar.source)
```

Where *scriptVar* is the name of the script.

Example: Retrieve the source code as a single string

`print(test.source)`	Retrieve the source of a script named "test".
	Output looks similar to:
	`display.clear() display.settext("This is a test")`
	`print("This is a test")`

Retrieve a script using TSB Embedded

In TSB Embedded, from the User Scripts list, select the script you want to retrieve. The contents of the script are displayed.

Script example: Retrieve the content of scripts

This set of examples:

- Retrieves the source of a script using *scriptVar*.list()
- Retrieves the source of a script using *scriptVar*.source

Example: Retrieve the content of a script with scriptVar.list()

`test.list()`	Request a listing of the source of test.
	An example of the possible instrument output is shown here (note that the loadscript and endscript commands are included).
	
	Output:
	`loadscript scriptVarTest`
	`listTones = {100, 400, 800}`
	`for index in listTones do`
	`beeper.beep(.5, listTones[index])`
	`end`
	`endscript`

Example: Retrieve the content of a script with scriptVar.source

`print(test.source)`	Request a listing of the source of the script named `test`. The `loadscript` and `endscript` commands are not included.
	An example of the possible instrument output is:
	`listTones = {100, 400, 800}`
	`for index in listTones do`
	`   beeper.beep(.5, listTones[index])`
	`end`

Delete user scripts from the instrument

In most circumstances, you can delete a script using `script.delete()` (as described in Delete user scripts (on page 6-12)), and then turn the instrument off and back on again. However, if you cannot turn the instrument off, you can use the following steps to completely remove a script from the instrument.

When you completely remove a script, you delete all references to the script from the runtime environment, the `script.user.scripts` table, and nonvolatile memory.

To completely remove a script:

1. **Remove the script from the runtime environment.** Set any variables that refer to the script to `nil` or assign the variables a different value. For example, to remove the script `"beepTwoSec"` from the runtime environment, send the following code:
   ```
   beepTwoSec = nil
   ```
2. **Remove the script from the script.user.scripts table.** Set the name attribute to an empty string (`""`). This makes the script nameless, but does not make the script become the anonymous script. For example, to remove the script named `"beepTwoSec"`, send the following code:
   ```
   script.user.scripts.beepTwoSec.name = ""
   ```
3. **Remove the script from nonvolatile memory.** To delete the script from nonvolatile memory, send the command:
   ```
   script.delete("name")
   ```
   Where *`name`* is the name that the script was saved as. For example, to delete `"beepTwoSec"`, send:
   ```
   script.delete("beepTwoSec")
   ```

Restore a script to the runtime environment

You can retrieve a script that was removed from the runtime environment but is still saved in nonvolatile memory.

To restore a script from nonvolatile memory into the runtime environment, you can use `script.restore("scriptName")`, where *`scriptName`* is the user-defined name of the script to be restored.

For example, to restore a user script named `"test9"` from nonvolatile memory:

```
script.restore("test9")
```

Memory considerations for the runtime environment

The runtime environment has a fixed amount of memory for storing user scripts and other runtime information.

You can check the amount of memory in the instrument using the `memory.used()` and `memory.available()` functions. These functions return the percentage of memory that is used or available. When you send this command, memory used or available is returned as a comma-delimited string with percentages for used memory.

The amount of memory is returned in the format *`systemMemory`*, *`scriptMemory`*, *`patternMemory`*, where:

- *`systemMemory`* is the percentage of memory used or available in the instrument
- *`scriptMemory`* is the percentage of memory used or available in the instrument to store user scripts
- *`patternMemory`* is the percentage of memory used or available in the instrument to store channel patterns

For example, if you send the command:

```
MemUsed = memory.used()
print(MemUsed)
```

The return is similar to `69.14, 0.16, 12.74`, where `69.14` is the percentage of memory used in the instrument, `0.16` is the percentage used for script storage, and `12.74` is the percentage used for channel pattern storage.

Refer to memory.available() (on page 7-135) and memory.used() (on page 7-136) for more detail on using these functions.

To increase the available memory:

- Turn the instrument off and on. This deletes scripts that have not been saved and reloads only scripts that have been stored in nonvolatile memory.
- Remove unneeded scripts from nonvolatile memory. Scripts are loaded from nonvolatile memory into the runtime environment when the instrument is turned on. See Delete user scripts from the instrument (on page 6-44).
- Reduce the number of TSP-Link™ nodes.
- Delete unneeded channel patterns (this affects only pattern memory, not instrument memory). See Channel patterns (on page 2-14).
- Delete unneeded global variables from the runtime environment by setting them to `nil`.
- Set the source attribute of all scripts to `nil`.
- Adjust the `collectgarbage()` settings in Lua. See Lua memory management (on page 6-29) for information.
- Review scripts to optimize their memory usage. In particular, you can see memory gains by changing string concatenation lines into a Lua table of string entries. You can then use the `table.concat()` function to create the final string concatenation.

The following example shows an example that optimizes a channel pattern that consists of five channels.

Example

String concatenation lines	Optimized with the table.concat function

`testPattern = "1A03"`
`testPattern = testPattern .. ",1B03"`
`testPattern = testPattern .. ",1C03"`
`testPattern = testPattern .. ",1D03"`
`testPattern = testPattern .. ",1E03"`
`print(testPattern)`
	`testTable = { }`
	`testTable[1] = "1A03,"`
	`testTable[2] = "1B03,"`
	`testTable[3] = "1C03,"`
	`testTable[4] = "1D03,"`
	`testTable[5] = "1E03"`
	`testPattern = table.concat(testTable)`
	`print(testPattern)`

The output is: `1A03,1B03,1C03,1D03,1E03`	The output is: `1A03,1B03,1C03,1D03,1E03`

CAUTION

If the instrument encounters memory allocation errors when memory used is above 95 percent, the state of the instrument cannot be guaranteed. After attempting to save any important data, it is recommended that you turn off power to the instrument and turn it back on to return the instrument to a known state. Cycling power resets the runtime environment. Unsaved scripts and channel patterns are lost.

TSP-Link system and running simultaneous test scripts

You can use the TSP-Link™ expansion interface to expand your test system to include up to 32 addressable TSP™ enabled instruments that use the TSP-LINK interface. The expanded system can be stand-alone or computer-based.

Stand-alone system: You can run a script from the front panel of any instrument (node) connected to the system. When a script is run, all nodes in the system go into remote operation (REM indicators turn on). The node running the script becomes the master and can control all of the other nodes, which become its subordinates. When the script is finished running, all the nodes in the system return to local operation (REM indicators turn off), and the master/subordinate relationship between nodes is dissolved.

Computer-based system: You can use a computer and a LAN, GPIB, or RS-232 interface to any single node in the system. This node becomes the interface to the entire system. When a command is sent through this node, all nodes go into remote operation (REM indicators turn on). The node that receives the command becomes the master and can control all of the other nodes, which become its subordinates. In a computer-based system, the master/subordinate relationship between nodes can only be dissolved by performing an abort operation.

TSP-Link nodes

Each instrument (node) attached to the TSP-Link™ network must be identified by assigning it a unique TSP-Link node number.

Commands for remote nodes are stored in the `node` table. An individual node is accessed as `node[N]`, where *N* is the node number assigned to the node.

All TSP-accessible remote commands can be accessed as elements of the specific node. The following attributes are examples of items you can access:

- `node[N].model`: The product model number string of the node.
- `node[N].revision`: The product revision string of the node.
- `node[N].serialno`: The product serial number string of the node.

You do not need to know the node number of the node that is running a script. The variable `localnode` is an alias for the node entry of the node where the script is running. For example, if a script is running on node 5, you can use the global variable `localnode` as an alias for `node[5]`. To access the product model number for this example, use `localnode.model`.

Connect the TSP-Link cable

Connect the TSP-Link connector to one of the TSP-Link connectors on the rear panel of the instrument.

The locations of the TSP-Link connectors are shown in the following figures.

NOTE

For an example of setting up a TSP-Linked system, see Work with the Series 2600B (on page 9-1).

Figure 74: Model 708B rear-panel TSP-Link connection

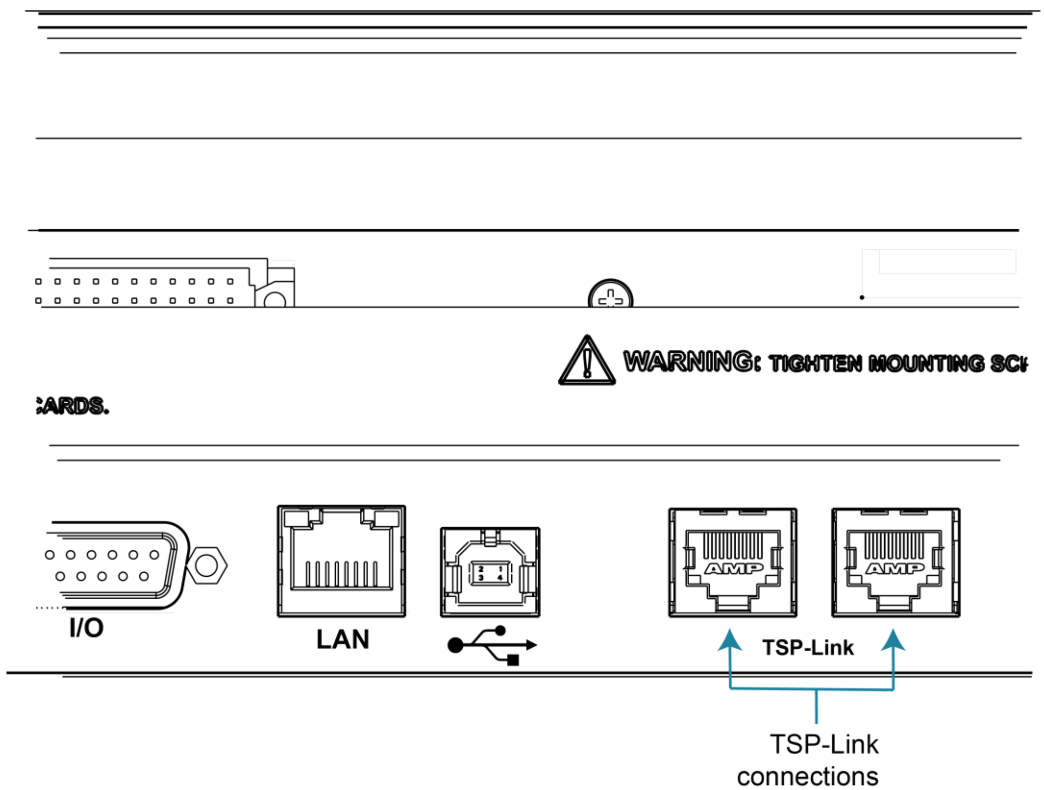
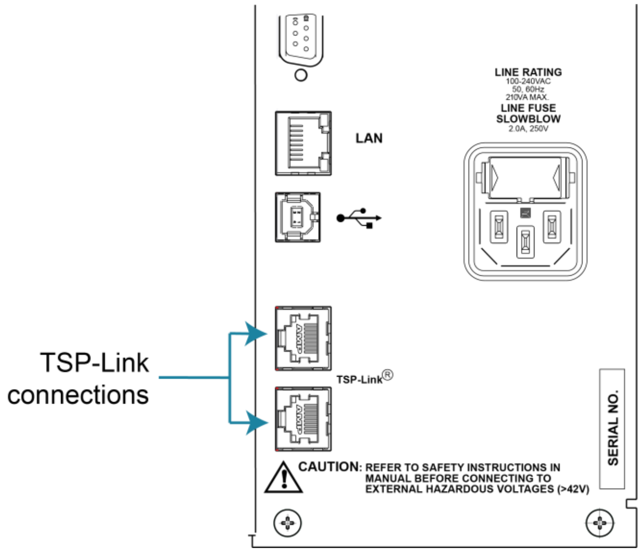


Figure 75: Model 707B rear-panel TSP-Link connection



Initialization

Before you can use a TSP-Link™ system, it must be initialized. For initialization to succeed, each instrument in a TSP-Link system must be assigned a different node number.

Assign node numbers

At the factory, each 707B and 708B instrument is assigned as node 1. The node number is stored in nonvolatile memory and remains in storage when the instrument is turned off. You can assign a node number to a Models 707B and 708B using the front panel or by using a remote command. There can only be 32 physical nodes, but you can assign node numbers from 1 to 64.

To assign a node number from the front panel of the instrument:

1. Press the **MENU** key, then select **TSPLINK > NODE**.
2. Press the navigation wheel ◷ and select the node number.
3. Press the **ENTER** key to save the number.

To assign a node number using a remote command:

Set the `tsplink.node` attribute of the instrument:

```
tsplink.node = N
```

Where *N* = `1` to `64`

To determine the node number of an instrument, you can read the `tsplink.node` attribute by sending the following command:

```
print(tsplink.node)
```

The node number is output. For example, if the node number is 1, a `1` is displayed.

Reset the TSP-Link network

After all the node numbers are set, you must initialize the system by performing a TSP-Link™ network reset.

NOTE

If you change the system configuration after initialization, you must reinitialize the system by performing a TSP-Link network reset. Changes that require that you reinitialize the TSP-Link network include turning off power or rebooting any instrument in the system, or rearranging or disconnecting the TSP-Link cable connections between instruments.

Front-panel operation

To reset the TSP-Link™ network from the front panel:

1. Power on all instruments connected to the TSP-Link network.
2. Press the **MENU** key, select **TSPLINK**, and then press the **ENTER** key.
3. Turn the navigation wheel ◷ to select **RESET**, and then press the **ENTER** key.

Remote programming

The commands associated with the TSP-Link™ system reset are listed in the following table.

TSP-Link reset commands

Command	Description
`tsplink.reset()`	Initializes the TSP-Link network
`tsplink.state`	Reads the state of the TSP-Link network: ■ `online` if the most recent TSP-Link reset was successful ■ `offline` if the reset operation failed

An attempted TSP-Link reset operation fails if any of the following conditions are true:

- Two or more instruments in the system have the same node number
- There are no other instruments connected to the instrument performing the reset (only if the expected number of nodes was not provided in the reset call)
- One or more of the instruments in the system is turned off
- The actual number of nodes is less than the expected number

The following code illustrates a TSP-Link reset operation and displays its state:

```
tsplink.reset()
print(tsplink.state)
```

If the reset operation is successful, `online` is output to indicate that communications with all nodes have been established.

Use TSP to run test scripts simultaneously

You can use TSP to run test scripts simultaneously on multiple nodes on the TSP-Link network. Running test scripts simultaneously improves functional testing, provides higher throughput, and expands system flexibility.

You can also use TSP to manage the resources that are allocated to the test scripts that are running simultaneously, and to use the data queue to facilitate real-time communications between nodes.

There are two methods you can use to run test scripts in parallel:

- Create multiple TSP-Link networks
- Use a single TSP-Link network with groups

The following table describes the functions of a single TSP-Link network. Each group in this example runs multiple test scripts at the same time or in parallel.

TSP-Link network group functions

Group number	Group members	Present function
0	Master node	▪ Initiates and runs a test script on Node 2 ▪ Initiates and runs a test script on Node 5 ▪ Initiates and runs a test script on Node 6
1	Group leader Node 2	▪ Runs the test script initiated by the master node ▪ Initiates remote operations on Node 3
	Node 3	▪ Performs remote operations initiated by Node 2
2	Group leader Node 5	▪ Runs the test script initiated by the master node ▪ Initiates remote operations on Node 4
	Node 4	▪ Performs remote operations initiated by Node 5
3	Group leader Node 6	▪ Runs the test script initiated by the master node

TSP-Link has three synchronization lines that function similarly to the digital I/O synchronization lines. See Digital I/O (on page 5-7) and Hardware trigger modes (on page 2-33) for more detailed information.

Use groups to manage nodes on TSP-Link network

The primary purpose of groups is to allow each group to run a different test script simultaneously.

A group can consist of one or more nodes. You must assign group numbers to each node using remote commands. If you do not assign a node to a group, it defaults to group 0, which is always grouped with the master node (regardless of the group to which the master node is assigned).

Master node overview

You can assign the master node to any group. You can also include other nodes in the group that includes the master. Note that any nodes that are set to group 0 are automatically included in the group that contains the master node, regardless of the group that is assigned to the master node.

The master node is always the node that coordinates activity on the TSP-Link network.

The master node:

- Is the only node that can use the `execute()` command on a remote node
- Cannot initiate remote operations on any node in a remote group if any node in that remote group is performing an overlapped operation (a command that continues to operate after the command that initiated it has finished running)
- Can execute the `waitcomplete()` command to wait for the group to which the master node belongs; to wait for another group; or to wait for all nodes on the TSP-Link network to complete overlapped operations (overlapped commands allow the execution of subsequent commands while device operations of the overlapped command are still in progress)

Group leader overview

Each group has a dynamic group leader. The last node in a group running any operation initiated by the master node is the group leader.

The following list describes the functionality of the group leader:

- Runs operations initiated by the master node
- Initiates remote operations on any node with the same group number
- Cannot initiate remote operations on any node with a different group number
- Can send the `waitcomplete` command without a parameter to wait for all nodes assigned to the same group number

Assign groups

Group numbers can range from zero (0) to 64. The default group number is 0. You can change the group number at any time. You can also add or remove a node to or from a group at any time.

Each time the power for a node is turned off, the group number for that node changes to 0.

The following example code dynamically assigns a node to a group:

```
-- Assign node 3 to group 1.
node[3].tsplink.group = 1
```

Run simultaneous test scripts

You can send the `execute()` command from the master node to initiate a test script and Lua code on a remote node. The `execute()` command places the remote node in the overlapped operation state. As a test script runs on the remote node, the master node continues to process other commands simultaneously.

Use the following code to send the `execute()` command for a remote node. The *N* parameter represents the node number that runs the test script (replace *N* with the node number).

To set the global variable "setpoint" on node N to 2.5:

```
node[N].execute("setpoint = 2.5")
```

The following code runs a test script that is defined on the local node. For this example, *scriptVar* is defined on the local node, which is the node that initiates the code to run on the remote node. The local node must be the master node.

To run scriptVar on node N:

```
node[N].execute(scriptVar.source)
```

The following code runs a test script that is defined on a remote node. For this example, *scriptVar* is defined on the remote node.

To run a script defined on the remote node:

```
node[N].execute("scriptVar()")
```

It is recommended that you copy large scripts to a remote node to improve system performance. See Copying test scripts across the TSP-Link network (on page 6-55) for more information.

Coordinate overlapped operations in remote groups

All overlapped operations on all nodes in a group must have completed before the master node can send a command to the group. If you send a command to a node in a remote group when an overlapped operation is running on any node in that group, errors occur.

You can execute the `waitcomplete()` command on the master node or group leader to wait for overlapped operations. The action of `waitcomplete()` depends on the parameters specified.

If you want to wait for completion of overlapped operations for:

- **All nodes in the local group:** Use `waitcomplete()` without a parameter from the master node or group leader.
- **A specific group:** Use `waitcomplete(N)` with a group number as the parameter from the master node. This option is not available for group leaders.
- **All nodes in the system:** Use `waitcomplete(0)` from the master node. This option is not available for group leaders.

For additional information, refer to waitcomplete() (on page 7-253).

The following code shows two examples that use the `waitcomplete()` command from the master node:

```
-- Wait for each node in group N to complete all overlapped operations.
waitcomplete(N)
-- Wait for all groups on the TSP-Link network to complete overlapped operations.
waitcomplete(0)
```

A group leader can issue the `waitcomplete()` command to wait for the local group to complete all overlapped operations.

The following code is an example of how to use the `waitcomplete()` command from a group leader:

```
-- Wait for all nodes in the local group to complete all overlapped operations.
waitcomplete()
```

NOTE

Presently, the Models 707B and 708B have no overlapped commands implemented. However, other TSP-enabled products, such as the Series 2600B System SourceMeter™ Instruments, have overlapped commands. Therefore, when the Model 707B or 708B is a TSP master to a subordinate device with overlapped commands, use this function to wait until all overlapped operations are complete.

Use the data queue for real-time communications

Nodes that are running test scripts at the same time can store data in the data queue for real-time communications. Each instrument has an internal data queue that uses a first-in, first-out (FIFO) structure to store data. You can use the data queue to post numeric values, strings, and tables.

Use the data queue commands to:

- Share data between test scripts running in parallel
- Access data from a remote group or a local node on a TSP-Link network at any time

You cannot access the reading buffers or global variables from any node in a remote group while a node in that group is performing an overlapped operation. However, you can use the data queue to retrieve data from any node in a group that is performing an overlapped operation. In addition, the master node and the group leaders can use the data queue to coordinate activities.

Tables in the data queue consume one entry. When a node stores a table in the data queue, a copy of the data in the table is made. When the data is retrieved from the data queue, a new table is created on the node that is retrieving the data. The new table contains a separate copy of the data in the original table, with no references to the original table or any subtables.

You can access data from the data queue even if a remote group or a node has overlapped operations in process. See the `dataqueue` (on page 5-7) commands for more information.

Copy test scripts across the TSP-Link network

To run a large script on a remote node, copy the test script to the remote node to increase the speed of test script initiation.

The code in the following example copies a test script across the TSP-Link™ network, creating a copy of the script on the remote node with the same name.

```
-- Add the source code from the script
-- testScript to the data queue.
node[2].dataqueue.add(testScript.source)
-- Create a new script on the remote node
-- using the source code from testScript.
node[2].execute(testScript.name ..
   "= script.new(dataqueue.next(), [[" .. testScript.name .. "]])")
```

TSP-Net

TSP-Net provides a simple socket-like programming interface to Test Script Processor (TSP) enabled instruments. Using the TSP-Net library, the Models 707B and 708B can control ethernet-enabled devices directly through its LAN port. This enables the Models 707B and 708B to communicate directly with a device that is that is not TSP-enabled without the use of a controlling computer.

Using TSP-Net library methods, you can transfer string data to and from a remote instrument, transfer and format data into Lua variables, and clear input buffers. The TSP-Net library is only accessible using commands from a remote command interface when you are using the TSP command language.

While you can use TSP-Net commands to communicate with any ethernet-enabled instrument, specific TSP-Net commands exist for TSP-enabled instruments to allow for support of features unique to the TSP scripting engine. These features include script downloads, reading buffer access, wait completion, and handling of TSP scripting engine prompts.

Using TSP-Net commands with TSP-enabled instruments, a 707B and 708B can download a script to another TSP-enabled instrument and have both instruments run scripts independently. The 707B and 708B can read the data from the remote instrument and either manipulate the data or send the data to a different remote instrument on the LAN.

You can use TSP-Net to connect to a computer; you can use a script on the instrument to transfer data directly to your computer hard drive.

With TSP-Net, you can simultaneously connect to a maximum of 32 devices using standard TCP/IP networking techniques through the LAN port of the 707B and 708B.

Use TSP-Net with any ethernet-enabled instrument

NOTE

Refer to TSP command reference (on page 7-1) for details about the commands presented in this section.

The 707B and 708B LAN port is auto-sensing (Auto-MDIX), so you can use either a LAN crossover cable or a LAN straight-through cable to connect directly from the Models 707B and 708B to an ethernet device or to a hub.

To set up communications to a remote ethernet-enabled instrument that is enabled for TSP™:

1. Send the following command to configure TSP-Net to send an abort command when a connection to a TSP instrument is established:

```
tspnet.tsp.abortonconnect = 1
```

   If the scripts are allowed to run, the connection is made, but the remote instrument may be busy.

2. Send the command:

```
connectionID = tspnet.connect(ipAddress)
```

   Where:

   - *connectionID* is the connection ID that is used as a handle in all other TSP-Net function calls.
   - *ipAddress* is the IP address, entered as a string, of the remote instrument.

See tspnet.connect() (on page 7-239) for additional detail.

To set up communications to a remote ethernet-enabled device that is not enabled for TSP:

Send the command:

```
connectionID = tspnet.connect(ipAddress, portNumber, initString)
```

Where:

- *connectionID* is the connection ID that is used as a handle in all other `tspnet` function calls.
- *ipAddress* is the IP address, entered as a string, of the remote device.
- *portNumber* is the port number of the remote device.
- *initString* is the initialization string that is to be sent to *ipAddress*.

See tspnet.connect() (on page 7-239) for additional detail.

To communicate to a remote ethernet device from the Models 707B and 708B:

1. Connect to the remote device using one of the previous procedures. If the Models 707B and 708B cannot make a connection to the remote device, it generates a timeout event. Use `tspnet.timeout` to set the timeout value. The default timeout value is 20 s.
2. Use `tspnet.write()` or `tspnet.execute()` to send strings to a remote device. If you use:
   - `tspnet.write()`: Strings are sent to the device exactly as indicated. You must supply any needed termination characters.
   - `tspnet.execute()`: The instrument appends termination characters to all strings that are sent. Use `tspnet.termination()` to specify the termination character.
3. To retrieve responses from the remote instrument, use `tspnet.read()`. The TSP instrument suspends operation until the remote device responds or a timeout event is generated. To check if data is available from the remote instrument, use `tspnet.readavailable()`.
4. Disconnect from the remote device using the `tspnet.disconnect()` function. Terminate all remote connections using `tspnet.reset()`.

Example script

The following example demonstrates how to connect to a remote device that is not enabled for TSP™, and send and receive data from this device:

```
-- Set tspnet timeout to 5 s.
tspnet.timeout = 5
--  Establish connection to another device with IP address 192.168.1.51
-- at port 1394.
id_instr = tspnet.connect("192.168.1.51", 1394, "*rst\r\n")
-- Print the device ID from connect string.
print("ID is: ", id_instr)
-- Set the termination character to CRLF. You must do this
-- for each connection after the connection has been made.
tspnet.termination(id_instr, tspnet.TERM_CRLF)
-- Send the command string to the connected device.
tspnet.write(id_instr, "login admin\r\n")
-- Read the available data, then print it.
tspnet.write(id_instr, "*idn?\r\n")
print("instrument write/read returns: ", tspnet.read(id_instr))
-- Disconnect all existing TSP-Net sessions.
tspnet.reset()
```

This example produces a return such as:

```
ID is:    1
instrument write/read returns:   SUCCESS: Logged in
instrument write/read returns:   KEITHLEY INSTRUMENTS,MODEL 707B,04089762,1.6.3d
```

TSP-Net compared to TSP-Link to communicate with TSP-enabled devices

The TSP-Link[TM] network interface is the preferred communications method for most applications where communications occurs between the Model 707B or 708B and another TSP-enabled instrument.

One of the advantages of using the TSP-Link network interface is that TSP-Link connections have three trigger lines that are available to each device on the TSP-Link network. You can use any one of the trigger lines to perform hardware triggering between devices on the TSP-Link network. Refer to Hardware trigger modes (on page 2-33) for details.

However, if the distance between the Model 707B or 708B and the TSP-enabled device is longer than 15 feet, use TSP-Net commands.

TSP-Net instrument commands: General device control

The following instrument commands provide general device control:

tspnet.clear() (on page 7-238)
tspnet.connect() (on page 7-239)
tspnet.disconnect() (on page 7-240)
tspnet.execute() (on page 7-241)
tspnet.idn() (on page 7-242)
tspnet.read() (on page 7-243)
tspnet.readavailable() (on page 7-244)
tspnet.reset() (on page 7-244)
tspnet.termination() (on page 7-245)
tspnet.timeout (on page 7-246)
tspnet.write() (on page 7-250)

TSP-Net instrument commands: TSP-enabled device control

The following instrument commands provide TSP-enabled device control:

tspnet.tsp.abort() (on page 7-246)
tspnet.tsp.abortonconnect (on page 7-247)
tspnet.tsp.rbtablecopy() (on page 7-248)
tspnet.tsp.runscript() (on page 7-249)

Example: Using tspnet commands

```
function telnetConnect(ipAddress, userName, password)
   -- Connect through telnet to a computer.
   id = tspnet.connect(ipAddress, 23, "")
   -- Read the title and login prompt from the computer.
   print(string.format("from computer--> (%s)", tspnet.read(id, "%n")))
   print(string.format("from computer--> (%s)", tspnet.read(id, "%s")))
   -- Send the login name.
   tspnet.write(id, userName .. "\r\n")
   -- Read the login echo and password prompt from the computer.
   print(string.format("from computer--> (%s)", tspnet.read(id, "%s")))
   -- Send the password information.
   tspnet.write(id, password .. "\r\n")
   -- Read the telnet banner from the computer.
   print(string.format("from computer--> (%s)", tspnet.read(id, "%n")))
   print(string.format("from computer--> (%s)", tspnet.read(id, "%n")))
   print(string.format("from computer--> (%s)", tspnet.read(id, "%n")))
   print(string.format("from computer--> (%s)", tspnet.read(id, "%n")))
end

function test_tspnet()
   tspnet.reset()
   -- Connect to a computer using telnet.
   telnetConnect("192.0.2.1", "my_username", "my_password")
   -- Read the prompt back from the computer.
   print(string.format("from computer--> (%s)", tspnet.read(id, "%n")))
   -- Change directory and read the prompt back from the computer.
   tspnet.write(id, "cd c:\\\r\n")
   print(string.format("from computer--> (%s)", tspnet.read(id, "%s")))
   -- Make a directory and read the prompt back from the computer.
   tspnet.write(id, "mkdir TEST_TSP\r\n")
   print(string.format("from computer--> (%s)", tspnet.read(id, "%s")))
   -- Change to the newly created directory.
   tspnet.write(id, "cd c:\\TEST_TSP\r\n")
   print(string.format("from computer--> (%s)", tspnet.read(id, "%s")))
   -- if you have data print it to the file.
   -- 11.2 is an example of data collected.
   cmd = "echo " .. string.format("%g", 11.2) .. " >> datafile.dat\r\n"
   tspnet.write(id, cmd)
   print(string.format("from computer--> (%s)", tspnet.read(id, "%s")))
   tspnet.disconnect(id)
end
test_tspnet()
```

Overview of instrument drivers

To use an instrument connected to a computer, you need to send instrument commands to control it. This can be tedious, as the programmer has to learn the low-level syntax and also deal with how their programming language or development environment interfaces to the remote communications interface driver or VISA I/O library. Keithley and most other test and measurement companies supply instrument drivers for their instruments.

To see what drivers are available for your instrument:

1. Go to the Product Support and Downloads web page (tek.com/support/product-support).
2. Enter the model number of your instrument.
3. Select **Software** from the list.
4. Select and download the file.

For LabVIEW™, you can also go to the NI website and search their instrument driver database.

All Keithley drivers come with examples written in several programming languages that show you how to do common tasks with the instruments. The examples are available in the drivers and through Test Script Builder.

Instrument driver types

There are several different styles of instrument drivers. Keithley provides the following instrument drivers for the 707B and 708B:

- A native LabVIEW driver
- An IVI-C driver
- An IVI-COM driver

You need to pick the style that best suits the application development environment that you are using. For example, if you are using LabVIEW, pick a native LabVIEW driver. If a native LabVIEW driver is not available, you can use an IVI-C driver because LabVIEW has the option of creating a wrapper for the IVI-C driver.

LabVIEW supports IVI-COM drivers, but they are not preferred. However, if they are the only driver types available for the instrument, they can be used.

If LabWindows/CVI or C/C++ is your programming language, an IVI-C driver is the best option. For Microsoft™ Visual Basic™ 6.0 and any .NET language (C#, VB.NET, and so on), an IVI-COM driver is the best option.

Sometimes instrument vendors do not provide all three driver types. Most languages can accommodate other driver types, but this is not optimal.

The following sections describe the different driver types in more detail.

VXIPnP drivers

VXI*plug&play* (VXIPnP) style drivers are Win32 DLLs that have some standard functions defined by the IVI Foundation (ivifoundation.org), such as:

- init
- close
- error_message
- reset
- self_test
- read
- initiate
- fetch
- abort

The application programming interface (API) was defined so that users of instruments have a familiar API from instrument to instrument. There are some basic guidelines when creating APIs for your instrument, such as using VISA data types and how to construct the CVI hierarchy.

Interchangeable Virtual Instruments (IVI) style drivers

The IVI Foundation (ivifoundation.org) defined a set of application programming interfaces (APIs) for instruments, including digital multimeters, arbitrary waveform/function generators, DC power supplies, AC power supplies, oscilloscopes, switches, spectrum analyzers, RF signal generators, and power meters.

There are two types of IVI drivers, IVI-COM and IVI-C. IVI-COM drivers use Microsoft^{TM} COM technology to expose driver functionality. IVI-C drivers use conventional Microsoft Windows^{TM} DLLs to export simple C-based functions.

For more information about IVI drivers and the differences between the COM, C, and .NET interfaces, see Understanding the Benefits of IVI (ivifoundation.org/docs/Understanding%20the%20Benefits%20of%20IVI[3].pdf).

LabVIEW drivers

Native LabVIEW drivers

A native LabVIEW™ driver is a LabVIEW driver that is created using entirely built-in LabVIEW VIs. It does not make any calls to external DLLs or Library files. This makes the driver portable to all the platforms and operating systems that LabVIEW and VISA supports (such as Linux™ on x86, Mac OS™ X, and Microsoft™ Windows™).

LabVIEW driver wrappers

All IVI-C drivers have a function panel file (file name extension `.fp`) that shows a hierarchy of the function calls into a DLL. It is a tool that guides a user to select the correct function call in the driver, because a DLL only has a flat API entry point scheme (unlike COM or .NET).

Any CVI-generated `.fp` files can be imported into LabVIEW and LabVIEW generates a wrapper for the DLL. The drawback here is that the driver is dependent on the DLL, which is not portable and is therefore specific to the Windows operating system.

Section 7

TSP command reference

In this section:

Command programming notes

Placeholder text

This manual uses italicized text to represent the parts of remote commands that must be replaced by user specified values. The following examples show typical uses of italicized text.

Example 1:

```
beeper.enable = state
```

Where *state* can be a value (`beeper.ON` or `beeper.OFF`) or an integer (`1` or `0`) that you specify. For example, to set this attribute on, you send one of the following commands:

```
beeper.enable = beeper.ON
```

```
beeper.enable = 1
```

Example 2:

```
digio.trigger[N].assert()
```

Where *N* is an integer (`1` to `14`) that you specify. For example, to assert trigger line 7, you send:

```
digio.trigger[7].assert()
```

To assert a trigger line with a variable as the integer, you send:

```
triggerline = 7
```

```
digio.trigger[triggerline].assert()
```

Example 3:

The attribute returns a string that contains information about a card in a specified slot and is shown as:

`slot[`*`slot`*`].idn`

For example, to get information about the card installed in slot 1, send:

`print(slot[1].idn)`

Syntax rules

Use these syntax requirements to build well-formed instrument control commands.

Instrument commands are case sensitive. Refer to the command reference descriptions for the correct case.

The white space in lists of parameters in functions is optional. For example, the following functions are equivalent:

```
digio.writebit(3,0)
digio.writebit (3, 0)
```

All functions must have a set of parentheses `()` immediately following the function, even if there are no parameters specified. For example:

```
waitcomplete(G)
timezone = localnode.gettimezone()
```

If there are multiple parameters, they must be separated by commas (**`,`**). For example:

```
beeper.beep(0.5, 2400)
```

If the parameter values are in a range, use a colon (**`:`**) to separate them. For example:

```
scan.create("1A01:1A10")
```

Time and date values

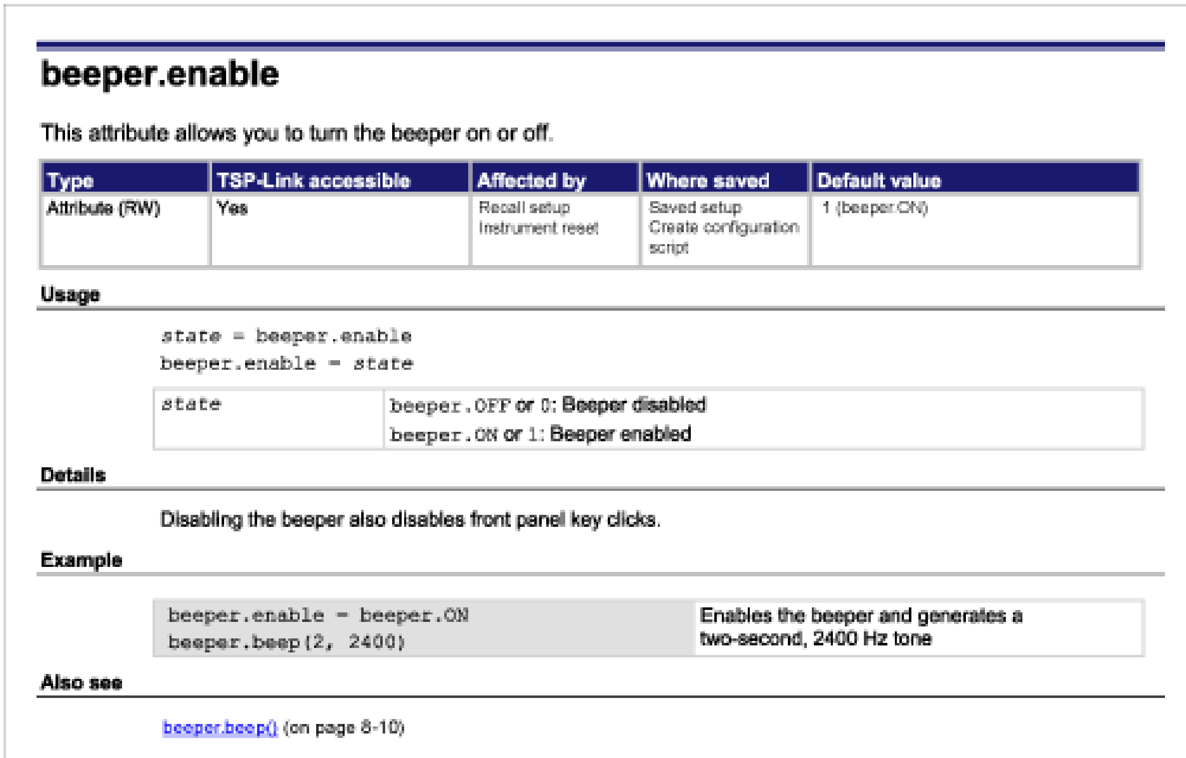
Time and date values are represented as the number of seconds since some base. There are three time bases:

- **UTC 12:00 am Jan 1, 1970.** Some examples of UTC time are reading buffer base timestamps, adjustment dates, and the value returned by `os.time()`.
- **Instrument on.** References time to when the instrument was turned on. The value returned by `os.clock()` is referenced to the turn-on time.
- **Event.** Time referenced to an event, such as the first reading stored in a reading buffer.

Use the TSP command reference

The Test Script Processor (TSP™) command reference contains detailed descriptions of each of the TSP commands that you can use to control your instrument. Each command description is broken into subsections. The following figure shows an example of a command description.

Figure 76: Example instrument command description



beeper.enable

This attribute allows you to turn the beeper on or off.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Recall setup Instrument reset	Saved setup Create configuration script	1 (beeper.ON)

Usage

```
state = beeper.enable
beeper.enable = state
```

state	`beeper.OFF` or `0`: Beeper disabled `beeper.ON` or `1`: Beeper enabled

Details

Disabling the beeper also disables front panel key clicks.

Example

`beeper.enable = beeper.ON` `beeper.beep(2, 2400)`	Enables the beeper and generates a two-second, 2400 Hz tone

Also see

beeper.beep() (on page 8-10)

The subsections contain information about the command. The subsections are:

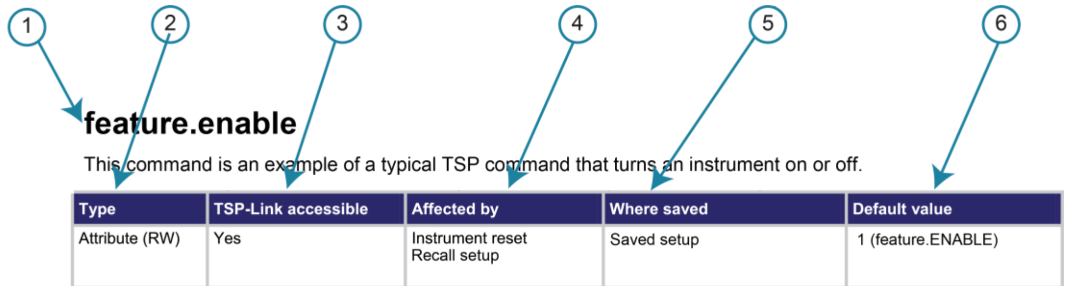
- Command name and summary table
- Usage
- Details
- Example
- Also see

The content of each of these subsections is described in the following topics.

Command name and summary table

Each instrument command description starts with the command name, followed by a brief description and a table with relevant information for each command. Definitions for the numbered items in the following figure are listed after the figure.

Figure 77: TSP command name and summary table



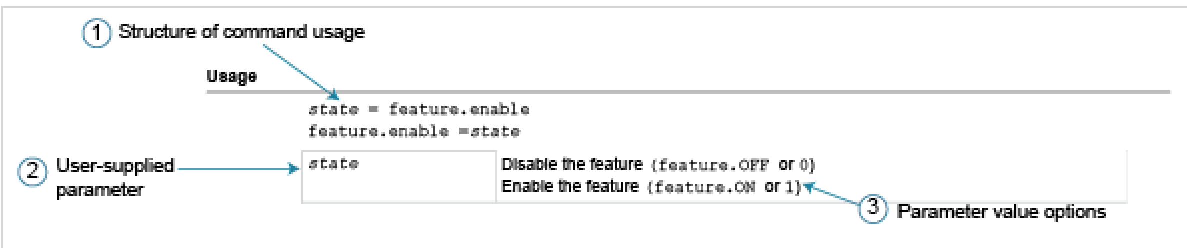
1. **Instrument command name**. Indicates the beginning of the command description. It is followed by a brief description of what the command does.
2. **Type of command**. Commands can be functions, attributes, or constants. If the command is an attribute, it can be read-only (R), read-write (RW), or write-only (W). For detail on commands, see Introduction to TSP operation (on page 5-1).
3. **TSP-Link accessible**. **Yes** or **No**; indicates whether or not the command can be accessed through a TSP-Link network.
4. **Affected by**. Commands or actions that may change the setting of this command.
   - **LAN restore defaults:** This command is reset to the default value when `lan.restoredefaults()` is sent.
   - **Digital I/O trigger N reset:** This command is reset to the default value when `digio.trigger[N].reset()` is sent.
   - **Recall setup:** This command is stored as part of the saved setup and is changed to the value stored in the saved setup when the setup is recalled.
   - **Instrument reset:** This command is reset to the default value when `reset()`, `localnode.reset()`, or `*RST` is sent.
   - **Power cycle:** This command is set to the default value when the instrument power is cycled.

5. **Where saved**. Indicates where the command settings reside once they are used on an instrument. Options include:
   - **Not saved:** Command is not saved anywhere and must be typed each time you use it.
   - **Nonvolatile memory:** Storage area in the instrument where information is saved when the instrument is turned off.
   - **Saved setup:** Command is saved as part of the saved setup.
6. **Default value:** Lists the default value or constant for the command. The parameter values are defined in the Usage or Details sections of the command description.

Command usage

The Usage section of the remote command listing shows how to properly structure the command. Each line in the Usage section is a separate variation of the command usage. All possible command usage options are shown.

Figure 78: TSP usage description



1 **Structure of command usage:** Shows how to organize the parts of the command. If a parameter is shown to the left of the command, it is the return when you print the command. Items to the right are the parameters or other items you need to enter when setting the command.

2 **User-supplied parameters:** Indicated by italics. For example, for the function `beeper.beep(duration, frequency)`, replace *duration* with the number of seconds and *frequency* with the frequency of the tone. Send `beeper.beep(2, 2400)` to generate a two-second, 2400 Hz tone.

Some commands have optional parameters. If there are optional parameters, they must be entered in the order presented in the Usage section. You cannot leave out any parameters that precede the optional parameter. Optional parameters are shown as separate lines in usage, presented in the required order with each valid permutation of the optional parameters.

For example:

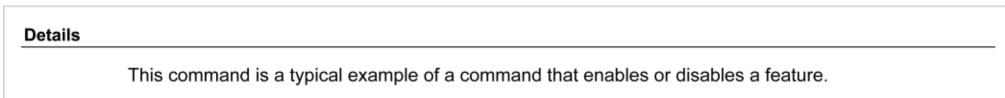
```
printbuffer(startIndex, endIndex, buffer1)
printbuffer(startIndex, endIndex, buffer1, buffer2)
```

3 **Parameter value options:** Descriptions of the options that are available for the user-defined parameter.

Command details

This section lists additional information you need to know to successfully use the remote command.

Figure 79: TSP Details description



Example section

The Example section of the remote command description shows examples of how you can use the command.

Figure 80: TSP example code

1. Actual example code that you can copy from this table and paste into your own programming application.
2. Description of the code and what it does. This may also contain example output of the code.

Related commands and information

The Also see section of the remote command description lists additional commands or sections that are related to the command.

Figure 81: TSP Also see description

TSP commands

The TSP commands available for the instrument are listed in alphabetical order.

beeper.beep()

This function generates an audible tone.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
beeper.beep(duration, frequency)
```

duration	The amount of time to play the tone (0.001 s to 100 s)
frequency	The frequency of the tone in hertz (Hz)

Details

You can use the beeper of the Models 707B and 708B to provide an audible signal at a specified frequency and time duration. For example, you can use the beeper to signal the end of a lengthy sweep.

The beeper does not sound if it is disabled. It can be disabled or enabled with the beeper enable command, or through the front panel.

Example

```
beeper.enable = beeper.ON
beeper.beep(2, 2400)
```

Enables the beeper and generates a two-second, 2400 Hz tone.

Also see

beeper.enable (on page 7-8)

beeper.enable

This command allows you to turn the beeper on or off.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Recall setup Instrument reset	Saved setup Create configuration script	1 (beeper.ON)

Usage

```
state = beeper.enable
beeper.enable = state
```

state	Disable the beeper: `beeper.OFF` or `0` Enable the beeper: `beeper.ON` or `1`

Details

This command enables or disables the beeper. When enabled, a beep signals that a front-panel key has been pressed. Disabling the beeper also disables front-panel key clicks.

Example

```
beeper.enable = beeper.ON
beeper.beep(2, 2400)
```

Enables the beeper and generates a two-second, 2400 Hz tone.

Also see

beeper.beep() (on page 7-7)

bit.bitand()

This function performs a bitwise logical AND operation on two numbers.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
result = bit.bitand(value1, value2)
```

result	Result of the logical AND operation
value1	Operand for the logical AND operation
value2	Operand for the logical AND operation

Details

Any fractional parts of `value1` and `value2` are truncated to form integers. The returned `result` is also an integer.

Example

```
testResult = bit.bitand(10, 9)
print(testResult)
```

Performs a logical AND operation on decimal 10 (binary 1010) with decimal 9 (binary 1001), which returns a value of decimal 8 (binary 1000).

Output:

```
8.0000000e+00
```

Also see

Bit manipulation and logic operations (on page 5-5)
bit.bitor() (on page 7-9)
bit.bitxor() (on page 7-10)

bit.bitor()

This function performs a bitwise logical OR operation on two numbers.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
result = bit.bitor(value1, value2)
```

result	Result of the logical OR operation
value1	Operand for the logical OR operation
value2	Operand for the logical OR operation

Details

Any fractional parts of `value1` and `value2` are truncated to make them integers. The returned `result` is also an integer.

Example

```
testResult = bit.bitor(10, 9)
print(testResult)
```

Performs a bitwise logical OR operation on decimal 10 (binary 1010) with decimal 9 (binary 1001), which returns a value of decimal 11 (binary 1011).

Output:

```
1.1000000e+01
```

Also see

Bit manipulation and logic operations (on page 5-5)
bit.bitand() (on page 7-8)
bit.bitxor() (on page 7-10)

bit.bitxor()

This function performs a bitwise logical XOR (exclusive OR) operation on two numbers.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`result = bit.bitxor(value1, value2)`

result	Result of the logical XOR operation
value1	Operand for the logical XOR operation
value2	Operand for the logical XOR operation

Details

Any fractional parts of *value1* and *value2* are truncated to make them integers. The returned *result* is also an integer.

Example

```
testResult = bit.bitxor(10, 9)
print(testResult)
```

Performs a logical `XOR` operation on decimal 10 (binary 1010) with decimal 9 (binary 1001), which returns a value of decimal 3 (binary 0011).
Output:
`3.0000000e+00`

Also see

Bit manipulation and logic operations (on page 5-5)
bit.bitand() (on page 7-8)
bit.bitor() (on page 7-9)

bit.clear()

This function clears a bit at a specified index position.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`result = bit.clear(value, index)`

result	Result of the bit manipulation
value	Specified number
index	One-based bit position within value to clear (1 to 32)

Details

Any fractional part of *value* is truncated to make it an integer. The returned *result* is also an integer.

The least significant bit of *value* is at *index* position 1; the most significant bit is at *index* position 32.

Example

```
testResult = bit.clear(15, 2)
print(testResult)
```

The binary equivalent of decimal 15 is 1111. If you clear the bit at *index* position 2, the returned decimal value is 13 (binary 1101).

Output:

```
1.3000000e+01
```

Also see

Bit manipulation and logic operations (on page 5-5)
bit.get() (on page 7-11)
bit.set() (on page 7-13)
bit.test() (on page 7-15)
bit.toggle() (on page 7-16)

bit.get()

This function retrieves the weighted value of a bit at a specified index position.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
result = bit.get(value, index)
```

result	Result of the bit manipulation
value	Specified number
index	One-based bit position within *value* to get (1 to 32)

Details

This function returns the value of the bit in *value* at *index*. This is the same as returning *value* with all other bits set to zero (0).

The least significant bit of *value* is at *index* position 1; the most significant bit is at *index* position 32.

If the indexed bit for the number is set to zero (0), the result is zero (0).

Example

```
testResult = bit.get(10, 4)
print(testResult)
```

The binary equivalent of decimal 10 is 1010. If you get the bit at index position 4, the returned decimal value is 8.

Output:

```
8.0000000e+00
```

Also see

Bit manipulation and logic operations (on page 5-5)
bit.clear() (on page 7-10)
bit.set() (on page 7-13)
bit.test() (on page 7-15)
bit.toggle() (on page 7-16)

bit.getfield()

This function returns a field of bits from the value starting at the specified index position.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

result = bit.getfield(*value*, *index*, *width*)

result	Result of the bit manipulation
value	Specified number
index	One-based bit position within *value* to get (1 to 32)
width	The number of bits to include in the field (1 to 32)

Details

A field of bits is a contiguous group of bits. This function retrieves a field of bits from *value* starting at *index*.

The *index* position is the least significant bit of the retrieved field. The number of bits to return is specified by *width*.

The least significant bit of *value* is at *index* position 1; the most significant bit is at *index* position 32.

Example

```
myResult = bit.getfield(13, 2, 3)
print(myResult)
```

The binary equivalent of decimal 13 is 1101.
The field at *index* position 2 and *width* 3 consists of the binary bits 110. The returned value is decimal 6 (binary 110).

Output:

```
6.0000000e+00
```

Also see

Bit manipulation and logic operations (on page 5-5)
bit.get() (on page 7-11)
bit.set() (on page 7-13)
bit.setfield() (on page 7-14)

bit.set()

This function sets a bit at the specified index position.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
result = bit.set(value, index)
```

result	Result of the bit manipulation
value	Specified number
index	One-based bit position within *value* to set (1 to 32)

Details

This function returns *result*, which is *value* with the indexed bit set. The *index* must be between 1 and 32.

The least significant bit of *value* is at *index* position 1; the most significant bit is at *index* position 32.

Any fractional part of *value* is truncated to make it an integer.

Example

```
testResult = bit.set(8, 3)

print(testResult)
```

The binary equivalent of decimal 8 is 1000. If the bit at *index* position 3 is set to 1, the returned value is decimal 12 (binary 1100).
Output:
`1.2000000e+01`

Also see

Bit manipulation and logic operations (on page 5-5)
bit.clear() (on page 7-10)
bit.get() (on page 7-11)
bit.getfield() (on page 7-12)
bit.setfield() (on page 7-14)
bit.test() (on page 7-15)
bit.toggle() (on page 7-16)

bit.setfield()

This function overwrites a bit field at a specified index position.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
result = bit.setfield(value, index, width, fieldValue)
```

result	Result of the bit manipulation
value	Specified number
index	One-based bit position in *value* to set (1 to 32)
width	The number of bits to include in the field (1 to 32)
fieldValue	Value to write to the field

Details

This function returns *result*, which is *value* with a field of bits overwritten, starting at *index*. The *index* specifies the position of the least significant bit of *value*. The *width* bits starting at *index* are set to *fieldValue*.

The least significant bit of *value* is at *index* position 1; the most significant bit is at *index* position 32.

Before setting the field of bits, any fractional parts of *value* and *fieldValue* are truncated to form integers.

If *fieldValue* is wider than *width*, the most significant bits of the *fieldValue* that exceed the width are truncated. For example, if *width* is 4 bits and the binary value for *fieldValue* is 11110 (5 bits), the most significant bit of *fieldValue* is truncated and a binary value of 1110 is used.

Example

```
testResult = bit.setfield(15, 2, 3, 5)
print(testResult)
```

The binary equivalent of decimal 15 is 1111. After overwriting it with a decimal 5 (binary 101) at *index* position 2, the returned `value` is decimal 11 (binary 1011).

Output:

```
1.1000000e+01
```

Also see

Bit manipulation and logic operations (on page 5-5)
bit.get() (on page 7-11)
bit.set() (on page 7-13)
bit.getfield() (on page 7-12)

bit.test()

This function returns the Boolean value (`true` or `false`) of a bit at the specified index position.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`result = bit.test(value, index)`

result	Result of the bit manipulation
value	Specified number
index	One-based bit position within value to test (1 to 32)

Details

This function returns *result*, which is the result of the tested bit.

The least significant bit of *value* is at *index* position 1; the most significant bit is at *index* position 32.

If the indexed bit for *value* is 0, *result* is `false`. If the bit of *value* at *index* is 1, the returned value is `true`.

If *index* is bigger than the number of bits in *value*, the result is `false`.

Example

```
testResult = bit.test(10, 4)
print(testResult)
```

The binary equivalent of decimal 10 is 1010. Testing the bit at *index* position 4 returns a Boolean value of `true`.

Output:

`true`

Also see

Bit manipulation and logic operations (on page 5-5)
bit.clear() (on page 7-10)
bit.get() (on page 7-11)
bit.set() (on page 7-13)
bit.toggle() (on page 7-16)

bit.toggle()

This function toggles the value of a bit at a specified index position.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
result = bit.toggle(value, index)
```

result	Result of the bit manipulation
value	Specified number
index	One-based bit position within *value* to toggle (1 to 32)

Details

This function returns *result*, which is the result of toggling the bit *index* in *value*.

Any fractional part of *value* is truncated to make it an integer. The returned value is also an integer.

The least significant bit of *value* is at *index* position 1; the most significant bit is at *index* position 32.

The indexed bit for *value* is toggled from 0 to 1, or 1 to 0.

Example

```
testResult = bit.toggle(10, 3)
print(testResult)
```

The binary equivalent of decimal 10 is 1010. Toggling the bit at *index* position 3 returns a decimal value of 14 (binary 1110).
Output:
`1.4000000e+01`

Also see

Bit manipulation and logic operations (on page 5-5)
bit.clear() (on page 7-10)
bit.get() (on page 7-11)
bit.set() (on page 7-13)
bit.test() (on page 7-15)

channel.clearforbidden()

This function clears the list of channels specified from being forbidden to close.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
channel.clearforbidden("channelList")
```

channelList	The channels to set, using standard channel naming

Details

The *channelList* parameter indicates the channels that will no longer be forbidden to close, and may include:

- `allslots` or `slotX` (where *X* is `1` to `6` for Model 707B, or 1 for Model 708B)
- Channel ranges or individual channels

This function allows all items in the *channelList* parameter to be closed. It removes the forbidden to close attribute that can be applied to a channel using `channel.setforbidden()`.

Command processing stops as soon as an error is detected. If an error is found, the channels are not cleared from being forbidden to close.

Example

```
channel.reset("slot1")
channel.setforbidden("1A01:1A05")

channel.clearforbidden("1A02,1A03")
print(channel.getforbidden("slot1"))
```

Reset the channels on slot 1.
Set channels 1A01, 1A02, 1A03, 1A04, and 1A05 to be forbidden to close.
Change 1A02 and 1A03 to be allowed to close.
Retrieve the list of forbidden channels.
Output:
`1A01,1A04,1A05`

Also see

channel.getforbidden() (on page 7-29)
channel.setforbidden() (on page 7-44)

channel.close()

This function closes the channels and channel patterns that are specified by the channel list parameter.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
channel.close("channelList")
```

channelList	The channels and channel patterns to close, using standard channel naming

Details

Channels closed with this command are appended to the already closed channels (no previously closed channels are opened by this command).

Actions associated with this function include:

- Close the specified items in *channelList*
- Incur the settling time and any user-specified delay

An error is generated if:

- The parameter string contains slot*X*, where *X* is 1 to 6, or allslots
- A forbidden item is specified
- Specified channel does not support being closed

Once an error is detected, the command stops processing and no channels are closed.

Example

```
channel.open("allslots")
channel.pattern.setimage("1B02,1B04,1B06", "Chans")
channel.close("1A01:1A05, 1C03, Chans")
print(channel.getclose("slot1"))
```

Close a variety of channels, directly and with a channel pattern; note that the output sorts the channels
Output:
`1A01;1A02;1A03;1A04;1A05;1B02;1B04;1B06;1C03`

Also see

channel.exclusiveclose() (on page 7-23)
channel.exclusiveslotclose() (on page 7-24)
channel.getclose() (on page 7-26)
channel.open() (on page 7-35)

channel.connectrule

This attribute controls the connection rule for closing and opening channels in the instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup	Create configuration script Save setup	channel.BREAK_BEFORE_MAKE

Usage

```
rule = channel.connectrule
channel.connectrule = rule
```

rule	▪ `channel.BREAK_BEFORE_MAKE` or `1`: Break before make (BBM) connections for relays in the instrument ▪ `channel.MAKE_BEFORE_BREAK` or `2`: Make before break (MBB) connections for relays in the instrument ▪ `channel.OFF` or `0`: Does not guarantee a connection rule; the instrument closes relays as efficiently as possible to improve speed performance without applying a rule

Details

The connection rule describes the order in which switch channels are opened and closed when using `channel.exclusiveclose()`, `channel.exclusiveslotclose()`, and scanning commands such as `scan.execute()` and `scan.background()`. These commands may both open and close switch channels in a single command. The connection rule dictates the algorithm used by the instrument to order the opening and closing of switches.

The connection rule affects the operating time of these commands. These commands do not allow the instrument to continue execution until the settle time of the relays has expired.

When the connection rule is set to break before make, the instrument ensures that all switch channels open before any switch channels close.

When the connection rule is set to make before break, the instrument ensures that all switch channels close before any switch channels open. This behavior should be applied with caution because it will connect two test devices together for the duration of the switch close settle time.

For break before make and make before break, when switch channels are both opened and closed, this command executes not less than the addition of both the open and close settle times of the indicated switch channels.

With no connection rule (set to `channel.OFF`), the instrument attempts to simultaneously open and close switch channels in order to minimize the command execution time. This results in faster performance at the expense of guaranteed switch position. During the operation, multiple switch channels may simultaneously be in the close position. Make sure your device under test can withstand this possible condition. When switch channels are both opened and closed, the command executes not less than the greater of either the open or close settle times of the indicated switch channels.

NOTE

You cannot guarantee the sequence of open and closure operations when the channel connect rule is set to `OFF`. It is highly recommended that you implement cold switching when the channel connect rule is set to `OFF`.

In general, the settling time of single commands that open and close switch channels depends on several factors, such as card type and channel numbers. However, the opening and closing of two sequential channels including no others can be guaranteed as follows:

- Break before make: Open settle time + close settle time
- Make before break: Close settle time + open settle time
- `channel.OFF`: Maximum of open settle time or close settle time

CAUTION

This behavior is also affected by channel.connectsequential and any additional user delay times.

Make before break (also known as hot switching) can dry-weld reed relays so that they will always be on. Hot switching is recommended only when external protection is provided.

Example

`channel.connectrule = channel.BREAK_BEFORE_MAKE`
Sets the connect rule in the instrument to `channel.BREAK_BEFORE_MAKE`

Also see

channel.connectsequential (on page 7-21)
channel.exclusiveclose() (on page 7-23)
channel.exclusiveslotclose() (on page 7-24)
scan.background() (on page 7-146)
scan.execute() (on page 7-149)

channel.connectsequential

This attribute controls whether or not channels are closed sequentially.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup	Create configuration script	channel.OFF

Usage

```
sequential = channel.connectsequential
channel.connectsequential = sequential
```

sequential	Disable sequential connections: `channel.OFF` or `0` Enable sequential connections: `channel.ON` or `1`

Details

When `channel.connectsequential` is enabled, the list of channel actions is acted on sequentially. No two relays are opened or closed simultaneously.

Using a sequential close allows you to determine the time for a close operation to happen. For example, if you close three channels and each takes 4 ms to close (assuming no additional user delay times), with sequential on, it will take 12 ms. With sequential off, it may be 4, 8 or 12 ms, depending on whether or not the card can close multiple channels at once.

The order in which channels are opened or closed is not guaranteed with sequential off.

The sequential setting affects all channels in the instrument.

Example

`channel.connectsequential = channel.ON`
Specifies that channels close sequentially.

Also see

channel.connectrule (on page 7-19)
Switch and scan (on page 2-1)

channel.createspecifier()

This function creates a string channel descriptor from a series of card-dependent integer arguments.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`specifier = channel.createspecifier(slotX, row, column)`

specifier	The formatted string for the channel
X	The slot number (1 to 6)
row	The row number to use (see the matrix card documentation for limits); map row letters to numbers (A = 1, B = 2, and so on)
column	Specifies the column number to use

Details

The arguments depend on the card type in the specified slot. This command can only create valid channel descriptors; if an illegal argument is sent for the type of card in the specified slot, an error is generated.

Example 1

```
cs = channel.createspecifier(1, 1, 1)
print(cs)
```

Creates a channel descriptor for row 1, column 1 on the card in slot 1.
Output:
`1A01`

Example 2

```
count = 0
for row = 1, 8 do
   for col = 1, 12 do
      ch = channel.createspecifier(1, row, col)
      count = count + tonumber(channel.getcount(ch))
   end
end
print("Count is " .. count .. ".")
```

Assuming an 8×12 matrix card in slot 1, this example calculates the sum of the counts on all channels.
Output:
`Count is 1060656.`

Also see

None

channel.exclusiveclose()

This function closes the specified channels so that they are the only channels that are closed on the instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`channel.exclusiveclose("channelList")`

channelList	A string listing the channels and channel patterns to exclusively close

Details

This command allows you to close specific channels and open any other channels on the instrument.

When you send this command:

- Any presently closed channels that are not specified in *channelList* are opened.
- The channels in *channelList* are closed.
- Settling and user-specified delay times are applied as defined by the connection rules and delay settings.

If the *channelList* parameter is an empty string or a string of spaces, all channels are opened. Therefore, sending `channel.exclusiveclose("")` is equivalent to `channel.open(channel.getclose("allslots"))`. However, sending the equivalent commands when nothing is closed generates an error because `nil` (the response of `channel.getclose("allslots")`) is being sent to the open command.

An error is generated if:

- The parameter string contains `slotX`, where *X* = 1 to 6 or `allslots`
- A specified channel or channel pattern is invalid
- Channel number does not exist for slot specified
- Slot is empty
- A forbidden item is specified

Once an error is detected, the command stops processing. Channels open or close only if no errors are found.

Example

```
channel.exclusiveclose("")
channel.close("1D01,1E12")
print(channel.getclose("slot1"))

channel.pattern.setimage("1B02,1B04,1B06", "myChans")
channel.exclusiveclose("1A01:1A05, 1C03, myChans")
print(channel.getclose("slot1"))
```

Open all channels because the `channelList` parameter is empty.
Close `1D01` and `1E12`.
Output:
`1D01;1E12`

Create the pattern `myChans`.
Exclusively close the channels in `myChans` and additional specified channels. The originally closed channels (`1D01` and `1E12`) are now open.
Output:
`1A01;1A02;1A03;1A04;1A05;1B02;1B04;1B06;1C03`

Also see

channel.close() (on page 7-18)
channel.connectrule (on page 7-19)
channel.connectsequential (on page 7-21)
channel.exclusiveslotclose() (on page 7-24)
channel.getclose() (on page 7-26)
channel.open() (on page 7-35)

channel.exclusiveslotclose()

This function closes the specified channels and channel patterns on the associated slots and opens any channels that are not specified.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
channel.exclusiveslotclose("channelList")
```

channelList	A string that lists the channels and channel patterns to exclusively close on the cards in associated slots

Details

This command allows you to bundle the closing of channels with the opening of channels. Any presently closed channels open if they are not specified to be closed on the slots related to the channels in *channelList*. Using this command guarantees that only the specified channels and channel patterns are closed on the slots associated with channels in the *channelList*.

When this command is sent:

- Closed channels for the associated slots are opened if they are not specified in the *channelList*
- Channels specified by the items in *channelList* are closed

Any settling times and user-specified delay times are incurred before command processing is complete

For example, if row 1, column 1 channels are closed on each of the six slots, specifying a *channelList* parameter of "2A02, 4A04" opens the row 1, column 1 channels (slots 2 and 4 only). Then, the row 1, column 2 channel on slot 2, and the row 1, column 4 channel on slot 4 close. The row 1, column 2 channels remain closed on slots 1, 3, 5, and 6.

An error is generated if:

- The parameter string contains slot*X* (where *X* = 1 to 6) or allslots
- The parameter string is empty or parameter string with just spaces
- A specified channel is invalid or does not exist for the slot
- Channel pattern does not exist or the image of the pattern is an empty channel list
- A forbidden item is specified

Once an error is detected, the command stops processing. Channels open or close only if no errors are found and remain unchanged with any parsing or syntax error.

Example

```
channel.open("allslots")
channel.close("1A01,2A01,3A01,4A01,5A01,6A01")
channel.exclusiveslotclose("3A03")
print(channel.getclose("allslots"))
```

Open all channels on all slots.
Close row A column 1 channels on all slots.
Open row A column 1 on slot 3 and close row A column 3 on slot 3 without affecting any other slot with closed channels.
Output:
1A01;2A01;3A03;4A01;5A01;6A01

Also see

channel.close() (on page 7-18)
channel.exclusiveclose() (on page 7-23)
channel.getclose() (on page 7-26)
channel.open() (on page 7-35)

channel.getclose()

This function queries for the closed channels indicated by the channel list parameter.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`closed = channel.getclose("channelList")`

closed	A string listing the channels that are presently closed in the specified channel list parameter
channelList	A string representing the channels and channel patterns that will be queried

Details

Use this command to return a list of closed measurement channels, including the paired channel for 4-wire measurements. It does not return nonmeasurement channels.

If more than one channel is closed, they are semicolon-delimited in the string. When the *channelList* contains a channel pattern, only the closed channels in that image are returned.

You can use `"allslots"` to query for all channels closed. The instrument returns only information for valid slots and channels.

An error message is generated if an empty parameter string is specified or if the specified channel list contains no valid channels that can be closed.

If none of the channels in the channel list is closed, `nil` is returned.

Example 1

```
channelList = "1A01:1H12"
channel.close("1A01")
print(channel.getclose(channelList))
channel.close("1C03")
print(channel.getclose(channelList))
```

For this example, assume there is a card or pseudocard in slot 1 with no previously closed channels. The output is:

```
1A01
1A01;1C03
```

Example 2

```
channel.close("1B03:1B05")
print(channel.getclose("allslots"))
```

For this example, assume there is a card or pseudocard in slot 1 with no previously closed channels. The output is:

```
1B03;1B04;1B05
```

Also see

channel.close() (on page 7-18)
channel.getstate() (on page 7-33)
channel.open() (on page 7-35)
Data retrieval commands (on page 5-4)

channel.getcount()

This function returns the number of times the relays have been closed for the specified channels.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`counts = channel.getcount("channelList")`

counts	A comma-delimited string that lists the number of times the channel has closed
channelList	A string listing the items to query, which can include: ■ Channels ■ Channel patterns (channels are listed in the order in which they are listed in the pattern) ■ `slotX`, where *X* is `1` to `6` for the 707B or `1` for the 708B ■ `allslots`

Details

The Models 707B and 708B keeps an internal count of the number of times each relay has been closed. This count can help you determine when relays require replacement. Refer to the matrix card documentation for the contact life specifications for the relays.

If channels are specified, the count values are returned in the order in which the channels are specified. If slots are specified, the response lists the channels starting from lowest to highest. Because each slot is processed completely before going to the next, all slot 1 channels are listed before slot 2 channels.

If *channelList* includes a pattern, you can use `channel.pattern.getimage()` with the pattern name to see the channel order and the channels to which the close counts pertain.

The counts reported for the following cards indicate the number of closures since the last power cycle of the card:

- 7072
- 7072-HV
- 7173-50
- 7174A

For all other cards, the number of closures is the closures that have occurred over the lifetime of the card.

Relay closures are counted only when a relay cycles from open to closed state.

It is good practice to get the relay count at the end of a program. This saves the latest count to memory.

Example

```
channel.pattern.setimage("1A01,1B02,1C01","Path")
PathList = channel.pattern.getimage("Path")
print(PathList)
print(channel.getcount(PathList))
print(channel.getcount("Path"))
```

Gets the close counts for channels in a channel pattern called `"Path"`.
Sample output:

```
1A01,1B02,1C01
11001,11014,11025
11001,11014,11025
```

Also see

channel.pattern.getimage() (on page 7-37)
channel.pattern.setimage() (on page 7-38)
Data retrieval commands (on page 5-4)

channel.getdelay()

This function queries for the additional delay time for the specified channels.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`delayTimes = channel.getdelay("channelList")`

delayTimes	A comma-delimited string consisting of the delay times (in seconds) for channels specified in *channelList*
channelList	A string listing the channels to query for their delay times

Details

The *channelList* parameter may contain `slotX` (where *X* equals 1 to 6 for Model 707B, or 1 for Model 708B) or `allslots`.

A command, after closing the state of channels, incurs the delay time indicated in the response for a channel before it completes. However, the internal settling time must elapse before the user delay is incurred. Therefore, the sequence is:

1. Command is processed
2. Channel is closed
3. Settling time is incurred
4. User delay is incurred
5. Command completes

The delay times are comma-delimited in the same order that the items were specified in the *channelList* parameter. A value of zero (0) indicates that no additional delay time is incurred before a close command completes.

An error message is generated for the following reasons:

- The specified channels do not support a delay time
- A channel pattern is specified

Command processing stops as soon as an error is detected and a `nil` response is generated.

NOTE

Pseudocards do not support user delays, so this value is always zero (0) if a pseudocard is used.

Example

```
print(channel.getdelay("1A07,1B05,1C03"))
channel.setdelay("slot1", 3.1)
DelayTimes = channel.getdelay("1A07,1B05,1C03")
print(DelayTimes)
```

Get the existing delays for the listed channels.
Output:
`0.0000000e+00,0.0000000e+00,0.0000000e+00`
Set a delay on all channels in slot 1.
Verify that the delay was set for the listed channels.
Output:
`3.1000000e+00,3.1000000e+00,3.1000000e+00`

Also see

channel.setdelay() (on page 7-43)
Data retrieval commands (on page 5-4)

channel.getforbidden()

This function returns a string listing the channels in the channel list that are forbidden to close.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Recall setup	Create configuration script Save setup	Permitted to close

Usage

```
forbiddenList = channel.getforbidden("channelList")
```

forbiddenList	Comma-delimited string listing the channels in the channel list that are forbidden to close
channelList	A string listing the channels and channel patterns that are to be checked to see if they are forbidden to close

Details

The *channelList* parameter indicates which channels to check, and may include:

- `allslots` or `slotX` (where *X* equals 1 to 6 for Model 707B, or 1 for Model 708B)
- Channel ranges or individual channels
- Channel patterns

If there are no channels in the scope of the *channelList* that are on the forbidden list, the string returned is empty or `nil`. The format of the channels in the response string is slot, row, column for matrix channels.

Example

```
channel.reset("slot1")
channel.setforbidden("1A01:1A05")
print(channel.getforbidden("allslots"))
print(channel.getforbidden("slot1"))
print(channel.getforbidden("1A01:1A03,1B04,1B08,1B12"))
```

Reset the channels.
Set channels 1A01, 1A02, 1A03, 1A04, and 1A05 to be forbidden.
List the forbidden channels on all slots, slot 1, and list of channels.
Output:

```
1A01,1A02,1A03,1A04,1A05
1A01,1A02,1A03,1A04,1A05
1A01,1A02,1A03
```

Also see

channel.clearforbidden() (on page 7-17)
channel.setforbidden() (on page 7-44)
Data retrieval commands (on page 5-4)

channel.getlabel()

This function retrieves the label associated with one or more channels.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Recall setup	Create configuration script Save setup	slot, row, column identifier

Usage

`label = channel.getlabel("channelList")`

label	A string listing the comma-delimited labels for items in *channelList*
channelList	A string listing the channels to query for the label associated with them

Details

The *channelList* parameter can contain more than one channel. If it does, a comma delimits the labels for the channels. The return string lists the labels in the same order that the channels were specified. The *channelList* parameter cannot be an empty string and must be a valid channel.

The *channelList* parameter can contain `slotX` (where *X* equals 1 to 6 for Model 707B, or 1 for Model 708B) or `allslots`. In this case, the channels are listed before the analog backplane relays.

An error is generated if:

- The slot is empty
- The specified channel is not on the installed card
- A channel pattern is specified

Command processing stops as soon as an error is detected, and then a `nil` response is generated. No partial list of labels is returned.

Example

```
channel.reset("1A01")
print(channel.getlabel("1A01"))
channel.setlabel("1A01", "Device")
print(channel.getlabel("1A01"))
```

Reset the channel.
Print the default label of the channel.
Set the label to `Device`.
Return the new label.
Output:
```
1A01
Device
```

Also see

channel.setlabel() (on page 7-45)
Data retrieval commands (on page 5-4)

channel.getlabelcolumn()

This function retrieves the label that was assigned to a column.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset	Create configuration script	Empty string or blank

Usage

`label = channel.getlabelcolumn("channelList")`

label	A string that lists the comma-delimited labels for the items in *channelList*
channelList	A string that lists the channels to query for the labels associated with them

Details

The *channelList* parameter can contain more than one channel. If it does, a comma delimits the labels for the channels. The return string lists the labels in the same order that the channels were specified. The *channelList* parameter cannot be an empty string and must be a valid channel.

The *channelList* parameter can contain `slotX` (where *X* equals 1 to 6 for Model 707B, or 1 for Model 708B) or `allslots`. It can also contain a label. However, if the label exists, it is in the returned response and not the numeric channel number.

You cannot specify a channel pattern.

Example

```
channel.setlabelcolumn("1A01", "DUT1")
channel.setlabelcolumn("1A03", "DUT2")
print(channel.getlabelcolumn("1a01:1a12"))
```

Label all the column labels on a card.
Output:
`DUT1,,DUT2,,,,,,,,,`
Also note the change on the display.

Also see

channel.getlabelrow() (on page 7-32)
channel.setlabelcolumn() (on page 7-46)
channel.setlabelrow() (on page 7-48)

channel.getlabelrow()

This function retrieves the label assigned to a row.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset	Create configuration script	Empty string or blank

Usage

label = `channel.getlabelrow(`*channelList*`)`

label	A string that lists the comma-delimited labels for the items in *channelList*
channelList	A string that lists the channels to query for the labels associated with them

Details

The *channelList* parameter can contain more than one channel. If it does, a comma delimits the labels for the channels. The return string lists the labels in the same order that the channels were specified. The *channelList* parameter cannot be an empty string and must be a valid channel.

The *channelList* parameter can contain `slotX` (where *X* equals 1 to 6 for Model 707B, or 1 for Model 708B) or `allslots`. It can also contain a label, but it cannot contain a channel pattern.

Example

```
channel.setrowlabel("1A01", "DUT1")
channel.setrowlabel("1C01", "DUT2")
print(channel.getlabelrow("1A01, 1B01, 1C01, 1D01, 1E01, 1F01, 1G01, 1H01"))
```

Label the row labels on a card.
Output:
`DUT1,,DUT2,,,,,,`
Also note the change on the display.

Also see

channel.getlabelcolumn() (on page 7-31)
channel.setlabelcolumn() (on page 7-46)
channel.setlabelrow() (on page 7-48)

channel.getstate()

This function returns the state indicators of the channels in the instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Recall setup	Not saved	0

Usage

```
state = channel.getstate("channelList")
state = channel.getstate("channelList", indicatorMask)
```

state	Return string listing the comma-delimited states for the channels in *channelList*
channelList	String specifying the channels to query, using normal channel list syntax
indicatorMask	Value to specify only certain indicators; if omitted, all indicators are returned

Details

Each bit in the *state* represents a different indicator. Therefore, multiple indicators can be present (the OR operation is performed bitwise). The optional state *indicatorMask* can be used to return only certain indicators. If there is no *indicatorMask*, then all indicators are returned.

Indicators can be latched or unlatched, depending on other system settings. Latched indicators mean that the condition has occurred since the last reset command (or power cycle). Unlatched indicators mean that the condition occurred when the `channel.getstate()` command was issued.

Although the `channel.getstate()` command returns a string representing a number, this can be changed to a number and then compared to one of the provided Lua constants.

The only state information is an indicator of relay state (`channel.IND_CLOSED`).

Example

```
print(channel.getstate("4A01:4B08"))

channel.pattern.setimage("1A01,2B02,3C03", "Path")
print(channel.getstate("Path"))
print(channel.getstate("3C03"))

-- Unmasking the return value must be done
-- one channel at a time.
if bit.bitand(channel.IND_CLOSED,
   tonumber(channel.getstate("4A10"))) == 1 then
   print("CLOSED")
else
   print("OPENED")
end
```

Query the state of the first 20 channels on slot 4.
See the state of channels in channel pattern called `Path`.
The `channel.IND_CLOSED` command equates to the number 1. Because the state is a bit-oriented value, you must perform a logical AND operation on the state to the overload constant to isolate it from other indicators.
The `tonumber()` command only works with a single channel. When multiple channels are returned (for example, `channel.getstate("slot4")`), this string must be parsed by the comma delimiter to find each value.

Also see

channel.getclose() (on page 7-26)

channel.gettype()

This function returns the type associated with a channel.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`type = channel.gettype("channelList")`

type	Returns a comma-delimited list of the type of channels in *channelList*
channelList	String specifying the channels to query, using normal *channelList* syntax

Details

The channel type is defined by the physical hardware of the card on which the channel exists. The only valid channel type for the 707B and 708B is `channel.TYPE_SWITCH` or `1`.

Example

```
print(channel.gettype("1A01"))
```

Query the channel type of row 1, column 1, in slot 1.

Also see

None

channel.open()

This function opens the specified channels and channel patterns.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
channel.open("channelList")
```

channelList	String listing the channels and channel patterns to open

Details

This function opens the specified channels based on the switching configuration of the channel.

The settling time associated with a channel must elapse before the command completes. User delay is not added when a relay opens.

Example 1

`channel.open("1A01:1A05, 3B03, Chans")`
Opens channels in row 1, columns 01 to 05 on slot 1; row 2, column 03 on slot 3; and the channels in the channel pattern `Chans`.

Example 2

`channel.open("slot3, slot5")`
Opens all channels on slots 3 and 5.

Example 3

`channel.open("allslots")`
Opens all channels on all slots.

Also see

channel.close() (on page 7-18)
channel.exclusiveclose() (on page 7-23)
channel.exclusiveslotclose() (on page 7-24)
channel.getclose() (on page 7-26)
channel.getdelay() (on page 7-28)
channel.pattern.getimage() (on page 7-37)
channel.pattern.setimage() (on page 7-38)
channel.pattern.snapshot() (on page 7-40)
channel.getstate() (on page 7-33)
channel.setdelay() (on page 7-43)
channel.setforbidden() (on page 7-44)

channel.pattern.catalog()

This function creates a list of the user-created channel patterns.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
for name in channel.pattern.catalog() do
   ...
end
```

name	The user-defined name of the channel pattern that is assigned by the catalog function during the `for` loop

Details

This function allows you to print or delete all user-created channel patterns in the runtime environment. The entries that are returned are listed in random order.

Example

```
channel.pattern.setimage("1A01,1A02", "patternA")
channel.pattern.setimage("1B01,1B02", "patternB")
channel.pattern.setimage("1C01,1C02", "patternC")

for name in channel.pattern.catalog() do
   print(name .. " = " ..
      channel.pattern.getimage(name))
   channel.pattern.delete(name)
end
```

This example prints the names and items associated with all user-created channel patterns. It then deletes the channel pattern.

```
patternC = 1C01,1C02
patternA = 1A01,1A02
patternB = 1B01,1B02
```

Also see

channel.pattern.delete() (on page 7-37)
channel.pattern.getimage() (on page 7-37)
channel.pattern.setimage() (on page 7-38)
channel.pattern.snapshot() (on page 7-40)

channel.pattern.delete()

This function deletes a channel pattern.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
channel.pattern.delete("name")
```

name	A string representing the name of the channel pattern to delete

Details

An error is generated if the name does not exist as a channel pattern.

Example

`channel.pattern.delete("Channels")`
Deletes a channel pattern called `Channels`.

Also see

channel.pattern.catalog() (on page 7-36)
channel.pattern.getimage() (on page 7-37)
channel.pattern.setimage() (on page 7-38)
channel.pattern.snapshot() (on page 7-40)

channel.pattern.getimage()

This function queries a channel pattern for associated channels.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Recall setup Pole setting change	Create configuration script Save setup	Not applicable

Usage

```
channelList = channel.pattern.getimage("name")
```

channelList	A string specifying a list of channels that are represented by the name
name	A string representing the name of the channel pattern to query

Details

The returned string lists the channels in the slot, row, column format, even if a channel pattern was used to create it. Results for multiple channel patterns are delimited by a semicolon (`;`). Commas delimit the specific channels in a single channel pattern in the string.

If you change a pole setting for a channel that is associated with a channel pattern, the channel pattern is deleted. Be sure to configure the pole settings for channels (`channel.setpole`) before creating a channel pattern.

Example

```
channel.pattern.setimage("1A01:1A05", "myPattern")
channel.pattern.setimage("1B01,1B03,1B05", "myRoute")

myImage = channel.pattern.getimage("myPattern")
print(myImage)
print(channel.pattern.getimage("myRoute"))
print(channel.pattern.getimage("myRoute, myPattern"))
```

Using a Model 7174 (or similar model) card in slot 1, this example creates two channel patterns and then queries these patterns.
Output:

```
1A01,1A02,1A03,1A04,1A05
1B01,1B03,1B05
1B01,1B03,1B05;1A01,1A02,1A03,1A04,1A05
```

Also see

channel.pattern.catalog() (on page 7-36)
channel.pattern.delete() (on page 7-37)
channel.pattern.setimage() (on page 7-38)

channel.pattern.setimage()

This function creates a channel pattern and associates it with the specified name.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Recall setup Pole setting change	Create configuration script Save setup	Not applicable

Usage

`channel.pattern.setimage("`*`channelList`*`", "`*`name`*`")`

channelList	A string listing the channels and channel patterns to use when creating the new channel pattern
name	A string representing the name to associate with the new channel pattern

Details

If *name* is used for an existing channel pattern, that pattern is overwritten with the new pattern channel image (if no errors occur). The previous image associated with the name is lost.

The channel pattern is not created if an error is detected. You can create a channel pattern with an empty *channelList* parameter. Once a channel pattern is created, the only way to add a channel to an existing pattern is to delete the old pattern and recreate the pattern with the new items.

If you change a pole setting for a channel that is associated with a channel pattern, the channel pattern is deleted. Be sure to configure the pole settings for channels (`channel.setpole`) before creating a channel pattern.

Channel patterns are stored when you run the `createconfigscript()` command or `setup.save()` command.

Channel patterns are lost when power is cycled. Use `setup.recall()` or a script created with `createconfigscript()` to restore them.

The following restrictions exist when naming a channel pattern:

- The name must contain only letters, numbers, or underscores
- The name must start with a letter
- The name is case sensitive

Examples of valid names:

- `Channels`
- `Chans`
- `chans`
- `Path1`
- `Path20`
- `path_3`

Examples of invalid names:

- `1path` (invalid because it starts with a number)
- `my chans` (invalid because it contains a space)
- `My,chans` (invalid because it contains a comma)
- `Path1:10` (invalid because it contains a colon)

An error is generated if:

- The *`name`* parameter already exists as a label
- Any channel is forbidden to close
- Insufficient memory exists to create the channel pattern
- The parameter string contains `slot`*`X`* (where *`X`* equals 1 to 6 for Model 707B, or 1 for Model 708B) or `allslots`
- The name parameter contains a space character
- The pattern name exceeds 19 characters

Example 1

```
channel.pattern.setimage("1A01:1A10", "Channels")

oldList = channel.pattern.getimage("Channels")
newList = oldList .. ", 1C11"
channel.pattern.delete("Channels")
channel.pattern.setimage(newList, "Channels")
channel.close("Channels, 1D11")

Items = channel.pattern.getimage("Channels")
channel.pattern.setimage(Items, "Pattern")
channel.pattern.delete("Channels")
```

For this example, assume there is a Keithley Model 7174 or similar card in slot 1.
Creates a pattern, appends a channel to the pattern by retrieving the pattern and recreating it, and then renames the pattern.

Also see

createconfigscript() (on page 7-49)
channel.pattern.catalog() (on page 7-36)
channel.pattern.delete() (on page 7-37)
channel.pattern.getimage() (on page 7-37)
channel.pattern.snapshot() (on page 7-40)

channel.pattern.snapshot()

This function creates a channel pattern.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Recall setup Pole setting change	Create configuration script Save setup	Not applicable

Usage

```
channel.pattern.snapshot("name")
```

name	A string representing the name to associate with the present state of channels

Details

This command stores an image of presently closed channels and associates them with the *name* parameter.

If *name* is already used for an existing channel pattern, that pattern is overwritten with the new pattern channel image (if no errors occur).

The following restrictions exist when naming a channel pattern:

- The name must contain only letters, numbers, or underscores
- The name must start with a letter
- The name is case sensitive

Examples of valid names:

- `Channels`
- `Chans`
- `chans`
- `Path1`
- `Path20`
- `path_3`

Examples of invalid names:

- `1path` (invalid because it starts with a number)
- `my chans` (invalid because it contains a space)
- `My,chans` (invalid because it contains a comma)
- `Path1:10` (invalid because it contains a colon)

An error is generated if:

- The *name* parameter already exists as a label
- Insufficient memory exists to save the channel pattern and name in persistent memory
- The pattern name exceeds 19 characters or contains a space

Issuing this function on an existing pattern invalidates the existing scan list (the pattern may or may not be used in the present scan list). Creating a new pattern does not invalidate the existing scan list.

Channel patterns are stored when you run the `createconfigscript()` command or `setup.save()` command.

Channel patterns are lost when power is cycled. Use `setup.recall()` or a script created with `createconfigscript()` to restore them.

If you change a pole setting for a channel that is associated with a channel pattern, the channel pattern is deleted. Be sure to configure the pole setting for channels (`channel.setpole`) before creating a channel pattern.

Example

`channel.pattern.snapshot("voltagePath")`
Creates a pattern named `voltagePath` that contains the presently closed channels.

Also see

createconfigscript() (on page 7-49)
channel.pattern.catalog() (on page 7-36)
channel.pattern.delete() (on page 7-37)
channel.pattern.getimage() (on page 7-37)
channel.pattern.setimage() (on page 7-38)

channel.reset()

This function resets the specified channel list items to factory default settings.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
channel.reset("channelList")
```

channelList	A string that lists the items to reset; the string can include: ▪ `allslots` ▪ `slotX`, where *X* is the slot number ▪ channel patterns ▪ channels, including a range of channels

Details

For the items specified in *channelList*, the following actions occur:

- Any closed channels open
- Additional user delay is set to zero (0)
- Labels are removed
- If the channel is forbidden to close, it is cleared from being forbidden to close
- If the channels are used in channel patterns, the channel patterns that contain the channels are deleted.

If you use this function to reset a channel that is involved in scanning, it invalidates the existing scan list. The list has to be recreated before scanning again.

The rest of the instrument settings are unaffected. To reset the entire system to factory default settings, use the `reset()` command.

Example 1

```
channel.reset("allslots")
```

Performs a reset operation on all channels on the instrument.

Example 2

```
channel.reset("slot1")
```

Resets channels on slot 1 only.

Example 3

```
channel.reset("3A01:3A05")
```

Resets only row 1, columns 1 through 5 on slot 3.

Example 4

`channel.reset("5C05, 5D16")`	
Resets row 3, column 5, and row 4 column 16, on slot 5.	

Also see

Channel functions and attributes (on page 5-6)
reset() (on page 7-142)

channel.setdelay()

This function sets additional delay time for specified channels.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Recall setup Power cycle	Create configuration script Save setup	0

Usage

```
channel.setdelay("channelList", delay)
```

channelList	A string listing the channels for which to add delay time
delay	Delay time for the selected channels; minimum is 0 seconds

Details

The user delay is an additional delay that is added after a channel is closed. You can use this delay to allow additional settle time for a signal on that channel. For most cards, the resolution of the delay is 10 µs. However, check the documentation for your card to verify. To see if the delay value was modified after setting, use the `channel.getdelay()` command to query.

Channel patterns get their delay from the channels that comprise the pattern. Therefore, specify the delay for a pattern through the channels. A pattern incurs the longest delay of all channels comprising that pattern.

An error message is generated if:

- The value is an invalid setting for the specified channel
- A channel pattern is specified
- The channel is for an empty slot

Command processing will stop as soon as an error is detected and no delay times will be modified.

NOTE

Pseudocards do not replicate the additional delay time.

Example 1

`channel.setdelay("1A03, 1A05", 50e-6)`
Sets row 1 and columns 3 and 5 of slot 1 for a delay time of 50 µs.

Example 2

`channel.setdelay("slot1", 0)`
Sets the channels on slot 1 for 0 delay time.

Also see

channel.getdelay() (on page 7-28)

channel.setforbidden()

This function prevents the closure of specified channels.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Recall setup	Create configuration script Save setup	Not forbidden

Usage

```
channel.setforbidden("channelList")
```

channelList	A string that lists the channels to make forbidden to close

Details

The *channelList* parameter indicates the scope of channels affected and may include:

- `allslots` or `slotX` (where *X* equals 1 to 6 for Model 707B, or 1 for Model 708B)
- Channel ranges or individual channels

This function prevents all items contained in the channel list parameter from closing. It applies the "forbidden to close" attribute to the specified channels. To remove the "forbidden to close" attribute, use `channel.clearforbidden()`.

If a channel that is being set to forbidden is used in a channel pattern, the channel pattern is deleted when the channel is set to forbidden. If the *channelList* parameter includes a channel pattern, the channel pattern is deleted when the channels in the patterns are successfully set to forbidden to close.

The channels in the *channelList* parameter must be installed in the instrument.

If the scan list contains a channel that is forbidden, the scan list is invalidated.

Example

```
channel.pattern.setimage("1A01,1A02", "patternA")
channel.pattern.setimage("1B01,1B02", "patternB")
channel.pattern.setimage("1C01,1C02", "patternC")

for name in channel.pattern.catalog() do
   print(name .. " = " .. channel.pattern.getimage(name))
end
channel.setforbidden("1A02, 1B01")
for name in channel.pattern.catalog() do
   print(name .. " = " .. channel.pattern.getimage(name))
end
```

Create three channel patterns and then print.
Set forbidden for one channel from `patternA` and `patternB`, then print the catalog again. Only `patternC` should remain.
Assuming no existing channel patterns, the output is:

```
patternC = 1C01,1C02
patternA = 1A01,1A02
patternB = 1B01,1B02
patternC = 1C01,1C02
```

Also see

channel.clearforbidden() (on page 7-17)
channel.getforbidden() (on page 7-29)

channel.setlabel()

This function sets the label associated with a channel.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Recall setup Power cycle	Create configuration script Save setup	No label

Usage

```
channel.setlabel("channelList", "labelname")
```

channelList	A string that lists the channel to which to set the label; the string cannot contain spaces
labelname	A string that contains the label for the channel in *channelList*, up to 19 characters

Details

This command sets the label of the specified channel to the label value. The channel attributes associated with the channel remain unchanged except for the label.

After defining a label, you can use it to specify the channel instead of using the channel specifier.

The label must be unique. In addition, it cannot be the same as the name of a channel pattern, row label, or column label. If you specify a label that already exists, an error message is generated that indicates a parameter error and channel that that is already associated the specified label.

To clear a label, set it to an empty string ("").

Example 1

```
channel.setlabel("1A01", "start")
channel.close("start")
print(channel.getclose("allslots"))
```

Sets the label for channel row 1 and column 01 on slot 1 to `start`.

Output:

`1A01`

Example 2

```
channel.setlabel("1A01", "")
```

Clears the label for channel row 1 and column 01, name is returned to `1A01`.

Example 3

```
channel.setlabel("1A01", " ")
```

Also clears the label for channel row 1 and column 01, slot 1 and returns it to `1A01`.

Also see

channel functions and attributes (on page 5-6)
channel.setlabelcolumn() (on page 7-46)
channel.setlabelrow() (on page 7-48)

channel.setlabelcolumn()

This function assigns a label to a column.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset	Create configuration script	Empty string or blank

Usage

```
channel.setlabelcolumn(channel, label)
```

channel	A string that specifies the channel that contains the column to which the label will be applied
label	A string that lists the label for the channel column, up to eight characters

Details

A column label can be applied to columns of a matrix card. The label is then used on the display and can be accessed in a channel list. For *channel*, you can specify any channel in the column. You cannot specify a channel pattern.

The row and column *label* parameters must be unique. In addition, they cannot be the same as the name of a channel pattern or channel label.

After defining a column label, you can use it with a row label to specify a channel instead of the channel specifier.

On the crosspoint display, the first four characters of the label are displayed. On the bottom display, the full label is displayed.

NOTE

Since a column label is common to all channels in that column, you only need to assign the column label to one channel.

Example 1

```
channel.setlabelrow("1B01", "SMU2")
channel.setlabelcolumn("1B02", "DUT2")
channel.close("SMU2+DUT2")
print(channel.getclose("allslots"))
```

Sets the label for the slot 1, row 2 to `SMU2` and slot 1, column 2 to `DUT2`.
Use the labels to close a channel.
Output:
```
1B02
```

Example 2

```
channel.setlabelcolumn("1A01", "")
```

Clears the label for column 1 on slot 1 and returns it to the default.

Example 3

```
channel.setlabelcolumn("1A01", " ")
```

Clears the label for column 1 on slot 1 and returns it to the default.

Example 4

```
channel.setlabelcolumn("2B01", "TwoC")
print(channel.getlabelcolumn("slot2"))
```

This example assumes a Model 7072 in slot 2.
Set the label to be `TwoC`, which assigns the label to all channels in the column. Output:
```
TwoC,,,,,,,,,,,,TwoC,,,,,,,,,,,,TwoC,,,,,,,,,,,,TwoC,,,,,,,,,,,,TwoC,,,,,,,,,,,,Tw
   oC,,,,,,,,,,,,TwoC,,,,,,,,,,,,TwoC,,,,,,,,,,,
```

Also see

channel.getlabelcolumn() (on page 7-31)
channel.getlabelrow() (on page 7-32)
channel.setlabelrow() (on page 7-48)

channel.setlabelrow()

This function assigns a label to a row.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset	Create configuration script	Empty string or blank

Usage

`channel.setlabelrow(`*`channel`*`, `*`label`*`)`

channel	A string that specifies the channel that contains the row to which to apply a label
label	A string that lists the label for the channel row, up to eight characters

Details

A row label can be applied to columns of a matrix card. The label is used on the display and can be accessed in a channel list. For *channel*, you can specify any channel in the row. You cannot specify a channel pattern.

The row and column *label* parameters must be unique. In addition, they cannot be the same as the name of a channel pattern.

You can only set labels for slots and channels that are installed in the instrument.

After defining a row label, you can use it to specify a channel instead of the default channel designation.

On the crosspoint display, the first four characters of the label are displayed. On the bottom display, the full label is displayed.

Labels can only be set for matrix cards.

NOTE

Since a row label is common to all channels in that row, you only need to assign the row label to one channel.

Example 1

```
channel.setlabelrow("1B01", "SMU2")
channel.setlabelcolumn("1B02", "DUT2")
channel.close("SMU2+DUT2")
print(channel.getclose("allslots"))
```

Sets the label for the slot 1, row 2 to `SMU2` and slot 1, column 2 to `DUT2`.
Use the labels to close a channel.
Output:
`1B02`

Example 2

```
channel.setlabelrow("1A01", "")
```

Clears the label for row 1 on slot 1 and return to the default.

Example 3

`channel.setlabelrow("1A01", " ")`	
Clears the label for row 1 on slot 1 and return to the default.	

Example 4

```
channel.setlabelrow("2B01", "Row2")
print(channel.getlabelrow("slot2"))
```

This example assumes a 7072 in slot 2.
Set the label to be `Row2`, which assigns the label to all channels in the row. The output is now :

```
,,,,,,,,,,,,Row2,Row2,Row2,Row2,Row2,Row2,Row2,Row2,Row2,Row2,Row2,Row2,,,,,,,,,,,
    ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
```

Also see

channel.getlabelcolumn() (on page 7-31)
channel.getlabelrow() (on page 7-32)
channel.setlabelcolumn() (on page 7-46)

createconfigscript()

This function creates a setup file that captures most of the present settings of the instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
createconfigscript("scriptName")
```

scriptName	A string that represents the name of the script that will be created

Details

If *scriptName* is set to `autoexec`, the autoexec script in the instrument is replaced by the new configuration script.

If *scriptName* is set to the name of an existing script, the existing script is overwritten.

Once created, the script that contains the settings can be run and edited like any other script.

Example

```
createconfigscript("August2013")
```

Captures the present settings of the instrument into a script named `August2013`.

Also see

Autoexec script (on page 6-9)
Manage scripts (on page 6-3)

dataqueue.add()

This function adds an entry to the data queue.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
result = dataqueue.add(value)
result = dataqueue.add(value, timeout)
```

result	The resulting value of `true` or `false` based on the success of the function
value	The data item to add; *value* can be of any type
timeout	The maximum number of seconds to wait for space in the data queue

Details

You cannot use the *timeout* value when accessing the data queue from a remote node. You can only use the *timeout* value while adding data to the local data queue.

The *timeout* value is ignored if the data queue is not full.

The `dataqueue.add()` function returns `false`:

- If the timeout expires before space is available in the data queue
- If the data queue is full and a *timeout* value is not specified

If the value is a table, a duplicate of the table and any subtables is made. The duplicate table does not contain any references to the original table or to any subtables.

Example

```
dataqueue.clear()
dataqueue.add(10)
dataqueue.add(11, 2)
result = dataqueue.add(12, 3)
if result == false then
   print("Failed to add 12 to the dataqueue")
end
print("The dataqueue contains:")
while dataqueue.count > 0 do
   print(dataqueue.next())
end
```

Clear the data queue.
Each line adds one item to the data queue.
Output:

```
The dataqueue contains:
10
11
12
```

Also see

dataqueue.CAPACITY (on page 7-51)
dataqueue.clear() (on page 7-52)
dataqueue.count (on page 7-53)
dataqueue.next() (on page 7-54)
Using the data queue for real-time communications (on page 6-55)

dataqueue.CAPACITY

This constant is the maximum number of entries that you can store in the data queue.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Constant	Yes			

Usage

```
count = dataqueue.CAPACITY
```

count	The variable that is assigned the value of `dataqueue.CAPACITY`

Details

This constant always returns the maximum number of entries that can be stored in the data queue.

Example

```
MaxCount = dataqueue.CAPACITY
while dataqueue.count < MaxCount do
   dataqueue.add(1)
end
print("There are " .. dataqueue.count .. " items in the data queue")
```

This example fills the data queue until it is full and prints the number of items in the queue.
Output:

```
There are 128 items in the data queue
```

Also see

dataqueue.add() (on page 7-50)
dataqueue.clear() (on page 7-52)
dataqueue.count (on page 7-53)
dataqueue.next() (on page 7-54)
Using the data queue for real-time communications (on page 6-55)

dataqueue.clear()

This function clears the data queue.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
dataqueue.clear()
```

Details

This function forces all `dataqueue.add()` commands that are in progress to time out and deletes all data from the data queue.

Example

```
MaxCount = dataqueue.CAPACITY
while dataqueue.count < MaxCount do
   dataqueue.add(1)
end
print("There are " .. dataqueue.count
   .. " items in the data queue")
dataqueue.clear()
print("There are " .. dataqueue.count
   .. " items in the data queue")
```

This example fills the data queue and prints the number of items in the queue. It then clears the queue and prints the number of items again.

Output:

```
There are 128 items in the data queue
There are 0 items in the data queue
```

Also see

dataqueue.add() (on page 7-50)
dataqueue.CAPACITY (on page 7-51)
dataqueue.count (on page 7-53)
dataqueue.next() (on page 7-54)
Using the data queue for real-time communications (on page 6-55)

dataqueue.count

This attribute contains the number of items in the data queue.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
count = dataqueue.count
```

count	The number of items in the data queue

Details

The count is updated as entries are added with `dataqueue.add()` and read from the data queue with `dataqueue.next()`. It is also updated when the data queue is cleared with `dataqueue.clear()`.

A maximum of `dataqueue.CAPACITY` items can be stored at any one time in the data queue.

Example

```
MaxCount = dataqueue.CAPACITY
while dataqueue.count < MaxCount do
   dataqueue.add(1)
end
print("There are " .. dataqueue.count
   .. " items in the data queue")
dataqueue.clear()
print("There are " .. dataqueue.count
   .. " items in the data queue")
```

This example fills the data queue and prints the number of items in the queue. It then clears the queue and prints the number of items again.

Output:

```
There are 128 items in the data queue
There are 0 items in the data queue
```

Also see

dataqueue.add() (on page 7-50)
dataqueue.CAPACITY (on page 7-51)
dataqueue.clear() (on page 7-52)
dataqueue.next() (on page 7-54)
Using the data queue for real-time communications (on page 6-55)

dataqueue.next()

This function removes the next entry from the data queue.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
value = dataqueue.next()
value = dataqueue.next(timeout)
```

value	The next entry in the data queue
timeout	The number of seconds to wait for data in the queue

Details

If the data queue is empty, the function waits up to the *timeout* value.

If data is not available in the data queue before the *timeout* expires, the return value is `nil`.

The entries in the data queue are removed in first-in, first-out (FIFO) order.

If the value is a table, a duplicate of the original table and any subtables is made. The duplicate table does not contain any references to the original table or to any subtables.

Example

```
dataqueue.clear()
for i = 1, 10 do
   dataqueue.add(i)
end
print("There are " .. dataqueue.count .. " items in the data queue")

while dataqueue.count > 0 do
   x = dataqueue.next()
   print(x)
end
print("There are " .. dataqueue.count .. " items in the data queue")
```

Clears the data queue, adds ten entries, then reads the entries from the data queue. Output:

```
There are 10 items in the data queue
1
2
3
4
5
6
7
8
9
10
There are 0 items in the data queue
```

Your output may differ depending on the setting of `format.asciiprecision`.

Also see

dataqueue.add() (on page 7-50)
dataqueue.CAPACITY (on page 7-51)
dataqueue.clear() (on page 7-52)
dataqueue.count (on page 7-53)
format.asciiprecision (on page 7-94)
Using the data queue for real-time communications (on page 6-55)

delay()

This function delays the execution of the commands that follow it.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
delay(seconds)
```

seconds	The number of seconds to delay (0 s to 100,000 s)

Details

The instrument delays execution of the commands for at least the specified number of seconds and fractional seconds. However, the processing time may cause the instrument to delay 5 μs to 10 μs (typical) more than the requested delay.

Example

```
beeper.beep(0.5, 2400)
delay(0.250)
beeper.beep(0.5, 2400)
```

Emit a double-beep at 2400 Hz. The sequence is 0.5 s on, 0.25 s off, 0.5 s on.

Also see

None

digio.readbit()

This function reads one digital I/O line.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
data = digio.readbit(N)
```

data	The value to read: 1 or 0
N	Digital I/O line number to be read: 1 to 14

Details

A returned value of zero (0) indicates that the line is low. A returned value of one (1) indicates that the line is high.

Example

```
--Read the value of digital I/O line 2
local value = digio.readbit(2)
print("The value of line 2 is: " .. value)
```

Reads the state of digital I/O line 2.
Output:
`2.00<expNote+00`

Also see

digio.readport() (on page 7-56)
digio.writebit() (on page 7-65)
digio.writeport() (on page 7-66)
Digital I/O port (on page 3-9)

digio.readport()

This function reads the digital I/O port.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
data = digio.readport()
```

data	The present value of the input lines on the digital I/O port

Details

The binary equivalent of the returned value indicates the value of the input lines on the I/O port. The least significant bit (bit B1) of the binary number corresponds to line 1; bit B14 corresponds to line 14.

For example, a returned value of 170 has a binary equivalent of 000000010101010, which indicates that lines 2, 4, 6, and 8 are high (1), and the other 10 lines are low (0).

Example

`data = digio.readport()` `print(data)`	
Assume lines 2, 4, 6, and 8 are set high when the I/O port is read. Output: `1.7000000e+02` This is binary 10101010.	

Also see

digio.readbit() (on page 7-55)
digio.writebit() (on page 7-65)
digio.writeport() (on page 7-66)
Digital I/O port (on page 3-9)

digio.trigger[N].assert()

This function asserts a trigger pulse on one of the digital I/O lines.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
digio.trigger[N].assert()
```

N	Digital I/O trigger line: 1 to 14

Details

The pulse width that is set determines how long the instrument asserts the trigger.

Example 1

```
digio.trigger[2].assert()
```

Asserts digital I/O trigger line 2.

Example 2

```
digio.trigger[2].logic = digio.LOGIC_NEGATIVE
digio.trigger[2].mode = digio.MODE_TRIGGER_OUT
digio.trigger[2].pulsewidth = 0
digio.trigger[2].assert()
digio.trigger[2].release()
```

Negative trigger out.

Example 3

```
digio.trigger[2].logic = digio.LOGIC_POSITIVE
digio.trigger[2].mode = digio.MODE_TRIGGER_OUT
digio.trigger[2].pulsewidth = 10e-6
digio.trigger[2].assert()}}
```

Positive trigger out.

Also see

digio.trigger[N].pulsewidth (on page 7-61)

digio.trigger[N].clear()

This function clears the trigger event on a digital I/O line.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`digio.trigger[N].clear()`

N	Digital I/O trigger line: 1 to 14

Details

The event detector of a trigger enters the detected state when an event is detected. It is cleared when `digio.trigger[N].wait()` or `digio.trigger[N].clear()` is called.

`digio.trigger[N].clear()` clears the event detector of the specified trigger line, discards the history of the trigger line, and clears the `digio.trigger[N].overrun` attribute.

Example

`digio.trigger[2].clear()`	Clears the trigger event detector on I/O line 2.

Also see

digio.trigger[N].overrun (on page 7-60)
digio.trigger[N].wait() (on page 7-64)

digio.trigger[N].EVENT_ID

This constant identifies the trigger event generated by the digital I/O line *N*.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Constant	Yes			

Usage

`eventID = digio.trigger[N].EVENT_ID`

eventID	The trigger event number
N	Digital I/O trigger line: 1 to 14

Details

To have another trigger object respond to trigger events generated by the trigger line, set the stimulus attribute of the other object to the value of this constant.

Example 1

`digio.trigger[5].stimulus = digio.trigger[3].EVENT_ID`
Uses a trigger event on digital I/O trigger line 3 to be the stimulus for digital I/O trigger line 5.

Example 2

`scan.trigger.arm.stimulus = digio.trigger[3].EVENT_ID`	
Uses a trigger event on digital I/O trigger line 3 to be the stimulus for starting a scan.	

Also see

None

digio.trigger[N].mode

This attribute sets the mode in which the trigger event detector and the output trigger generator operate on the given trigger line.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Digital I/O trigger *N* reset Recall setup	Create configuration script	0 (digio.TRIG_BYPASS)

Usage

```
triggerMode = digio.trigger[N].mode
digio.trigger[N].mode = triggerMode
```

triggerMode	The trigger mode; see **Details** for values
N	Digital I/O trigger line: 1 to 14

Details

Set *triggerMode* to one of the values in the following table.

Trigger mode values

triggerMode	**Description**
`digio.TRIG_BYPASS` or `0`	Allows direct control of the line.
`digio.TRIG_FALLING` or `1`	Detects falling-edge triggers as input; asserts a TTL-low pulse for output.
`digio.TRIG_RISING` or `2`	If the programmed state of the line is high, the `digio.TRIG_RISING` mode behavior is similar to `digio.TRIG_RISINGA`. If the programmed state of the line is low, the `digio.TRIG_RISING` mode behavior is similar to `digio.TRIG_RISINGM`. Only use this setting if necessary for compatibility with other Keithley products.
`digio.TRIG_EITHER` or `3`	Detects rising- or falling-edge triggers as input. Asserts a TTL-low pulse for output.
`digio.TRIG_SYNCHRONOUSA` or `4`	Detects the falling-edge input triggers and automatically latches and drives the trigger line low. Asserting the output trigger releases the latched line.
`digio.TRIG_SYNCHRONOUS` or `5`	Detects the falling-edge input triggers and automatically latches and drives the trigger line low. Asserts a TTL-low pulse as an output trigger.
`digio.TRIG_SYNCHRONOUSM` or `6`	Detects rising-edge triggers as input. Asserts a TTL-low pulse for output.
`digio.TRIG_RISINGA` or `7`	Detects rising-edge triggers as input. Asserts a TTL-low pulse for output.
`digio.TRIG_RISINGM` or `8`	Asserts a TTL-high pulse for output. Input edge detection is not possible in this mode.

When programmed to any mode except `digio.TRIG_BYPASS`, the output state of the I/O line is controlled by the trigger logic and the user-specified output state of the line is ignored.

Use of either `digio.TRIG_SYNCHRONOUSA` or `digio.TRIG_SYNCHRONOUSM` is preferred over `digio.TRIG_SYNCHRONOUS`. `digio.TRIG_SYNCHRONOUS` is provided for compatibility with the digital I/O and TSP-Link triggering on other Keithley products.

To control the line state, set the mode to `digio.TRIG_BYPASS` and use the `digio.writebit()` and `digio.writeport()` commands.

Example

```
digio.trigger[4].mode = 2
```

Sets the trigger mode for I/O line 4 to `digio.TRIG_RISING`.

Also see

digio.trigger[N].clear() (on page 7-58)
digio.trigger[N].reset() (on page 7-62)
digio.writebit() (on page 7-65)
digio.writeport() (on page 7-66)
Scan and trigger (on page 2-21)

digio.trigger[N].overrun

This attribute returns the event detector overrun status.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Instrument reset Digital I/O trigger *N* clear Digital I/O trigger *N* reset Recall setup	Not saved	Not applicable

Usage

```
overrun = digio.trigger[N].overrun
```

overrun	Trigger overrun state: `true` or `false`
N	Digital I/O trigger line: 1 to 14

Details

If this is `true`, an event was ignored because the event detector was already in the detected state when the event occurred.

This is an indication of the state of the event detector built into the line itself. It does not indicate if an overrun occurred in any other part of the trigger model or in any other detector that is monitoring the event.

Example

`overrun = digio.trigger[1].overrun` `print(overrun)`	
If there is no trigger overrun, the following text is output: `false`	

Also see

digio.trigger[N].clear() (on page 7-58)
digio.trigger[N].reset() (on page 7-62)

digio.trigger[N].pulsewidth

This attribute describes the length of time that the trigger line is asserted for output triggers.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Digital I/O trigger *N* reset Recall setup	Create configuration script	10e-6 (10 µs)

Usage

```
width = digio.trigger[N].pulsewidth
digio.trigger[N].pulsewidth = width
```

width	The pulse width in seconds
N	Digital I/O trigger line: 1 to 14

Details

Setting the pulse width to zero (0) seconds asserts the trigger indefinitely. To release the trigger line, use `digio.trigger[N].release()`.

Example

`digio.trigger[4].pulsewidth = 20e-6`	
Sets the pulse width for trigger line 4 to 20 µs.	

Also see

digio.trigger[N].assert() (on page 7-57)
digio.trigger[N].release() (on page 7-62)
digio.trigger[N].reset() (on page 7-62)

digio.trigger[N].release()

This function releases an indefinite length or latched trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
digio.trigger[N].release()
```

N	Digital I/O trigger line: 1 to 14

Details

Releases a trigger that was asserted with an indefinite pulsewidth time. It also releases a trigger that was latched in response to receiving a synchronous mode trigger. Only the specified trigger line is affected.

Example

`digio.trigger[4].release()`	Releases digital I/O trigger line 4.

Also see

digio.trigger[N].assert() (on page 7-57)
digio.trigger[N].pulsewidth (on page 7-61)

digio.trigger[N].reset()

This function resets trigger values to their factory defaults.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
digio.trigger[N].reset()
```

N	Digital I/O trigger line: 1 to 14

Details

This function resets the following attributes to factory default settings:

- `digio.trigger[N].mode`
- `digio.trigger[N].pulsewidth`
- `digio.trigger[N].stimulus`

It also clears `digio.trigger[N].overrun`.

Example

```
digio.trigger[3].mode = 2
digio.trigger[3].pulsewidth = 50e-6
digio.trigger[3].stimulus = digio.trigger[5].EVENT_ID
print(digio.trigger[3].mode, digio.trigger[3].pulsewidth,
   digio.trigger[3].stimulus)
digio.trigger[3].reset()
print(digio.trigger[3].mode, digio.trigger[3].pulsewidth,
   digio.trigger[3].stimulus)
```

Set the digital I/O trigger line 3 for a falling edge with a pulsewidth of 50 µs.
Use digital I/O line 5 to trigger the event on line 3.
Reset the line to factory default values.
Output before reset:
`2.0000000e+00        5.0000000e-05        5.0000000e+00`
Output after reset:
`0.0000000e+00        1.0000000e-05        0.0000000e+00`

Also see

digio.trigger[N].mode (on page 7-59)
digio.trigger[N].overrun (on page 7-60)
digio.trigger[N].pulsewidth (on page 7-61)
digio.trigger[N].stimulus (on page 7-63)

digio.trigger[N].stimulus

This attribute selects the event that causes a trigger to be asserted on the digital output line.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Digital I/O trigger *N* reset Save setup	Create configuration script	0

Usage

```
triggerStimulus = digio.trigger[N].stimulus
digio.trigger[N].stimulus = triggerStimulus
```

triggerStimulus	The event identifier for the triggering event
N	Digital I/O trigger line: 1 to 14

Details

Set this attribute to zero (0) to disable the automatic trigger output.

Do not use the stimulus attribute for generating output triggers under script control. Use `digio.trigger[N].assert()` instead.

The trigger stimulus for a digital I/O line may be set to one of the existing trigger event IDs, described in the following table.

Trigger event IDs

Trigger event ID	Description
`digio.trigger[N].EVENT_ID`	An edge (either rising, falling, or either based on the configuration of the line) on the digital input line.
`display.trigger.EVENT_ID`	The trigger key on the front panel is pressed.
`trigger.EVENT_ID`	A `*trg` message on the active command interface. If GPIB is the active command interface, a GET message also generates this event.
`trigger.blender[N].EVENT_ID`	A combination of events has occurred.
`trigger.timer[N].EVENT_ID`	A delay expired.
`tsplink.trigger[N].EVENT_ID`	An edge (either rising, falling, or either based on the configuration of the line) on the TSP-Link trigger line.
`lan.trigger[N].EVENT_ID`	A LAN trigger event has occurred.
`scan.trigger.EVENT_SCAN_READY`	Scan ready event.
`scan.trigger.EVENT_SCAN_START`	Scan start event.
`scan.trigger.EVENT_CHANNEL_READY`	Channel ready event.
`scan.trigger.EVENT_SCAN_COMP`	Scan complete event.
`scan.trigger.EVENT_IDLE`	Idle event.

Example 1

```
digio.trigger[3].stimulus = 0
```

Clear the trigger stimulus of digital I/O line 3.

Example 2

```
digio.trigger[3].stimulus = scan.trigger.EVENT_CHANNEL_READY
```

Set the trigger stimulus of digital I/O line 3 to be the channel ready event during a scan.

Also see

digio.trigger[N].assert() (on page 7-57)
digio.trigger[N].clear() (on page 7-58)
digio.trigger[N].reset() (on page 7-62)

digio.trigger[N].wait()

This function waits for a trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
triggered = digio.trigger[N].wait(timeout)
```

triggered	The value is `true` if a trigger is detected, or `false` if no triggers are detected during the timeout period
N	Digital I/O trigger line: 1 to 14
timeout	Timeout in seconds

Details

This function pauses trigger operation up to the seconds set by *timeout* for an input trigger. If one or more trigger events are detected since the last time `digio.trigger[N].wait()` or `digio.trigger[N].clear()` was called, this function returns a value immediately. After waiting for a trigger with this function, the event detector is automatically reset and ready to detect the next trigger. This is true regardless of the number of events detected.

Example

```
triggered = digio.trigger[4].wait(3)
print(triggered)
```

Waits up to three seconds for a trigger to be detected on trigger line 4, then outputs the results.
Output if no trigger is detected:
`false`
Output if a trigger is detected:
`true`

Also see

digio.trigger[N].clear() (on page 7-58)

digio.writebit()

This function sets a digital I/O line high or low.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`digio.writebit(N, data)`

N	Digital I/O trigger line: 1 to 14
data	The value to write to the bit: ▪ 0 (low) ▪ Non-zero (high)

Details

If the output line is write-protected using the `digio.writeprotect` attribute, the command is ignored.

The `reset()` function does not affect the present state of the digital I/O lines.

Use the `digio.writebit()` and `digio.writeport()` commands to control the output state of the synchronization line when trigger operation is set to `digio.TRIG_BYPASS`.

The data must be zero (0) to clear the bit. Any value other than zero (0) sets the bit.

Example

`digio.writebit(4, 0)`	Sets digital I/O line 4 low (0).

Also see

digio.readbit() (on page 7-55)
digio.readport() (on page 7-56)
digio.trigger[N].mode (on page 7-59)
digio.writeport() (on page 7-66)
digio.writeprotect (on page 7-67)

digio.writeport()

This function writes to all digital I/O lines.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
digio.writeport(data)
```

data	Value to write to the port: 0 to 16383

Details

The binary representation of *data* indicates the output pattern to be written to the I/O port. For example, a *data* value of 170 has a binary equivalent of 00000010101010. Lines 2, 4, 6, and 8 are set high (1), and the other 10 lines are set low (0).

Write-protected lines are not changed.

The `reset()` function does not affect the present states of the digital I/O lines.

Use the `digio.writebit()` and `digio.writeport()` commands to control the output state of the synchronization line when trigger operation is set to `digio.TRIG_BYPASS`.

Example

`digio.writeport(255)`
Sets digital I/O Lines 1 through 8 high (binary 00000011111111).

Also see

digio.readbit() (on page 7-55)
digio.readport() (on page 7-56)
digio.writebit() (on page 7-65)
digio.writeprotect (on page 7-67)

digio.writeprotect

This attribute contains the write-protect mask that protects bits from changes from the `digio.writebit()` and `digio.writeport()` functions.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup	Create configuration script	0

Usage

```
mask = digio.writeprotect
digio.writeprotect = mask
```

mask	The value that specifies the bit pattern for write-protect mask

Details

Bits that are set to one cause the corresponding line to be write-protected.

The binary equivalent of *mask* indicates the mask to be set for the I/O port. For example, a mask value of 7 has a binary equivalent of 00000000000111. This mask write-protects lines 1, 2, and 3.

Example

`digio.writeprotect = 15`	Write-protects lines 1, 2, 3, and 4.

Also see

digio.writebit() (on page 7-65)
digio.writeport() (on page 7-66)

display.clear()

This function clears all lines of the front-panel display.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
display.clear()
```

Details

This function switches to the user screen and then clears the front-panel display.

The `display.clear()`, `display.setcursor()`, and `display.settext()` functions are overlapped commands. That is, the script does not wait for one of these commands to complete. These functions do not immediately update the display. For performance considerations, they update the display as soon as processing time becomes available.

Also see

display.setcursor() (on page 7-83)
display.settext() (on page 7-84)

display.getannunciators()

This function reads the annunciators (indicators) that are presently turned on.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

annunciators = display.getannunciators()

annunciators	The bitmasked value that shows which indicators are turned on

Details

This function returns a bitmasked value showing which indicators are turned on. The 16-bit binary equivalent of the returned value is the bitmask. The return value is a sum of set annunciators, based on the weighted value, as shown in the following table.

Annunciator (indicator) bitmasked values and equivalent constants			
Indicator	**Bit**	**Weighted value**	**Equivalent constant**
FILT	1	1	display.ANNUNCIATOR_FILTER
MATH	2	2	display.ANNUNCIATOR_MATH
4W	3	4	display.ANNUNCIATOR_4_WIRE
AUTO	4	8	display.ANNUNCIATOR_AUTO
ARM	5	16	display.ANNUNCIATOR_ARM
TRIG	6	32	display.ANNUNCIATOR_TRIGGER
* (asterisk)	7	64	display.ANNUNCIATOR_STAR
SMPL	8	128	display.ANNUNCIATOR_SAMPLE
EDIT	9	256	display.ANNUNCIATOR_EDIT
ERR	10	512	display.ANNUNCIATOR_ERROR
REM	11	1024	display.ANNUNCIATOR_REMOTE
TALK	12	2048	display.ANNUNCIATOR_TALK
LSTN	13	4096	display.ANNUNCIATOR_LISTEN
SRQ	14	8192	display.ANNUNCIATOR_SRQ
REAR	15	16384	display.ANNUNCIATOR_REAR
REL	16	32768	display.ANNUNCIATOR_REL

Example 1

```
testAnnunciators = display.getannunciators()
print(testAnnunciators)

rem = bit.bitand(testAnnunciators, 1024)
if rem > 0 then
   print("REM is on")
else
   print("REM is off")
end
```

REM indicator is turned on.
Output:

```
1.2800000e+03
REM is on
```

Example 2

`print(display.ANNUNCIATOR_EDIT)`
`print(display.ANNUNCIATOR_TRIGGER)`
`print(display.ANNUNCIATOR_AUTO)`

Output:
`2.5600000e+02`
`3.2000000e+01`
`8.0000000e+00`

Also see

bit.bitand() (on page 7-8)

display.getcursor()

This function reads the present position of the cursor on the front-panel display.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`row, column, style` `= display.getcursor()`

`row`	The row where the cursor is: `1` (top row); `2` (bottom row)
`column`	The column where the cursor is: ▪ If the cursor is in the top row: 1 to 20 ▪ If the cursor is in the bottom row: 1 to 32
`style`	Visibility of the cursor: ▪ Invisible: `0` ▪ Blinking: `1`

Details

This function switches the front-panel display to the user screen (the text set by `display.settext()`), and then returns values to indicate the row that contains the cursor and the column position and cursor style.

Columns are numbered from left to right on the display.

Example 1

`testRow, testColumn = display.getcursor()` `print(testRow, testColumn)`
This example reads the cursor position into local variables and prints them. Example output: `1.0000000e+00      1.0000000e+00`

Example 2

`print(display.getcursor())`
This example prints the cursor position directly. In this example, the cursor is in row 1 at column 3, with an invisible cursor: `1.0000000e+00    3.0000000e+00    0.0000000e+00`

Also see

display.gettext() (on page 7-71)
display.screen (on page 7-81)
display.setcursor() (on page 7-83)
display.settext() (on page 7-84)

display.getlastkey()

This function retrieves the key code for the last pressed key.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
keyCode = display.getlastkey()
```

keyCode	A returned value that represents the last front-panel key pressed; see **Details** for more information

Details

A history of the key code for the last pressed front-panel key is maintained by the instrument. When the instrument is turned on, or when it is transitioning from local to remote operation, the key code is set to 0 (`display.KEY_NONE`).

Pressing the EXIT (LOCAL) key normally aborts a script. To use this function with the EXIT (LOCAL) key, you must set `display.locallockout` to `display.LOCK`.

The following table lists the *keyCode* value for each front-panel action.

Key codes

Value	Key list		Value	Key list
0	display.KEY_NONE		83	display.KEY_RUN
66	display.KEY_DELETE		84	display.KEY_TRIG
67	display.KEY_EXIT		86	display.KEY_STEP
69	display.KEY_CLOSE		87	display.KEY_CHAN
70	display.KEY_SLOT		90	display.KEY_INSERT
72	display.KEY_DISPLAY		91	display.KEY_MENU
74	display.KEY_ENTER		93	display.KEY_OPEN
76	display.KEY_LOAD		94	display.KEY_PATT
77	display.KEY_SCAN		97	display.WHEEL_ENTER
79	display.KEY_OPENALL		107	display.WHEEL_LEFT
80	display.KEY_CONFIG		114	display.WHEEL_RIGHT

NOTE

When using this function, use built-in constants, such as `display.KEY_STEP`, rather than the numeric value, such as `86`. This allows for better forward compatibility with firmware revisions.

Example

```
key = display.getlastkey()
print(key)
```

On the front panel, press the **MENU** key and then send the code shown here. This retrieves the key code for the last pressed key.
Output:
`6.8000000e+01`

Also see

display.locallockout (on page 7-78)
display.sendkey() (on page 7-82)

display.gettext()

This function reads the text displayed on the front panel.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
text = display.gettext()
text = display.gettext(embellished)
text = display.gettext(embellished, row)
text = display.gettext(embellished, row, columnStart)
text = display.gettext(embellished, row, columnStart, columnEnd)
```

text	The returned value, which contains the text that is presently displayed
embellished	Indicates type of returned text: `false` (simple text); `true` (text with embedded character codes)
row	Selects the row from which to read the text: `1` (row 1); `2` (row 2). If *row* is not included, both rows of text are read
columnStart	Selects the first column from which to read text; for row 1, the valid column numbers are 1 to 20; for row 2, the valid column numbers are 1 to 32; if nothing is selected, 1 is used
columnEnd	Selects the last column from which to read text; for row 1, the valid column numbers are 1 to 20; for row 2, the valid column numbers are 1 to 32; the default is 20 for row 1, and 32 for row 2

Details

Using the command without any parameters returns both lines of the front-panel display.

The `$N` character code is included in the returned value to show where the top line ends and the bottom line begins. This is not affected by the value of *embellished*.

When *embellished* is set to `true`, all other character codes are returned along with the message. When *embellished* is set to `false`, only the message and the `$N` character code is returned. For information on the embedded character codes, see display.settext() (on page 7-84).

The display is not switched to the user screen (the screen set using `display.settext()`). Text is read from the active screen.

Example 1

```
display.clear()
display.setcursor(1, 1)
display.settext("ABCDEFGHIJ$DKLMNOPQRST")
display.setcursor(2, 1)
display.settext("abcdefghijklm$Bnopqrstuvwxyz$F123456")
print(display.gettext())
print(display.gettext(true))
print(display.gettext(false, 2))
print(display.gettext(true, 2, 9))
print(display.gettext(false, 2, 9, 10))
```

This example shows how to retrieve the display text in multiple ways. The output is:

```
ABCDEFGHIJKLMNOPQRST$Nabcdefghijklmnopqrstuvwxyz123456
$RABCDEFGHIJ$DKLMNOPQRST$N$Rabcdefghijklm$Bnopqrstuvwxyz$F123456
abcdefghijklmnopqrstuvwxyz123456
$Rijklm$Bnopqrstuvwxyz$F123456
ij
```

Example 2

```
display.clear()
display.settext("User Screen")
text = display.gettext()
print(text)
```

This outputs all text in both lines of the display:

```
User Screen       $N
```

This indicates that the message “User Screen” is on the top line. The bottom line is blank.

Also see

display.clear() (on page 7-67)
display.getcursor() (on page 7-69)
display.setcursor() (on page 7-83)
display.settext() (on page 7-84)

display.inputvalue()

This function displays a formatted input field on the front-panel display that the operator can edit.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
display.inputvalue("format")
display.inputvalue("format", default)
display.inputvalue("format", default, minimum)
display.inputvalue("format", default, minimum, maximum)
```

format	A string that defines how the input field is formatted; see **Details** for more information
default	The default value for the input value
minimum	The minimum input value
maximum	The maximum input value

Details

The *format* parameter uses zeros (0), the decimal point, polarity sign, and exponents to define how the input field is formatted. The *format* parameter can include the options shown in the following table.

Option	Description	Examples
E	Include the E to display the value exponentially	0.00000e+0
+	Allows operators to enter positive or negative values; if the "+" sign is not included, the operator cannot enter a negative value	+0.00
0	Defines the digit positions for the value; up to six zeros (0)	+00.0000e+00
.	Include to have a decimal point appear in the value	+0.00

The *default* parameter is the value shown when the value is first displayed.

You can use the *minimum* and *maximum* parameters to limit the values that can be entered. When + is not selected for *format*, the minimum limit must be more than or equal to zero (0). When limits are used, you cannot enter values above or below these limits.

The input value is limited to ±1e37.

Before calling `display.inputvalue()`, you should send a message prompt to the operator using `display.prompt()`. Make sure to position the cursor where the edit field should appear.

After this command is sent, script execution pauses until a user enters a value and presses the **ENTER** key.

For positive and negative entry (plus sign (+) used for the value field and the exponent field), polarity of a nonzero value or exponent can be toggled by positioning the cursor on the polarity sign and turning the navigation wheel ◔. Polarity is also toggled when using the navigation wheel ◔ to decrease or increase the value or exponent past zero. A zero (0) value or exponent (for example, +00) is always positive and cannot be toggled to negative polarity.

After executing this command and pressing the EXIT (LOCAL) key, the function returns `nil`.

Example

```
display.clear()
display.settext("Enter value between$N  -0.10 and 2.00:  ")
value = display.inputvalue("+0.00", 0.5, -0.1, 2.0)
print("Value entered = ", value)
```

Displays an editable field (+0.50) for operator input. The valid input range is −0.10 to +2.00, with a default of 0.50.

Output:

```
Value entered =    1.3500000e+00
```

Also see

display.prompt() (on page 7-79)
display.setcursor() (on page 7-83)
display.settext() (on page 7-84)

display.loadmenu.add()

This function adds an entry to the USER TESTS menu, which can be accessed by pressing the **LOAD** key on the front panel.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
display.loadmenu.add("displayName", "code")
display.loadmenu.add("displayName", "code", memory)
```

displayName	The name that is added to the USER TESTS menu
code	The code that is run from the USER TESTS menu
memory	Determines if code is saved to nonvolatile memory: ■ Does not save the code to nonvolatile memory: `0` or `display.DONT_SAVE` ■ Saves the code to nonvolatile memory (default): `1` or `display.SAVE`

Details

After adding code to the load menu, you can run it from the front panel by pressing the **LOAD** key, then selecting **USER** to select from the available code to load. Pressing the **RUN** key then runs the script.

You can add items in any order. They are always displayed in alphabetical order when the menu is selected.

Any Lua code can be included in the *code* parameter. If *memory* is set to `display.SAVE`, the entry (name and code) is saved in nonvolatile memory. Scripts, functions, and variables used in the code are not saved by `display.SAVE`. Functions and variables need to be saved with the code. If the code is not saved in nonvolatile memory, it is lost when the Model 707B or 708B is turned off. See **Example 2** below.

If you do not make a selection for *memory*, the code is automatically saved to nonvolatile memory.

NOTE

You can create a script that defines several functions, and then use the `display.loadmenu.add()` command to add items that call those individual functions. This allows the operator to run tests from the front panel.

Example 1

```
display.loadmenu.add("Test9", "Test9()")
```

Assume a user script named `Test9` is loaded into the runtime environment. Adds the menu entry to the USER TESTS menu to run the script after loading.

Example 2

```
display.loadmenu.add("Test", "DUT1() beeper.beep(2, 500)", display.SAVE)
```

Assume a script with a function named `DUT1` is loaded into the instrument and the script has not been saved in nonvolatile memory.

Now assume you want to add a test named `Test` to the USER TESTS menu. You want the test to run the function named `DUT1` and sound the beeper. This example adds `Test` to the menu, defines the code, and then saves the `displayName` and code in nonvolatile memory.

When `Test` is run from the front-panel USER TESTS menu, the function named `DUT1` executes and the beeper beeps for two seconds.

Now assume you turn off instrument power. Because the script was not saved in nonvolatile memory, the function named `DUT1` is lost when you turn the instrument on. When `Test` is run again from the front panel, an error is generated because `DUT1` no longer exists in the instrument as a function.

Example 3

```
display.loadmenu.add("Part1", "testpart([[Part1]], 5.0)", display.SAVE)
```

Adds an entry called `Part1` to the front-panel USER TESTS load menu for the code `testpart([[Part1]], 5.0)` and saves it in nonvolatile memory.

Also see

display.loadmenu.delete() (on page 7-77)

display.loadmenu.catalog()

This function creates an iterator for the user menu items accessed using the LOAD key on the front panel.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
for displayName in display.loadmenu.catalog() do body end
for displayName, code in display.loadmenu.catalog() do body end
```

displayName	The name displayed in the menu
code	The code associated with the *displayName*
body	The body of the code to process the entries in the loop

Details

Each time through the loop, *displayName* and *code* take on the values in the USER TESTS menu.

The instrument goes through the list in random order.

Example

```
for displayName, code in display.loadmenu.catalog() do
   print(displayName, code)
end
```

Output:

```
Test DUT1() beeper.beep(2, 500)
Part1 testpart([[Part1]], 5.0)
Test9 Test9()
```

Also see

display.loadmenu.add() (on page 7-74)
display.loadmenu.delete() (on page 7-77)

display.loadmenu.delete()

This function removes an entry from the USER TESTS menu, which can be accessed using the **LOAD** key on the front panel.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
display.loadmenu.delete("displayName")
```

displayName	The name of the entry to be deleted from the USER TESTS menu

Details

If you delete an entry from the USER TESTS menu, you can no longer run it by pressing the **LOAD** key.

Example

```
display.loadmenu.delete("Test9")
for displayName, code in display.loadmenu.catalog() do
   print(displayName, code)
end
```

Deletes the entry named `Test9`.

Output:

```
Test   DUT1() beeper.beep(2, 500)
Part1   testpart([[Part1]], 5.0)
```

Also see

display.loadmenu.add() (on page 7-74)
display.loadmenu.catalog() (on page 7-76)

display.locallockout

This attribute describes whether or not the EXIT (LOCAL) key on the instrument front panel is enabled.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Power cycle	Not saved	0 (display.UNLOCK)

Usage

```
lockout = display.locallockout
display.locallockout = lockout
```

lockout	Unlocks EXIT (LOCAL) key: `0` or `display.UNLOCK` Locks out EXIT (LOCAL) key: `1` or `display.LOCK`

Details

Set `display.locallockout` to `display.LOCK` to prevent the user from interrupting remote operation by pressing the EXIT (LOCAL) key.

Set this attribute to `display.UNLOCK` to allow the EXIT (LOCAL) key to interrupt script or remote operation.

Example

```
display.locallockout = display.LOCK
```

Disables the front-panel EXIT (LOCAL) key.

Also see

None

display.menu()

This function presents a menu on the front-panel display.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
selection = display.menu("name", "items")
```

selection	Name of the variable that holds the selected menu item
name	Menu name to display on the top line
items	Menu items to display on the bottom line

Details

The menu consists of the menu name string on the top line and a selectable list of items on the bottom line. The menu items must be a single string with each item separated by whitespace. The name for the top line is limited to 20 characters.

After sending this command, script execution pauses for the operator to select a menu item. An item is selected by rotating the navigation wheel ◷ to place the blinking cursor on the item, and then pressing the navigation wheel ◷ (or the ENTER key). When an item is selected, the text of that selection is returned.

Pressing the EXIT (LOCAL) key does not abort the script while the menu is displayed, but it does return `nil`. The script can be aborted by calling the exit function when `nil` is returned.

Example

```
selection = display.menu("Menu", "Test1 Test2 Test3")
print(selection)
```

Displays a menu with three menu items. If the second menu item is selected, selection is given the value `Test2`.

Output:

`Test2`

Also see

None

display.prompt()

This function prompts the user to enter a parameter from the front panel of the instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
display.prompt("format", "units", "help")
display.prompt("format", "units", "help", default)
display.prompt("format", "units", "help", default, minimum)
display.prompt("format", "units", "help", default, minimum, maximum)
```

format	A string that defines how the input field is formatted; see **Details** for more information
units	Set the units text string for the top line (eight characters maximum); this indicates the units (for example, "V" or "A") for the value
help	Text string to display on the bottom line (32 characters maximum)
default	The value that is shown when the value is first displayed
minimum	The minimum input value that can be entered
maximum	The maximum input value that can be entered (must be more than minimum)

Details

This function creates an editable input field at the present cursor position, and an input prompt message on the bottom line. Example of a displayed input field and prompt:

`0.00V`

`Input 0 to +2V`

The *format* parameter uses zeros (0), the decimal point, polarity sign, and exponents to define how the input field is formatted.

The *format* parameter can include the options shown in the following table.

Option	Description	Examples
E	Include the E to display the value exponentially. Include a plus sign (+) for positive/negative exponent entry. Do not include the plus sign (+) to prevent negative value entry. 0 defines the digit positions for the exponent.	`0.00000E+0`
+	Allows operators to enter positive or negative values. If the plus sign (+) is not included, the operator cannot enter a negative value.	`+0.00`
0	Defines the digit positions for the value. You can use up to six zeros (0).	`+00.0000E+00`
.	The decimal point where needed for the value.	`+0.00`

You can use the *minimum* and *maximum* parameters to limit the values that can be entered. When a plus sign (+) is not selected for *format*, the minimum limit must be greater than or equal to zero (0). When limits are used, the operator cannot enter values above or below these limits.

The input value is limited to ±1e37.

After sending this command, script execution pauses for the operator to enter a value and press **ENTER**.

For positive and negative entry (plus sign (+) used for the value field and the exponent field), polarity of a nonzero value or exponent can be toggled by positioning the cursor on the polarity sign and turning the navigation wheel ◔. Polarity also toggles when using the navigation wheel ◔ to decrease or increase the value or exponent past zero. A zero value or exponent (for example, +00) is always positive and cannot be toggled to negative polarity.

After executing this command and pressing the **EXIT (LOCAL)** key, the value returns `nil`.

Example

```
value = display.prompt("0.00", "V", "Input 0 to +2V", 0.5, 0, 2)
print(value)
```

The above command prompts the operator to enter a voltage value. The valid input range is 0 to +2.00, with a default of 0.50:

```
0.50V
Input 0 to +2V
```

If the operator enters 0.70, the output is:

```
7.0000000e-01
```

Also see

display.inputvalue() (on page 7-73)

display.screen

This attribute contains the selected display screen.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset	Create configuration script	display.MAIN

Usage

```
displayID = display.screen
display.screen = displayID
```

displayID	One of the following values: ■ Displays the main screen: 1 or `display.MAIN` ■ Displays the user screen: 2 or `display.USER`

Details

Setting this attribute selects the display screen for the front panel. This performs the same action as pressing the DISPLAY key on the front panel. The text for the display screen is set by `display.settext()`.

Read this attribute to determine which of the available display screens was last selected.

NOTE

This does not support the CLOSED CHANNELS option that is available from the DISPLAY key.

Example

`display.screen = display.USER`
Selects the user display.

Also see

display.settext() (on page 7-84)

display.sendkey()

This function sends a code that simulates the action of a front-panel control.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
display.sendkey(keyCode)
```

keyCode	A parameter that specifies the key press to simulate; see **Details** for more information

Details

This command simulates pressing a front-panel key or navigation wheel or turning the navigation wheel one click to the left or right.

The following table lists the *keyCode* value for each front-panel action.

Key codes

Value	Key list		Value	Key list
66	`display.KEY_DELETE`		83	`display.KEY_RUN`
67	`display.KEY_EXIT`		84	`display.KEY_TRIG`
69	`display.KEY_CLOSE`		86	`display.KEY_STEP`
70	`display.KEY_SLOT`		87	`display.KEY_CHAN`
72	`display.KEY_DISPLAY`		90	`display.KEY_INSERT`
74	`display.KEY_ENTER`		91	`display.KEY_MENU`
76	`display.KEY_LOAD`		93	`display.KEY_OPEN`
77	`display.KEY_SCAN`		94	`display.KEY_PATT`
79	`display.KEY_OPENALL`		97	`display.WHEEL_ENTER`
80	`display.KEY_CONFIG`		107	`display.WHEEL_LEFT`
			114	`display.WHEEL_RIGHT`

NOTE

When using this function, send built-in constants, such as `display.KEY_STEP`, instead of the numeric value, such as `86`. This allows for better forward compatibility with firmware revisions.

Example

```
display.sendkey(display.KEY_RUN)
```

Simulates pressing the RUN key.

Also see

None

display.setcursor()

This function sets the position of the cursor.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
display.setcursor(row, column)
display.setcursor(row, column, style)
```

row	The row number for the cursor: 1 or 2
column	The active column position to set: ■ Row 1: 1 to 20 ■ Row 2: 1 to 32
style	Set the cursor: ■ Invisible (default): 0 ■ Blinking: 1

Details

Sending this command selects the user screen and then moves the cursor to the given location.

The `display.clear()`, `display.setcursor()`, and `display.settext()` functions are overlapped commands. That is, the script does not wait for one of these commands to complete. These functions do not immediately update the display. For performance considerations, they update the display as soon as processing time becomes available.

An out-of-range parameter for *row* sets the cursor to row 2. An out-of-range parameter for *column* sets the cursor to column 20 for row 1, or 32 for row 2.

An out-of-range parameter for *style* sets it to 0 (invisible).

A blinking cursor is only visible when it is positioned over displayed text. It cannot be seen when positioned over a space character.

Example

```
display.clear()
display.setcursor(1, 8)
display.settext("Hello")
display.setcursor(2, 14)
display.settext("World")
```

This example displays a message on the front panel, approximately center. Note that the top line of text is larger than the bottom line of text.
The front panel of the instrument displays `Hello` on the top line and `World` on the second line.

Also see

display.clear() (on page 7-67)
display.getcursor() (on page 7-69)
display.gettext() (on page 7-71)
display.screen (on page 7-81)
display.settext() (on page 7-84)

display.settext()

This function displays text on the front-panel user screen.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
display.settext("text")
```

text	Text message to be displayed, with optional character codes

Details

This function selects the user display screen and displays the given text.

After the instrument is turned on, the first time you use a display command to write to the display, the message "User Screen" is cleared. After the first write, you need to use `display.clear()` to clear the message.

The `display.clear()`, `display.setcursor()`, and `display.settext()` functions are overlapped commands. That is, the script does not wait for one of these commands to complete. These functions do not immediately update the display. For performance considerations, they update the display as soon as processing time becomes available.

The text starts at the present cursor position. After the text is displayed, the cursor is after the last character in the display message.

Top line text does not wrap to the bottom line of the display automatically. Any text that does not fit on the present line is truncated. If the text is truncated, the cursor remains at the end of the line.

The text remains on the display until replaced or cleared.

The character codes described in the following table can be included in the text string.

Display character codes

Character Code	Description
`$N`	Newline, starts text on the next line; if the cursor is already on line 2, text is ignored after the `$N` is received
`$R`	Sets text to normal intensity, nonblinking
`$B`	Sets text to blink
`$D`	Sets text to dim intensity
`$F`	Sets the text to background blink
`$$`	Escape sequence to display a single dollar symbol (`$`)

Example

```
display.clear()
display.settext("Normal $BBlinking$N")
display.settext("$DDim $FBackgroundBlink$R $$$$ 2 dollars")
```

This example sets the display to:
`Normal Blinking`
`Dim BackgroundBlink $$ 2 dollars`
with the named effect on each word.

Also see

display.clear() (on page 7-67)
display.getcursor() (on page 7-69)
display.gettext() (on page 7-71)
display.screen (on page 7-81)
display.setcursor() (on page 7-83)

display.trigger.EVENT_ID

This constant is the event ID of the event generated when the front-panel TRIG key is pressed.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Constant	Yes			

Usage

```
eventID = display.trigger.EVENT_ID
```

eventID	The trigger event number

Details

Set the stimulus of any trigger event detector to the value of this constant to have it respond to front -panel trigger key events.

Example

```
scan.trigger.channel.stimulus = display.trigger.EVENT_ID
```

Have the channel action of the trigger model be paced by a user pressing the front-panel TRIG key.

Also see

scan.trigger.channel.stimulus (on page 7-158)

display.waitkey()

This function captures the key code value for the next front-panel action.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
keyCode = display.waitkey()
```

keyCode	See **Details** for more information

Details

After you send this function, script execution pauses until a front-panel action occurs, such as pressing a key or the navigation wheel ◔ or turning the navigation wheel ◔. After the action, the value of the key or action is returned.

If the EXIT (LOCAL) key is pressed while this function is waiting for a front-panel action, the script is not aborted.

The following table lists the *keyCode* value for each front-panel action.

Key codes				
Value	**Key list**		**Value**	**Key list**
66	display.KEY_DELETE		84	display.KEY_TRIG
67	display.KEY_EXIT		86	display.KEY_STEP
69	display.KEY_CLOSE		87	display.KEY_CHAN
70	display.KEY_SLOT		90	display.KEY_INSERT
72	display.KEY_DISPLAY		91	display.KEY_MENU
74	display.KEY_ENTER		93	display.KEY_OPEN
76	display.KEY_LOAD		94	display.KEY_PATT
77	display.KEY_SCAN		97	display.WHEEL_ENTER
79	display.KEY_OPENALL		107	display.WHEEL_LEFT
80	display.KEY_CONFIG		114	display.WHEEL_RIGHT
83	display.KEY_RUN			

NOTE

When using this function, send built-in constants, such as `display.KEY_RUN`, instead of the numeric value of `83`. This allows for better forward compatibility with firmware revisions.

Example

```
key = display.waitkey()
print(key)
```

Pause script execution until the operator presses a key or the navigation wheel or rotates the navigation wheel.

Also see

display.getlastkey() (on page 7-70)
display.sendkey() (on page 7-82)
display.settext() (on page 7-84)

errorqueue.clear()

This function clears all entries out of the error queue.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
errorqueue.clear()
```

Details

See Error Available Bit (error or event queue) (on page 12-7) for additional information about the error queue.

Also see

errorqueue.count (on page 7-87)
errorqueue.next() (on page 7-88)

errorqueue.count

This attribute gets the number of entries in the error queue.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Power cycle Clearing error queue Reading error messages	Not applicable	Not applicable

Usage

```
count = errorqueue.count
```

count	The number of entries in the error queue

Example

```
count = errorqueue.count
print(count)
```

Returns the number of entries in the error queue.
The following output indicates that there are four entries in the error queue:
`4.0000000e+00`

Also see

errorqueue.clear() (on page 7-87)
errorqueue.next() (on page 7-88)

errorqueue.next()

This function reads the oldest entry from the error queue and removes it from the queue.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
errorCode, message, severity, errorNode = errorqueue.next()
```

errorCode	The error code number for the entry
message	The message that describes the error code
severity	The severity level (0, 10, 20, 30, or 40); see **Details** for more information
errorNode	The node number where the error originated

Details

Entries are stored in a first-in, first-out (FIFO) queue. This function reads the oldest entry and removes it from the queue.

Error codes and messages are listed in the Error summary list (on page 11-1).

If there are no entries in the queue, code 0, `Queue is Empty`, is returned.

Returned severity levels are described in the following table.

Severity level descriptions		
Number	**Level**	**Description**
0	**Informational**	Indicates that there are no entries in the queue
10	**Informational**	Indicates a status message or minor error
20	**Recoverable**	Indicates possible invalid user input; operation continues but you should take action to correct the error
30	**Serious**	Indicates a serious error that may require technical assistance, such as corrupted data
40	**Fatal**	Instrument is not operational

In an expanded system, each TSP-Link enabled instrument is assigned a node number. The variable *errorNode* stores the node number where the error originated.

Example

```
errorcode, message = errorqueue.next()
print(errorcode, message)
```

Reads the oldest entry in the error queue. The following output indicates that the queue is empty.
Output:
`0.0000000e+00 Queue Is Empty`

Also see

errorqueue.clear() (on page 7-87)
errorqueue.count (on page 7-87)

eventlog.all()

This function returns all entries from the event log as a single string and removes them from the event log.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
logString = eventlog.all()
```

logString	A listing of all event log entries

Details

This function returns all events in the event log. Logged items are shown from oldest to newest. The response is a string that has the messages delimited with a newline character.

This function also clears the event log.

If there are no entries in the event log, this function returns the value `nil`.

Example

```
print(eventlog.all())
```

Get and print all entries from the event log and remove the entries from the log.
Output:

```
17:26:35.690 10 Oct 2019, LAN0, 192.168.1.102, LXI, 0, 1570728395,
      1192037155.733269000, 0, 0x0
17:26:39.009 10 Oct 2019, LAN5, 192.168.1.102, LXI, 0, 1570728399,
      1192037159.052777000, 0, 0x0
```

Also see

eventlog.clear() (on page 7-90)
eventlog.count (on page 7-90)
eventlog.enable (on page 7-91)
eventlog.next() (on page 7-92)
eventlog.overwritemethod (on page 7-93)

eventlog.clear()

This function clears the event log.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
eventlog.clear()
```

Details

This command removes all messages from the event log.

Also see

eventlog.all() (on page 7-89)
eventlog.count (on page 7-90)
eventlog.enable (on page 7-91)
eventlog.next() (on page 7-92)
eventlog.overwritemethod (on page 7-93)

eventlog.count

This attribute returns the number of unread events in the event log.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Instrument reset Clearing event log Reading event log	Not applicable	Not applicable

Usage

```
N = eventlog.count
```

N	The number of events in the event log

Example

```
print(eventlog.count)
```

Displays the present number of events in the instrument event log.
Output looks similar to:
`3.0000000e+00`

Also see

eventlog.all() (on page 7-89)
eventlog.clear() (on page 7-90)
eventlog.enable (on page 7-91)
eventlog.next() (on page 7-92)
eventlog.overwritemethod (on page 7-93)

eventlog.enable

This attribute enables or disables the event log.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset	Create configuration script	eventlog.ENABLE

Usage

```
status = eventlog.enable
eventlog.enable = status
```

status	The enable status of the event log: ■ Enable: `1` or `eventlog.ENABLE` ■ Disable: `0` or `eventlog.DISABLE`

Details

When the event log is disabled (`eventlog.DISABLE` or `0`), no new events are added to the event log. You can, however, read and remove existing events.

When the event log is enabled, new events are logged.

Example

```
print(eventlog.enable)
eventlog.enable = eventlog.DISABLE
print(eventlog.enable)
```

Displays the present status of the Model 707B or 708B event log.
Output:
```
1.0000000e+00
0.0000000e+00
```

Also see

eventlog.all() (on page 7-89)
eventlog.clear() (on page 7-90)
eventlog.count (on page 7-90)
eventlog.next() (on page 7-92)
eventlog.overwritemethod (on page 7-93)

eventlog.next()

This function returns the oldest unread event message from the event log and removes it from the event log.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
logString = eventlog.next()
```

logString	The next log entry

Details

Returns the next entry from the event log and removes it from the log.

If there are no entries in the event log, returns the value `nil`.

Example 1

```
print(eventlog.next())
```

Get the oldest message in the event log and remove that entry from the log.
Output:
```
17:28:22.085 10 Oct 2019, LAN2, 192.168.1.102, LXI, 0, 1570728502, <no time>, 0,
    0x0
```

Example 2

```
print(eventlog.next())
```

If there is nothing in the event log, you get the following output:
```
nil
```

Also see

eventlog.all() (on page 7-89)
eventlog.clear() (on page 7-90)
eventlog.count (on page 7-90)
eventlog.enable (on page 7-91)
eventlog.overwritemethod (on page 7-93)

eventlog.overwritemethod

This attribute controls how the event log processes events if the event log is full.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup	Create configuration script	1 (eventlog.DISCARD_OLDEST)

Usage

```
method = eventlog.overwritemethod
eventlog.overwritemethod = method
```

method	Set to one of the following values: ▪ New entries are not logged: `0` or `eventlog.DISCARD_NEWEST` ▪ Old entries are deleted as new events are logged: `1` or `eventlog.DISCARD_OLDEST`

Details

When this attribute is set to `eventlog.DISCARD_NEWEST`, new entries are not logged.

When this attribute is set to `eventlog.DISCARD_OLDEST`, the oldest entry is discarded when a new entry is added.

Example

```
eventlog.overwritemethod = 0
```

When the log is full, the event log ignores new entries.

Also see

eventlog.all() (on page 7-89)
eventlog.clear() (on page 7-90)
eventlog.count (on page 7-90)
eventlog.enable (on page 7-91)
eventlog.next() (on page 7-92)

exit()

This function stops a script that is presently running.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
exit()
```

Details

Terminates script execution when called from a script that is being executed.

This command does not wait for overlapped commands to complete before terminating script execution. If overlapped commands are required to finish, use the `waitcomplete()` function before calling `exit()`.

Also see

waitcomplete() (on page 7-253)

format.asciiprecision

This attribute sets the precision (number of digits) for all numbers returned in the ASCII format.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	No	Instrument reset Recall setup	Create configuration script	10

Usage

```
precision = format.asciiprecision
format.asciiprecision = precision
```

precision	A number representing the number of digits to be printed for numbers printed with the `print()` and `printnumber()` functions; must be a number between `1` and `16`

Details

This attribute specifies the precision (number of digits) for numeric data printed with the `print()` and `printnumber()` functions. The `format.asciiprecision` attribute is only used with the ASCII format. The precision value must be a number from 0 to 16.

The precision is the number of significant digits printed. There is always one digit to the left of the decimal point. Be sure to include this digit when setting the precision.

Example

`format.asciiprecision = 10`
`x = 2.54`
`printnumber(x)`
`format.asciiprecision = 3`
`printnumber(x)`
	Output:
	`2.54000000000e+00`
	
	`2.54e+00`

Also see

format.byteorder (on page 7-95)
format.data (on page 7-96)
print() (on page 7-140)
printnumber() (on page 7-141)

format.byteorder

This attribute sets the binary byte order for the data that is printed using the `printnumber()` function.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup	Create configuration script Save setup	format.LITTLEENDIAN

Usage

```
order = format.byteorder
format.byteorder = order
```

order	Byte order value as follows: ■ Most significant byte first: `0`, `format.NORMAL`, `format.NETWORK`, or `format.BIGENDIAN` ■ Least significant byte first: `1`, `format.SWAPPED` or `format.LITTLEENDIAN`

Details

This attribute selects the byte order in which data is written when you are printing data values with the `printnumber()` function. The byte order attribute is only used with the `format.SREAL`, `format.REAL`, `format.REAL32`, and `format.REAL64` data formats.

`format.NORMAL`, `format.BIGENDIAN`, and `format.NETWORK` select the same byte order. `format.SWAPPED` and `format.LITTLEENDIAN` select the same byte order. Selecting which to use is a matter of preference.

Select the `format.SWAPPED` or `format.LITTLEENDIAN` byte order when sending data to a computer with a Microsoft Windows operating system.

Example

```
x = 1.23
format.data = format.REAL32
format.byteorder = format.LITTLEENDIAN
printnumber(x)
format.byteorder = format.BIGENDIAN
printnumber(x)
```

The output depends on the terminal program you use, but it looks something like:

```
#0¤p??
#0??p¤
```

Also see

format.asciiprecision (on page 7-94)
format.data (on page 7-96)
printnumber() (on page 7-141)

format.data

This attribute sets the data format for data that is printed using the `printnumber()` function.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	No	Instrument reset Recall setup	Create configuration script	1 (format.ASCII)

Usage

```
value = format.data
format.data = value
```

value	The format to use for data, set to one of the following values: ■ ASCII format: `1` or `format.ASCII` ■ Single-precision IEEE Std 754 binary format: `2`, `format.SREAL`, or `format.REAL32` ■ Double-precision IEEE Std 754 binary format: `3`, `format.REAL`, `format.REAL64`, or `format.DREAL`

Details

The precision of numeric values can be controlled with the `format.asciiprecision` attribute. The byte order of `format.SREAL`, `format.REAL`, `format.REAL32`, and `format.REAL64` can be selected with the `format.byteorder` attribute.

REAL32 and SREAL select the same single precision format. REAL and REAL64 select the same double precision format. They are alternative identifiers. Selecting which to use is a matter of preference.

The IEEE Std 754 binary formats use four bytes for single-precision values and eight bytes for double-precision values.

When data is written with any of the binary formats, the response message starts with `#0` and ends with a new line. When data is written with the ASCII format, elements are separated with a comma and space.

NOTE

Binary formats are not intended to be interpreted by humans.

Example

```
format.asciiprecision = 10
x = 3.14159265
format.data = format.ASCII
printnumber(x)
format.data = format.REAL64
printnumber(x)
```

Output a number represented by `x` in ASCII using a precision of 10, then output the same number in binary using double-precision format.

```
Output:
3.141592650e+00
#0ñÔÈSû!   @
```

Also see

format.asciiprecision (on page 7-94)
format.byteorder (on page 7-95)
printnumber() (on page 7-141)

gettimezone()

This function retrieves the local time zone.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

timeZone = gettimezone()

timeZone	The local time zone of the instrument

Details

See `settimezone()` for additional details about the time zone format and a description of the fields.

timeZone can be in either of the following formats:

- If one parameter was used with `settimezone()`, the format used is: `GMThh:mm:ss`
- If four parameters were used with `settimezone()`, the format used is: `GMThh:mm:ssGMThh:mm:ss,Mmm.w.dw/hh:mm:ss,Mmm.w.dw/hh:mm:ss`

Example

```
timezone = gettimezone()
```

Reads the value of the local time zone.

Also see

settimezone() (on page 7-170)

gpib.address

This attribute contains the GPIB address.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	No	Not applicable	Nonvolatile memory	16

Usage

```
address = gpib.address
gpib.address = address
```

address	The GPIB address of the instrument: 1 to 30

Details

The address can be set to any address value from 1 to 30. However, the address must be unique in the system. It cannot conflict with an address that is assigned to another instrument or to the GPIB controller.

A new GPIB address takes effect when the command to change it is processed. If there are response messages in the output queue when this command is processed, they must be read at the new address.

If command messages are being queued (sent before this command has executed), the new settings may take effect in the middle of a subsequent command message, so use care when setting this attribute from the GPIB interface.

Allow sufficient time for the command to be processed before attempting to communicate with the instrument again.

The `reset()` function does not affect the GPIB address.

Example

```
gpib.address = 26
address = gpib.address
print(address)
```

Sets the GPIB address and reads the address.
Output:
`26`

Also see

GPIB setup (on page 4-23)

lan.applysettings()

This function re-initializes the LAN interface with new settings.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
lan.applysettings()
```

Details

Disconnects all existing LAN connections to the instrument and re-initializes the LAN with the present configuration settings.

This function initiates a background operation. LAN configuration can be a lengthy operation. Although the function returns immediately, the LAN initialization continues to run in the background.

Even though the LAN configuration settings may not have changed since the LAN was last connected, new settings may take effect due to the dynamic nature of dynamic host configuration protocol (DHCP) or dynamic link local addressing (DLLA) configuration.

Re-initialization takes effect even if the configuration has not changed since the last time the instrument connected to the LAN.

Example

`lan.applysettings()`
Re-initialize the LAN interface with new settings.

Also see

None

lan.config.dns.address[N]

This command configures the DNS server IP addresses.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	LAN restore defaults	Nonvolatile memory	"0.0.0.0"

Usage

```
dnsAddress = lan.config.dns.address[N]
lan.config.dns.address[N] = "dnsAddress"
```

dnsAddress	DNS server IP address
N	Entry index: 1 or 2

Details

This attribute is an array of Domain Name System (DNS) server addresses. These addresses take priority for DNS lookups and are consulted before any server addresses that are obtained using DHCP. This allows local DNS servers to be specified that take priority over DHCP-configured global DNS servers.

You can specify up to two addresses. The address specified by `1` is consulted first for DNS lookups. *`dnsAddress`* must be a string specifying the IP address of the DNS server in dotted decimal notation.

Unused entries are returned as `0.0.0.0` when read. To disable an entry, set its value to `0.0.0.0` or the empty string `""`.

Although only two addresses may be manually specified here, the instrument uses up to three DNS server addresses. If two are specified here, only one that is given by a DHCP server is used. If no entries are specified here, up to three addresses that are given by a DHCP server are used.

Example

```
dnsaddress = "164.109.48.173"
lan.config.dns.address[1] = dnsaddress
```

Set the DNS address 1 to `164.109.48.173`.

Also see

lan.config.dns.domain (on page 7-100)
lan.config.dns.dynamic (on page 7-101)
lan.config.dns.hostname (on page 7-102)
lan.config.dns.verify (on page 7-103)
lan.restoredefaults() (on page 7-108)

lan.config.dns.domain

This command configures the dynamic DNS domain.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	LAN restore defaults	Nonvolatile memory	""

Usage

```
domain = lan.config.dns.domain
lan.config.dns.domain = "domain"
```

domain	Dynamic DNS registration domain; a string of 255 characters or less

Details

This attribute holds the domain to request during dynamic DNS registration. Dynamic DNS registration works with DHCP to register the domain specified in this attribute with the DNS server.

The length of the fully qualified host name (combined length of the domain and host name with separator characters) must be less than or equal to 255 characters. Although up to 255 characters are allowed, you must make sure the combined length is also no more than 255 characters.

Example

`print(lan.config.dns.domain)`
Outputs the present dynamic DNS domain. For example, if the domain is `Matrix`, the response is: `Matrix`

Also see

lan.config.dns.dynamic (on page 7-101)
lan.config.dns.hostname (on page 7-102)
lan.config.dns.verify (on page 7-103)
lan.restoredefaults() (on page 7-108)

lan.config.dns.dynamic

Enables or disables the dynamic DNS registration.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	LAN restore defaults	Nonvolatile memory	1 (lan.ENABLE)

Usage

```
state = lan.config.dns.dynamic
lan.config.dns.dynamic = state
```

state	The dynamic DNS registration state; it may be one of the following values: ■ Enabled: `1` or `lan.ENABLE` ■ Disabled: `0` or `lan.DISABLE`

Details

Dynamic DNS registration works with DHCP to register the host name with the DNS server. The host name is specified in the `lan.config.dns.hostname` attribute.

Example

`print(lan.config.dns.dynamic)`
Outputs the dynamic registration state. If dynamic DNS registration is enabled, the response is: `1.0000000e+00`

Also see

lan.config.dns.hostname (on page 7-102)
lan.restoredefaults() (on page 7-108)

lan.config.dns.hostname

This attribute defines the dynamic DNS host name.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Not applicable	Nonvolatile memory	Instrument specific (see **Details**)

Usage

```
hostName = lan.config.dns.hostname
lan.config.dns.hostname = "hostName"
```

hostName	The host name to use for dynamic DNS registration; the host name must be: ▪ A string ▪ 15 characters or less ▪ Start with a letter ▪ End with a letter or digit ▪ Contain only letters, digits, and hyphens

Details

This attribute holds the host name to request during dynamic DNS registration. Dynamic DNS registration works with DHCP to register the host name specified in this attribute with the DNS server.

The factory default value for *hostName* is `"K-<model number>-<serial number>"`, where `<model number>` and `<serial number>` are replaced with the actual model number and serial number of the instrument (for example, `"K-707B-1234567"`). The characters of *hostName* are separated by hyphens.

The length of the fully qualified host name (combined length of the domain and host name with separator characters) must be less than or equal to 255 characters. Although up to 15 characters can be entered here, you must make sure the combined length is no more than 255 characters.

Setting this attribute to an empty string (in other words, setting this attribute to a string of length zero or a string that consists entirely of whitespace characters) reverts the host name to the factory default value.

Example

```
print(lan.config.dns.hostname)
```

Outputs the present dynamic DNS host name.

Also see

lan.config.dns.domain (on page 7-100)
lan.config.dns.dynamic (on page 7-101)
lan.restoredefaults() (on page 7-108)

lan.config.dns.verify

This attribute defines the DNS host name verification state.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	LAN restore defaults	Nonvolatile memory	1 (lan.ENABLE)

Usage

```
state = lan.config.dns.verify
lan.config.dns.verify = state
```

state	DNS hostname verification state: ▪ DNS host name verification enabled: `1` or `lan.ENABLE` ▪ DNS host name verification disabled: `0` or `lan.DISABLE`

Details

When this is enabled, the instrument performs DNS lookups to verify that the DNS host name matches the value specified by `lan.config.dns.hostname`.

Example

`print(lan.config.dns.verify)`
Outputs the present DNS host name verification state. If it is enabled, the output is: `1.0000000e+00`

Also see

lan.config.dns.hostname (on page 7-102)
lan.restoredefaults() (on page 7-108)

lan.config.gateway

This attribute contains the LAN default gateway address.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	LAN restore defaults	Nonvolatile memory	"0.0.0.0"

Usage

```
gatewayAddress = lan.config.gateway
lan.config.gateway = "gatewayAddress"
```

gatewayAddress	LAN default gateway address; must be a string specifying the default IP address of the gateway in dotted decimal notation

Details

This attribute specifies the default gateway IP address to use when manual or DLLA configuration methods are used to configure the LAN. If DHCP is enabled, this setting is ignored.

This attribute does not indicate the actual setting that is presently in effect. Use the `lan.status.gateway` attribute to determine the present operating state of the LAN.

The IP address must be formatted in four groups of numbers, each separated by a decimal.

Example

```
print(lan.config.gateway)
```

Outputs the default gateway address. For example, you might see the output:
`192.168.0.1`

Also see

lan.restoredefaults() (on page 7-108)
lan.status.gateway (on page 7-110)

lan.config.ipaddress

This command specifies the LAN IP address.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	LAN restore defaults	Nonvolatile memory	"192.168.0.2"

Usage

```
ipAddress = lan.config.ipaddress
lan.config.ipaddress = "ipAddress"
```

ipAddress	LAN IP address; must be a string specifying the IP address in dotted decimal notation

Details

This command specifies the LAN IP address to use when the LAN is configured using the manual configuration method. This setting is ignored when DLLA or DHCP is used.

This attribute does not indicate the actual setting that is presently in effect. Use the `lan.status.ipaddress` attribute to determine the present operating state of the LAN.

Example

```
ipaddress = lan.config.ipaddress
```

Retrieves the presently set LAN IP address.

Also see

lan.restoredefaults() (on page 7-108)
lan.status.ipaddress (on page 7-111)

lan.config.method

This attribute contains the LAN settings configuration method.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	LAN restore defaults	Nonvolatile memory	0 (lan.AUTO)

Usage

```
method = lan.config.method
lan.config.method = method
```

method	The method for configuring LAN settings; it can be one of the following values: ■ Selects automatic sequencing of configuration methods: `0` or `lan.AUTO` ■ Use only manually specified configuration settings: `1` or `lan.MANUAL`

Details

This attribute controls how the LAN IP address, subnet mask, default gateway address, and DNS server addresses are determined.

When method is `lan.AUTO`, the instrument first attempts to configure the LAN settings using dynamic host configuration protocol (DHCP). If DHCP fails, it tries dynamic link local addressing (DLLA). If DLLA fails, it uses the manually specified settings.

When method is `lan.MANUAL`, only the manually specified settings are used. Neither DHCP nor DLLA are attempted.

Example

```
print(lan.config.method)
```

Outputs the present method.
For example:
`1.0000000e+00`

Also see

lan.restoredefaults() (on page 7-108)

lan.config.subnetmask

This attribute contains the LAN subnet mask.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	LAN restore defaults	Nonvolatile memory	"255.255.255.0"

Usage

```
mask = lan.config.subnetmask
lan.config.subnetmask = "mask"
```

mask	String that specifies the LAN subnet mask value in dotted decimal notation

Details

This attribute specifies the LAN subnet mask that is used when the manual configuration method is used to configure the LAN. This setting is ignored when DLLA or DHCP is used.

This attribute does not indicate the actual setting presently in effect. Use the `lan.status.subnetmask` attribute to determine the present operating state of the LAN.

Example

`print(lan.config.subnetmask)`
Outputs the LAN subnet mask, such as: `255.255.255.0`

Also see

lan.restoredefaults() (on page 7-108)
lan.status.subnetmask (on page 7-115)

lan.lxidomain

This attribute contains the LXI domain.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	LAN restore defaults	Nonvolatile memory	0

Usage

```
domain = lan.lxidomain
lan.lxidomain = domain
```

domain	The LXI domain number: 0 to 255

Details

This attribute sets the LXI domain number.

All outgoing LXI packets are generated with this domain number. All inbound LXI packets are ignored unless they have this domain number.

Example

`print(lan.lxidomain)`
Displays the LXI domain.

Also see

None

lan.nagle

This attribute controls the state of the LAN Nagle algorithm.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Power cycle	Not saved	0 (lan.DISABLE)

Usage

```
state = lan.nagle
lan.nagle = state
```

state	The state of the Nagle algorithm: ■ Enable the LAN Nagle algorithm for TCP connections: `1` or `lan.ENABLE` ■ Disable the Nagle algorithm for TCP connections: `0` or `lan.DISABLE`

Details

This attribute enables or disables the use of the LAN Nagle algorithm on transmission control protocol (TCP) connections.

Also see

lan.restoredefaults() (on page 7-108)

lan.reset()

This function resets the LAN interface.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
lan.reset()
```

Details

This function resets the LAN interface. It performs the commands `lan.restoredefaults()` and `lan.applysettings()`. It also resets the LAN password.

Also see

lan.applysettings() (on page 7-99)
lan.restoredefaults() (on page 7-108)
localnode.password (on page 7-128)

lan.restoredefaults()

This function resets LAN settings to default values.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
lan.restoredefaults()
```

Details

The settings that are restored are shown in the following table.

Settings that are restored to default

Attribute	Default setting
`lan.config.dns.address[N]`	`"0.0.0.0"`
`lan.config.dns.domain`	`""`
`lan.config.dns.dynamic`	`lan.ENABLE`
`lan.config.dns.hostname`	`"K-<model number>-<serial number>"`
`lan.config.dns.verify`	`lan.ENABLE`
`lan.config.gateway`	`"0.0.0.0"`
`lan.config.ipaddress`	`"0.0.0.0"`
`lan.config.method`	`lan.AUTO`
`lan.config.subnetmask`	`"255.255.255.0"`
`lan.lxidomain`	`0`
`localnode.password`	`"admin"`

This command is run when `lan.reset()` is sent.

Example

`lan.restoredefaults()`
Restores the LAN defaults.

Also see

lan.reset() (on page 7-107)
localnode.password (on page 7-128)

lan.status.dns.address[N]

This attribute contains the DNS server IP addresses.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

dnsAddress = lan.status.dns.address[*N*]

dnsAddress	DNS server IP address
N	Entry index: 1, 2, or 3

Details

This attribute is an array of Domain Name System (DNS) server addresses. The instrument can use up to three addresses.

Unused or disabled entries are returned as "0.0.0.0" when read. The *dnsAddress* returned is a string specifying the IP address of the DNS server in dotted decimal notation.

The value of lan.status.dns.address[1] is referenced first for all DNS lookups. The values of lan.status.dns.address[2] and lan.status.dns.address[3] are referenced second and third, respectively.

Example

```
print(lan.status.dns.address[1])
```

Outputs DNS server address 1, for example:
164.109.48.173

Also see

lan.config.dns.address[N] (on page 7-99)
lan.status.dns.name (on page 7-109)

lan.status.dns.name

This attribute contains the present DNS fully qualified host name.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

hostName = lan.status.dns.name

hostName	Fully qualified DNS host name that can be used to connect to the instrument

Details

A fully qualified domain name (FQDN) specifies its exact location in the tree hierarchy of the Domain Name System (DNS).

A FQDN is the complete domain name for a specific computer or host on the LAN. The FQDN consists of two parts: The host name and the domain name.

If the DNS host name for an instrument is not found, this attribute stores the IP address in dotted decimal notation.

Example

`print(lan.status.dns.name)`	
Outputs the dynamic DNS host name.	

Also see

lan.config.dns.address[N] (on page 7-99)
lan.config.dns.hostname (on page 7-102)

lan.status.duplex

This attribute contains the duplex mode presently in use by the LAN interface.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

duplex = `lan.status.duplex`

duplex	LAN duplex setting can be one of the following values: ▪ Half-duplex operation: `0` or `lan.HALF` ▪ Full-duplex operation: `1` or `lan.FULL`

Example

`print(lan.status.duplex)`
Outputs the present LAN duplex mode, such as: `1.0000000e+00`

Also see

None

lan.status.gateway

This attribute contains the gateway address presently in use by the LAN interface.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

gatewayAddress = `lan.status.gateway`

gatewayAddress	LAN gateway address presently in use

Details

The value of *gatewayAddress* is a string that indicates the IP address of the gateway in dotted decimal notation.

Example

```
print(lan.status.gateway)
```

Outputs the gateway address, such as:
`192.168.0.1`

Also see

lan.config.gateway (on page 7-103)

lan.status.ipaddress

This attribute contains the LAN IP address presently in use by the LAN interface.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
ipAddress = lan.status.ipaddress
```

ipAddress	LAN IP address specified in dotted decimal notation

Details

The IP address is a character string that represents the IP address assigned to the instrument.

Example

```
print(lan.status.ipaddress)
```

Outputs the LAN IP address currently in use, such as:
`192.168.0.2`

Also see

lan.config.ipaddress (on page 7-104)

lan.status.macaddress

This attribute contains the LAN MAC address.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
macAddress = lan.status.macaddress
```

macAddress	The instrument MAC address

Details

The MAC address is a character string representing the MAC address of the instrument in hexadecimal notation. The string includes colons that separate the address octets (see Example).

Example

`print(lan.status.macaddress)`
Outputs the MAC address of the instrument, for example: `08:00:11:00:00:57`

Also see

None

lan.status.port.dst

This attribute contains the LAN dead socket termination (DST) port number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
port = lan.status.port.dst
```

port	DST port number

Details

This attribute holds the TCP port number used to reset all other LAN socket connections.

To reset all LAN connections, open a connection to the DST port number.

Example

`print(lan.status.port.dst)`
Outputs the LAN DST port number, such as: `5.0300000e+03`

Also see

None

lan.status.port.rawsocket

This attribute contains the LAN raw socket connection port number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
port = lan.status.port.rawsocket
```

port	Raw socket port number

Details

The TCP port number is used to connect the instrument and to control the instrument over a raw socket communications interface.

Example

```
print(lan.status.port.rawsocket)
```

Outputs the LAN raw socket port number, such as:
`5.02500e+03`

Also see

None

lan.status.port.telnet

This attribute contains the LAN telnet connection port number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
port = lan.status.port.telnet
```

port	Telnet port number

Details

This attribute holds the TCP port number used to connect to the instrument to control it over a telnet interface.

Example

```
print(lan.status.port.telnet)
```

Get the LAN telnet connection port number.
Output:
`2.3000000e+01`

Also see

None

lan.status.port.vxi11

This attribute contains the LAN VXI-11 connection port number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

port = lan.status.port.vxi11

port	LAN VXI-11 port number

Details

This attribute stores the TCP port number used to connect to the instrument over a VXI-11 interface.

Example

```
print(lan.status.port.vxi11)
```

Outputs the VXI-11 number, such as:
`1.02400e+03`

Also see

None

lan.status.speed

This attribute contains the LAN speed.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

speed = lan.status.speed

speed	LAN speed in Mbps: 10 or 100

Details

This attribute indicates the transmission speed presently in use by the LAN interface.

Example

```
print(lan.status.speed)
```

Outputs the transmission speed of the instrument presently in use, such as:
`1.0000000e+02`

Also see

None

lan.status.subnetmask

This attribute contains the LAN subnet mask that is presently in use by the LAN interface.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
mask = lan.status.subnetmask
```

mask	A string specifying the subnet mask in dotted decimal notation

Details

Use this attribute to determine the present operating state of the LAN. This attribute returns the present LAN subnet mask value if the LAN is manually configured or when DLLA or DHCP is used.

Example

```
print(lan.status.subnetmask)
```

Outputs the subnet mask of the instrument that is presently in use, such as:
`255.255.255.0`

Also see

lan.config.subnetmask (on page 7-105)

lan.trigger[N].assert()

This function simulates the occurrence of the trigger and generates the corresponding event ID.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
lan.trigger[N].assert()
```

N	The LAN event number: 1 to 8

Details

Generates and sends a LAN trigger packet for the LAN event number specified.

Sets the pseudo line state to the appropriate state.

The following indexes provide the listed LXI events:

- 1:LAN0
- 2:LAN1
- 3:LAN2
- …
- 8:LAN7

Example

`lan.trigger[5].assert()`	
Creates a trigger with LAN packet 5.	

Also see

lan.lxidomain (on page 7-106)
lan.trigger[N].clear() (on page 7-116)
lan.trigger[N].mode (on page 7-120)
lan.trigger[N].overrun (on page 7-121)
lan.trigger[N].stimulus (on page 7-123)
lan.trigger[N].wait() (on page 7-125)

lan.trigger[N].clear()

This function clears the event detector for a LAN trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
lan.trigger[N].clear()
```

N	The LAN event number: 1 to 8

Details

The trigger event detector enters the detected state when an event is detected. This function clears a trigger event detector and discards the history of the trigger packet.

This function clears all overruns associated with this LAN trigger.

Example

`lan.trigger[5].clear()`	
Clears the event detector with LAN packet 5.	

Also see

lan.trigger[N].assert() (on page 7-115)
lan.trigger[N].overrun (on page 7-121)
lan.trigger[N].stimulus (on page 7-123)
lan.trigger[N].wait() (on page 7-125)

lan.trigger[N].connect()

This function prepares the event generator for outgoing trigger events.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
lan.trigger[N].connect()
```

N	The LAN event number: 1 to 8

Details

This command prepares the event generator to send event messages. For TCP connections, this opens the TCP connection.

The event generator automatically disconnects when either the protocol or IP address for this event is changed.

Example

```
lan.trigger[1].protocol = lan.MULTICAST
lan.trigger[1].connect()
lan.trigger[1].assert()
```

Set the protocol for LAN trigger 1 to be multicast when sending LAN triggers. Then, after connecting the LAN trigger, send a message on LAN trigger 1 by asserting it.

Also see

lan.trigger[N].assert() (on page 7-115)
lan.trigger[N].ipaddress (on page 7-120)
lan.trigger[N].overrun (on page 7-121)
lan.trigger[N].protocol (on page 7-122)
lan.trigger[N].stimulus (on page 7-123)
lan.trigger[N].wait() (on page 7-125)

lan.trigger[N].connected

This attribute stores the LAN event connection state.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

`connected = lan.trigger[N].connected`

connected	The LAN event connection state: ■ Connected: `true` ■ Not connected: `false`
N	The LAN event number: 1 to 8

Details

This read-only attribute is set to `true` when the LAN trigger is connected and ready to send trigger events following a successful `lan.trigger[N].connect()` command. If the LAN trigger is not ready to send trigger events, this value is `false`.

This attribute is also `false` when either `lan.trigger[N].protocol` or `lan.trigger[N].ipaddress` attributes are changed or the remote connection closes the connection.

Example

```
lan.trigger[1].protocol = lan.MULTICAST
print(lan.trigger[1].connected)
```

Outputs `true` if connected or `false` if not connected.

Example output:

`false`

Also see

lan.trigger[N].connect() (on page 7-117)
lan.trigger[N].ipaddress (on page 7-120)
lan.trigger[N].protocol (on page 7-122)

lan.trigger[N].disconnect()

This function disconnects the LAN trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
lan.trigger[N].disconnect()
```

N	The LAN event number: 1 to 8

Details

For TCP connections, this closes the TCP connection.

The LAN trigger automatically disconnects when either the `lan.trigger[N].protocol` or `lan.trigger[N].ipaddress` attributes for this event are changed.

Also see

lan.trigger[N].ipaddress (on page 7-120)
lan.trigger[N].protocol (on page 7-122)

lan.trigger[N].EVENT_ID

This constant is the event identifier used to route the LAN trigger to other subsystems (using stimulus properties).

Type	TSP-Link accessible	Affected by	Where saved	Default value
Constant	Yes			

Usage

```
lan.trigger[N].EVENT_ID
```

N	The LAN event number: 1 to 8

Details

Set the stimulus of any trigger event detector to the value of this constant to have it respond to incoming LAN trigger packets.

Example

```
digio.trigger[14].stimulus = lan.trigger[1].EVENT_ID
```

Route occurrences of triggers on LAN trigger 1 to digital I/O trigger 14.

Also see

None

lan.trigger[N].ipaddress

This attribute specifies the address of UDP or TCP listeners.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset LAN trigger N reset Recall setup	Create configuration script	"0.0.0.0"

Usage

```
ipAddress = lan.trigger[N].ipaddress
lan.trigger[N].ipaddress = "ipAddress"
```

ipAddress	The LAN address for this attribute as a string in dotted decimal notation
N	The LAN event number: 1 to 8

Details

Sets the IP address for outgoing trigger events.

Set to `"0.0.0.0"` for multicast.

After changing this setting, the `lan.trigger[N].connect()` command must be called before outgoing messages can be sent.

Example

```
lan.trigger[3].protocol = lan.TCP
lan.trigger[3].ipaddress = "192.168.1.100"
lan.trigger[3].connect()
```

Set the protocol for LAN trigger 3 to be `lan.TCP` when sending LAN triggers.
Use IP address `"192.168.1.100"` to connect the LAN trigger.

Also see

lan.trigger[N].connect() (on page 7-117)

lan.trigger[N].mode

This attribute sets the trigger operation and detection mode of the specified LAN event.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset LAN trigger N reset Recall setup	Create configuration script	0 (lan.TRIG_EITHER)

Usage

```
mode = lan.trigger[N].mode
lan.trigger[N].mode = mode
```

mode	A number representing the trigger mode (0 to 7); see the **Details** section for more information
N	A number representing the LAN event number (1 to 8)

Details

This command controls how the trigger event detector and the output trigger generator operate on the given trigger. These settings are intended to provide behavior similar to the digital I/O triggers.

LAN trigger mode values

Mode	Number	Trigger packets detected as input	LAN trigger packet generated for output with a...
`lan.TRIG_EITHER`	0	Rising or falling edge (positive or negative state)	negative state
`lan.TRIG_FALLING`	1	Falling edge (negative state)	negative state
`lan.TRIG_RISING`	2	Rising edge (positive state)	positive state
`lan.TRIG_RISINGA`	3	Rising edge (positive state)	positive state
`lan.TRIG_RISINGM`	4	Rising edge (positive state)	positive state
`lan.TRIG_SYNCHRONOUS`	5	Falling edge (negative state)	positive state
`lan.TRIG_SYNCHRONOUSA`	6	Falling edge (negative state)	positive state
`lan.TRIG_SYNCHRONOUSM`	7	Rising edge (positive state)	negative state

`lan.TRIG_RISING` and `lan.TRIG_RISINGA` are the same.

`lan.TRIG_RISING` and `lan.TRIG_RISINGM` are the same.

Use of either `lan.TRIG_SYNCHRONOUSA` or `lan.TRIG_SYNCHRONOUSM` instead of `lan.TRIG_SYNCHRONOUS` is preferred. Use of `lan.TRIG_SYNCHRONOUS` is provided for compatibility with older products and other Keithley products.

Example

```
print(lan.trigger[1].mode)
```

Outputs the present LAN trigger mode of LAN event 1.

Also see

Digital I/O (on page 3-9)
TSP-Link system expansion interface (on page 6-47)

lan.trigger[N].overrun

This attribute contains the overrun status of the LAN event detector.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	LAN trigger N clear LAN trigger N reset Instrument reset Recall setup	Not applicable	Not applicable

Usage

```
overrun = lan.trigger[N].overrun
```

overrun	The trigger overrun state for the specified LAN packet: `true` or `false`
N	The LAN event number: 1 to 8

Details

This command indicates whether an event has been ignored because the event detector was already in the detected state when the event occurred.

This is an indication of the state of the event detector built into the synchronization line itself. It does not indicate if an overrun occurred in any other part of the trigger model, or in any other construct that is monitoring the event.

It also is not an indication of an output trigger overrun.

Example

```
overrun = lan.trigger[5].overrun
print(overrun)
```

Checks the overrun status of a trigger on LAN5 and outputs the value, such as:
`false`

Also see

lan.trigger[N].assert() (on page 7-115)
lan.trigger[N].clear() (on page 7-116)
lan.trigger[N].stimulus (on page 7-123)
lan.trigger[N].wait() (on page 7-125)

lan.trigger[N].protocol

This attribute sets the LAN protocol to use for sending trigger messages.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset LAN trigger N reset Recall setup	Create configuration script	0 (lan.TCP)

Usage

```
protocol = lan.trigger[N].protocol
lan.trigger[N].protocol = protocol
```

protocol	The protocol to use for messages from the trigger: ■ `0` or `lan.TCP` ■ `1` or `lan.UDP` ■ `2` or `lan.MULTICAST`
N	The LAN event number: 1 to 8

Details

The LAN trigger listens for trigger messages on all supported protocols, but uses the designated protocol for sending outgoing messages. After changing this setting, `lan.trigger[N].connect()` must be called before outgoing event messages can be sent.

When the `lan.MULTICAST` protocol is selected, the `lan.trigger[N].ipaddress` attribute is ignored and event messages are sent to the multicast address 224.0.23.159.

Example

`print(lan.trigger[1].protocol)`
Get LAN protocol to use for sending trigger messages for LAN event 1.

Also see

lan.trigger[N].connect() (on page 7-117)
lan.trigger[N].ipaddress (on page 7-120)

lan.trigger[N].pseudostate

This attribute sets the simulated line state for the LAN trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset LAN trigger N reset Recall setup	Create configuration script	1

Usage

```
pseudostate = lan.trigger[N].pseudostate
lan.trigger[N].pseudostate = pseudostate
```

pseudostate	The simulated line state: `0` or `1`
N	A number representing the LAN event number: 1 to 8

Details

This attribute can be set to initialize the pseudo line state to a known value.

Setting this attribute does not cause the LAN trigger to generate any events or output packets.

Example

`print(lan.trigger[1].pseudostate)`
Get the present simulated line state for the LAN event 1.

Also see

None

lan.trigger[N].stimulus

This attribute specifies events that cause this trigger to assert.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset LAN trigger N reset Recall setup	Create configuration script	0

Usage

```
triggerStimulus = lan.trigger[N].stimulus
lan.trigger[N].stimulus = triggerStimulus
```

triggerStimulus	The LAN event identifier used to trigger the event; see **Details**
N	A number specifying the trigger packet over the LAN for which to set or query the trigger source: 1 to 8

Details

This attribute specifies which event causes a LAN trigger packet to be sent for this trigger. Set *triggerStimulus* to one of the existing trigger event IDs shown in the following table.

Trigger event IDs

Trigger event ID	Description
digio.trigger[*N*].EVENT_ID	An edge (either rising, falling, or either based on the configuration of the line) on the digital input line.
display.trigger.EVENT_ID	The trigger key on the front panel is pressed.
trigger.EVENT_ID	A *trg message on the active command interface. If GPIB is the active command interface, a GET message also generates this event.
trigger.blender[*N*].EVENT_ID	A combination of events has occurred.
trigger.timer[*N*].EVENT_ID	A delay expired.
tsplink.trigger[*N*].EVENT_ID	An edge (either rising, falling, or either based on the configuration of the line) on the TSP-Link trigger line.
lan.trigger[*N*].EVENT_ID	A LAN trigger event has occurred.
scan.trigger.EVENT_SCAN_READY	Scan ready event.
scan.trigger.EVENT_SCAN_START	Scan start event.
scan.trigger.EVENT_CHANNEL_READY	Channel ready event.
scan.trigger.EVENT_SCAN_COMP	Scan complete event.
scan.trigger.EVENT_IDLE	Idle event.

Set this attribute to zero (0) to disable automatic trigger generation.

If any events are detected before calling `lan.trigger[N].connect()`, the event is ignored and the action overrun is set.

Example

```
lan.trigger[5].stimulus = trigger.timer[1].EVENT_ID
```

Use timer 1 trigger event as the source for LAN packet 5 trigger stimulus.

Also see

lan.trigger[N].assert() (on page 7-115)
lan.trigger[N].clear() (on page 7-116)
lan.trigger[N].connect() (on page 7-117)
lan.trigger[N].overrun (on page 7-121)
lan.trigger[N].wait() (on page 7-125)

lan.trigger[N].wait()

This function waits for an input trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`triggered = lan.trigger[N].wait(timeout)`

triggered	Trigger detection indication: `true` or `false`
N	The trigger packet over LAN to wait for: 1 to 8
timeout	Maximum amount of time in seconds to wait for the trigger event

Details

If one or more trigger events have been detected since the last time `lan.trigger[N].wait()` or `lan.trigger[N].clear()` was called, this function returns immediately.

After waiting for a LAN trigger event with this function, the event detector is automatically reset and rearmed regardless of the number of events detected.

Example

`triggered = lan.trigger[5].wait(3)`
Wait for a trigger with LAN packet 5 with a timeout of 3 seconds.

Also see

lan.trigger[N].assert() (on page 7-115)
lan.trigger[N].clear() (on page 7-116)
lan.trigger[N].overrun (on page 7-121)
lan.trigger[N].stimulus (on page 7-123)

localnode.define.*

These constants indicate the number of available features of each feature type for each local node instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
CONSTANT (R)	—			
.MAX_TIMERS	Yes			
.MAX_DIO_LINES	Yes			
.MAX_TSPLINK_TRIGS	Yes			
.MAX_BLENDERS	Yes			
.MAX_BLENDER_INPUTS	Yes			
.MAX_LAN_TRIGS	Yes			
.MAX_CHANNEL_TRIGS	Yes			

Usage

```
maxNumber = localnode.define.MAX_TIMERS
maxNumber = localnode.define.MAX_DIO_LINES
maxNumber = localnode.define.MAX_TSPLINK_TRIGS
maxNumber = localnode.define.MAX_BLENDERS
maxNumber = localnode.define.MAX_BLENDER_INPUTS
maxNumber = localnode.define.MAX_LAN_TRIGS
maxNumber = localnode.define.MAX_CHANNEL_TRIGS
```

maxNumber	A variable assigned the value of the constant; the constant equals the maximum number available for the specified feature on the local node instrument

Details

These read-only constants indicate the maximum number of each feature type that is available. Feature types are timers, digital input/output lines, triggers, and blenders.

When using this command from a remote node, `localnode` should be replaced with the node reference, for example, `node[5].define.MAX_TIMERS`.

Example

`maxNumber = localnode.define.MAX_TIMERS`
Reads the maximum number of timers that are available for the presently active instrument.

Also see

None

localnode.description

This attribute stores a user-defined description and mDNS service name of the instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Not applicable	Nonvolatile memory	Instrument specific (see **Details**)

Usage

```
localnode.description = "description"
description = localnode.description
```

description	User-defined description and mDNS service name of the instrument; use a string of 63 characters or less

Details

This attribute stores a string that contains a description of the instrument. This value appears on the LXI welcome page and is used as the mDNS service name of the instrument.

The default value of this attribute contains `Keithley` *`ModelNumber`* `#`*`SSSSSSSS`*, where:

- *`ModelNumber`* is the model number of the instrument.
- `#`*`SSSSSSSS`* is the eight-digit serial number of the instrument. You can change it to a value that makes sense for your system.

To revert this description to the factory default value, set this attribute to an empty string. An empty string is a string of length zero or a string consisting entirely of whitespace characters.

When using this command from a remote node, replace `localnode` with the node reference, for example `node[5].description`.

Example

```
description = "System in Lab 05"
localnode.description = description
```

Set `description` to `System in Lab 05`.

Also see

None

localnode.license

This attribute returns the product license agreements.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Nonvolatile memory	Not applicable

Usage

`license_agreement` `= localnode.license`

`license_agreement`	The text of the license agreements

Example

```
print(localnode.license)
```

Returns the license agreements for the Models 707B and 708B.

Also see

None

localnode.model

This attribute stores the model number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
model = localnode.model
```

model	The model number of the instrument

Example

`print(localnode.model)`
Outputs the model number of the local node. For example: `707B`

Also see

localnode.serialno (on page 7-132)

localnode.password

This attribute stores the remote access password.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (W)	Yes	LAN reset	Nonvolatile memory	"admin"

Usage

```
localnode.password = "password"
```

password	A string that contains the remote interface password, up to 255 characters

Details

This attribute stores the password that is set for any remote interface.

The instrument continues to use the old password for all interactions until the command to change it executes. When changing the password, give the instrument time to execute the command before attempting to use the new password.

You can retrieve the password from the front panel through **MENU > LAN > STATUS > PASSWORD**.

You can reset the password by resetting the LAN from the front panel or by sending the `lan.reset()` command.

When using this command from a remote node, `localnode` should be replaced with the node reference, for example, `node[5].password`.

Example

`localnode.password = "N3wpa55w0rd"`
Changes the remote interface password to `N3wpa55w0rd`.

Also see

lan.reset() (on page 7-107)

localnode.prompts

This attribute determines if the instrument generates prompts in response to command messages.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Power cycle	Not saved	0 (disabled)

Usage

```
prompting = localnode.prompts
localnode.prompts = prompting
```

prompting	Prompting mode: ▪ Do not generate prompts: `0` ▪ Generate prompts: `1`

Details

When the prompting mode is enabled, the instrument generates prompts when the instrument is ready to take another command. Because the prompt is not generated until the previous command completes, enabling prompts provides handshaking with the instrument to prevent buffer overruns.

When prompting is enabled, the instrument might generate the following prompts:

- **TSP>.** The standard prompt, which indicates that the previous command completed normally.
- **TSP?.** The prompt that is issued if there are unread entries in the error queue when the prompt is issued. Like the TSP> prompt, it indicates that processing of the command is complete. It does not mean the previous command generated an error, only that there were still errors in the queue when the command processing was complete.
- **>>>>.** The continuation prompt, which occurs when downloading scripts. When downloading scripts, many command messages must be sent as a group. The continuation prompt indicates that the instrument is expecting more messages as part of the present command.

Commands do not generate prompts. The instrument generates prompts in response to command completion.

Prompts are enabled or disabled only for the remote interface that is active when you send the command. For example, if you enable prompts when the LAN connection is active, they are not enabled for a subsequent USB connection.

NOTE

Do not disable prompting when using Test Script Builder. Test Script Builder requires prompts and sets the prompting mode automatically. If you disable prompting, the instrument stops responding when you communicate using Test Script Builder because it is waiting for a common complete prompt from Test Script Builder.

Example

`localnode.prompts = 1`	
Enable prompting.	

Also see

localnode.prompts4882 (on page 7-130)
localnode.showerrors (on page 7-133)
tsplink.reset() (on page 7-226)

localnode.prompts4882

This attribute enables and disables the generation of prompts for IEEE Std 488.2 common commands.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Power cycle	Not saved	1 (enabled)

Usage

```
prompting = localnode.prompts4882
localnode.prompts4882 = prompting
```

prompting	IEEE Std 488.2 prompting mode: ■ Disable prompting: `0` ■ Enable prompting: `1`

Details

When this attribute is enabled, the IEEE Std 488.2 common commands generate prompts if prompting is enabled with the `localnode.prompts` attribute. If `localnode.prompts4882` is enabled, limit the number of `*trg` commands sent to a running script to 50 regardless of the setting of the `localnode.prompts` attribute.

When this attribute is disabled, IEEE Std 488.2 common commands do not generate prompts. When using the `*trg` command with a script that executes `trigger.wait()` repeatedly, disable prompting to avoid problems associated with the command interface input queue filling.

Example

`localnode.prompts4882 = 0`	
Disables IEEE Std 488.2 common command prompting.	

Also see

localnode.prompts (on page 7-129)

localnode.reset()

This function resets the local node instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
localnode.reset()
```

Details

A local node reset includes a `channel.reset("allslots")` and a `scan.reset()`. In addition:

- Other settings are restored to factory default settings
- Existing channel patterns are deleted
- All channels are opened

A `localnode.reset()` is different than a `reset()` because `reset()` resets the entire system.

If you want to reset a specific instrument or a subordinate node, use the `node[X].reset()` command. To do this, replace `localnode` with the node reference, for example, `node[5].reset()`.

Example

`localnode.reset()`
Resets the local node.

Also see

channel.reset() (on page 7-42)
reset() (on page 7-142)
scan.reset() (on page 7-152)

localnode.revision

This attribute stores the firmware revision level.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

revision = localnode.revision

revision	Firmware revision level

Details

This attribute indicates the revision number of the firmware that is presently running in the instrument.

When using this command from a remote node, replace `localnode` with the node reference. For example, `node[5].revision`.

Example

`print(localnode.revision)`
Outputs the present revision level. Sample output: `01.00a`

Also see

localnode.description (on page 7-126)
localnode.model (on page 7-128)
localnode.serialno (on page 7-132)

localnode.serialno

This attribute stores the serial number of the instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

serialno = localnode.serialno

serialno	The serial number of the instrument

Details

This indicates the instrument serial number.

Example

`display.clear()` `display.settext(localnode.serialno)`	
Clears the instrument display. Places the serial number of the instrument on the top line of its display.	

Also see

localnode.description (on page 7-126)
localnode.model (on page 7-128)
localnode.revision (on page 7-132)

localnode.showerrors

This attribute sets whether or not the instrument automatically sends generated errors.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Power cycle	Not saved	0 (disabled)

Usage

```
errorMode = localnode.showerrors
localnode.showerrors = errorMode
```

errorMode	Show error setting: ■ Show errors: `1` ■ Do not show errors: `0`

Details

If this attribute is set to `1`, the instrument automatically sends any generated errors stored in the error queue, and then clears the queue. Errors are processed either after executing a command message or immediately before issuing a prompt if prompts are enabled.

If this attribute is set to `0`, errors are left in the error queue and must be explicitly read or cleared.

When using this command from a remote node, replace `localnode` with the node reference, for example, `node[5].showerrors`.

Example

`localnode.showerrors = 1`
Enables sending of generated errors.

Also see

localnode.prompts (on page 7-129)

makegetter()

This function creates a function to get the value of an attribute.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`getter = makegetter(table, "attributeName")`

getter	The return value
table	Read-only table where the attribute is located
attributeName	A string representing the name of the attribute

Details

This function is useful for aliasing attributes to improve execution speed. Calling the function created with `makegetter()` executes more quickly than accessing the attribute directly.

Creating a getter function is only useful if it is going to be called several times. Otherwise, the overhead of creating the getter function outweighs the overhead of accessing the attribute directly.

Example

```
getRule = makegetter(channel, "connectrule")
-- (intervening code)
r = getRule()
```

Creates a getter function called `getRule`.
When `getRule()` is called, it returns the value of `connectrule`.

Also see

makesetter() (on page 7-134)

makesetter()

This function creates a function that, when called, sets the value of an attribute.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`setter = makesetter(table, "attributeName")`

setter	Function that sets the value of the attribute
table	Read-only table where the attribute is located
attributeName	The string name of the attribute

Details

This function is useful for aliasing attributes to improve execution speed. Calling the *setter* function execute more quickly than accessing the attribute directly.

Creating a *setter* function is only useful if it is going to be called several times. If you are not calling the *setter* function several times, it is more efficient to access the attribute directly.

Example

```
setRule = makesetter(channel, "connectrule")
r = setRule(channel.BREAK_BEFORE_MAKE)
```

Creates a setter function called `setRule`.
When `setRule()` is called, it configures the setting for `connectrule`. In this example, the connection rule is set to break-before-make.

Also see

makegetter() (on page 7-134)

memory.available()

This function reads and returns the amount of memory that is available in the instrument overall for storing user scripts and channel patterns.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

memoryAvailable = `memory.available()`

memoryAvailable	Comma-delimited string with percentages for available memory; the format is `systemMemory`, `scriptMemory`, `patternMemory`, where: ■ `systemMemory`: The percentage of memory available in the instrument ■ `scriptMemory`: The percentage of memory available in the instrument to store user scripts ■ `patternMemory`: The percentage of memory available in the instrument to store channel patterns

Details

Use this function to view the available memory in the overall instrument as well as the memory available for storing user scripts and channel patterns.

The response to this function is a single string that returns the overall instrument memory available, script memory available, and channel pattern memory available as comma-delimited percentages.

Example: Available memory

`memoryAvailable = memory.available()` `print(memoryAvailable)`

Reads and returns the amount of memory available in the instrument.
Output:
`51.56, 92.84, 100.00`
You can also use:
`print(memory.available())`

Example: Used and available memory

`print("Memory used:", memory.used())` `print("Memory available: ", memory.available())`

Reads and returns the amount memory used and memory available percentages.
Output:
`Memory used: 69.14, 0.16, 12.74`
`Memory available: 30.86, 99.84, 87.26`

Also see

memory.used() (on page 7-136)

memory.used()

This function reports the amount of memory used in the instrument overall, for user scripts and storing channel patterns.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
memoryUsed = memory.used()
```

memoryUsed	A comma-delimited string with percentages for used memory; the format is `systemMemory`, `scriptMemory`, `patternMemory`, where: ■ `systemMemory`: The percentage of memory used in the instrument ■ `scriptMemory`: The percentage of memory used in the instrument to store user scripts ■ `patternMemory`: The percentage of memory used in the instrument to store channel patterns

Details

The response to this function is a single string that shows the overall instrument memory used and the memory used for storing user scripts and channel patterns as comma-delimited percentages.

Example

`MemUsed = memory.used()` `print(MemUsed)`	
Reads the memory used in the instrument and returns the percentages. Output: `69.14, 0.16, 12.74`	

Also see

memory.available() (on page 7-135)

node[N].execute()

This function starts test scripts on a remote TSP-Link node.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes (see **Details**)			

Usage

```
node[N].execute("scriptCode")
```

N	The node number of this instrument: 1 to 63
scriptCode	A string containing the source code

Details

This command is only applicable to TSP-Link systems. You can use this command to use the remote master node to run a script on the specified node. This function does not run test scripts on the master node; only on the subordinate node when initiated by the master node.

This function may only be called when the group number of the node is different than the node of the master.

This function does not wait for the script to finish execution.

Example 1

`node[2].execute(sourcecode)`
Runs script code on node 2. The code is in a string variable called `sourcecode`.

Example 2

`node[3].execute("x = 5")`
Runs script code in string constant (`x = 5`) to set `x` equal to 5 on node 3.

Example 3

`node[32].execute(TestDut.source)`
Runs the test script stored in the variable `TestDut` (previously stored on the master node) on node 32.

Also see

Using TSP to run test scripts simultaneously (on page 6-51)
tsplink.group (on page 7-223)

node[N].getglobal()

This function returns the value of a global variable.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`value = node[N].getglobal("name")`

value	The value of the variable
N	The node number of this instrument: 1 to 63
name	The global variable name

Details

This function retrieves the value of a global variable from the runtime environment of this node.

Do not use this command to retrieve the value of a global variable from the local node. Instead, access the global variable directly. This command should only be used from a remote master when controlling this instrument over a TSP-Link network.

Example

`print(node[5].getglobal("test_val"))`
Retrieves and outputs the value of the global variable named `test_val` from node 5.

Also see

Using TSP to run test scripts simultaneously (on page 6-51)

node[N].setglobal()

This function sets the value of a global variable.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`node[N].setglobal("name", value)`

N	The node number of this instrument: 1 to 63
name	The global variable name
value	The value of the variable

Details

From a remote node, use this function to assign the given value to a global variable.

Do not use this command to create or set the value of a global variable from the local node (set the global variable directly instead). This command should only be used from a remote master when controlling this instrument over a TSP-Link network.

Example

`node[3].setglobal("x", 5)`
Sets the global variable `x` on node 3 to the value of 5.

Also see

Using TSP to run test scripts simultaneously (on page 6-51)

opc()

This function sets the operation complete status bit when all overlapped commands are completed.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
opc()
```

Details

This function causes the operation complete bit in the Standard Event Status Register to be set when all previously started local overlapped commands are complete.

Note that each node independently sets its operation complete bits in its own status model. Any nodes that are not actively performing overlapped commands set their bits immediately. All remaining nodes set their own bits as they complete their own overlapped commands.

Example

```
opc()
waitcomplete()
print("1")
```

Output:

```
1
```

Also see

Status model (on page 12-1)
waitcomplete() (on page 7-253)

print()

This function generates a response message.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
print(value1)
print(value1, value2)
print(value1, ..., valueN)
```

value1	The first argument to output
value2	The second argument to output
valueN	The last argument to output
...	One or more values separated with commas

Details

TSP-enabled instruments do not have inherent query commands. Like other scripting environments, the `print()` command and other related `print()` commands generate output. The `print()` command creates one response message.

The output from multiple arguments is separated with a tab character.

Numbers are printed using the `format.asciiprecision` attribute. If you want use Lua formatting, print the return value from the `tostring()` function.

Example 1

```
x = 10
print(x)
```

Example of an output response message:
`10`
Your output might be different, depending on the ASCII precision setting.

Example 2

```
x = true
print(tostring(x))
```

Example of an output response message:
`true`

Also see

format.asciiprecision (on page 7-94)

printnumber()

This function prints numbers using the configured format.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
printnumber(value1)
printnumber(value1, value2)
printnumber(value1, ..., valueN)
```

value1	First value to print in the configured format
value2	Second value to print in the configured format
valueN	Last value to print in the configured format
...	One or more values separated with commas

Details

There are multiple ways to use this function, depending on how many numbers are to be printed.

This function prints the given numbers using the data format specified by `format.data` and `format.asciiprecision`.

Example

```
format.asciiprecision = 10
x = 2.54
printnumber(x)
format.asciiprecision = 3
printnumber(x, 2.54321, 3.1)
```

Configure the ASCII precision to 10 and set `x` to 2.54.
Read the value of `x` based on these settings.
Change the ASCII precision to 3.
View how the change affects the output of `x` and some numbers.
Output:

```
2.54000000000e+00
2.54e+00, 2.54e+00, 3.10e+00
```

Also see

format.asciiprecision (on page 7-94)
format.byteorder (on page 7-95)
format.data (on page 7-96)
print() (on page 7-140)

reset()

This function resets commands to their default settings.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
reset()
reset(system)
```

system	What to reset: ■ If the node is the master, the entire system is reset (default): `true` ■ Only the local group is reset: `false`

Details

The `reset()` command in its simplest form resets the entire TSP-enabled system, including the controlling node and all subordinate nodes.

If you want to reset a specific instrument, use either the `localnode.reset()` or `node[X].reset()` command. Use the `localnode.reset()` command for the local instrument. Use the `node[X].reset()` command to reset an instrument on a subordinate node.

You can only reset the entire system using `reset(true)` if the node is the master. If the node is not the master node, executing this command generates an error.

Example

`reset(true)`
If the node is the master node, the entire system is reset; if the node is not the master node, an error is generated.

Also see

localnode.reset() (on page 7-131)

scan.abort()

This function aborts a running background scan.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.abort()
```

Details

If no scan is running, the call to this function is ignored.

NOTE

When a scan is aborted, the channels remain in the opened or closed states that they were in when the scan was aborted.

Example

```
scan.background()
scan.abort()
```

Starts a background scan and then aborts the scan.

Also see

scan.background() (on page 7-146)
Scan and trigger (on page 2-21)

scan.add()

This function adds channels to the scan list.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.add("channelList")
```

channelList	String specifying channels to add using normal channel list syntax

Details

Use this function to add channels and channel patterns to the present scan list. If the scan list does not exist, it also creates a scan list. See `scan.create()` for information about creating a scan list.

Channels and channel patterns added using this command are added to the end of the present list (appended) in the order specified in the channel list. Specifying multiple channels adds multiple steps to the scan.

If an error is encountered as channels are added to the list, subsequent channels in that channel list will not be added.

Example 1

```
scan.create()
for column = 1,5 do
   scan.add(channel.createspecifier(1,1,column))
end
```

Replaces the active scan list with an empty scan list.
Loops through columns 1 to 5.
Adds five channels to the scan list using the `channel.createspecifier()` command. The scan list now has row 1, columns 1 to 5 on slot 1 as the first five steps.

Example 2

```
scan.create("1A01:1A08")
scan.add("1A10")
scan.add("1A09")
```

Replaces the active scan list with an empty scan list, and then adds row A, columns 1 through 8 on slot 1 to the new scan list.
Adds row A, column 10 on slot 1 to the end of the scan list.
Adds row A, column 9 on slot 1 to the end of the scan list.
Scan list now includes channels 1A01 through 1A10, with channels 1A01 through 1A08 in order, followed by channel 1A10, and then channel 1A09.

Example 3

```
scan.create("")
```

Clears the old scan list and creates a new empty scan list.

Also see

scan.create() (on page 7-148)
Scan and trigger (on page 2-21)

scan.addimagestep()

This function allows you to include multiple channels in a single scan step.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.addimagestep("channelList")
```

channelList	String specifying a list of channels

Details

This function adds a list of channels to be closed simultaneously in a single step of a scan.

Example

```
scan.create()
scan.add("1D01")
scan.addimagestep("1A01, 1B01, 1C03")
scan.add("1F03")
scan.addimagestep("1A03, 1B03, 1C03")
scan.addimagestep("1A05, 1B05, 1C03")
scan.addimagestep("1A07, 1B07, 1C03")
scan.addimagestep("1A09, 1B09, 1C03")
print(scan.list())
```

Generate a scan list that has multiple steps, with some steps that include multiple channels.
Output:

```
Init) OPEN...
   1) STEP: 1D01
     CLOSE: 1D01
   2) STEP: 1A01, 1B01, 1C03
      OPEN: 1D01
     CLOSE: 1A01 1B01 1C03
   3) STEP: 1F03
      OPEN: 1A01 1B01 1C03
     CLOSE: 1F03
   4) STEP: 1A03, 1B03, 1C03
      OPEN: 1F03
     CLOSE: 1A03 1B03 1C03
   5) STEP: 1A05, 1B05, 1C03
      OPEN: 1A03 1B03
     CLOSE: 1A05 1B05
   6) STEP: 1A07, 1B07, 1C03
      OPEN: 1A05 1B05
     CLOSE: 1A07 1B07
   7) STEP: 1A09, 1B09, 1C03
      OPEN: 1A07 1B07
     CLOSE: 1A09 1B09
```

Also see

scan.add() (on page 7-144)
Scan and trigger (on page 2-21)

scan.background()

This function starts a scan and runs the scan in the background.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`state, scanCount, stepCount = scan.background()`

state	The result of scanning:
	`scan.EMPTY` or `0`
	`scan.BUILDING` or `1`
	`scan.RUNNING` or `2`
	`scan.ABORTED` or `3`
	`scan.FAILED` or `4`
	`scan.FAILED_INIT` or `5`
	`scan.SUCCESS` or `6`

scanCount	The present number of scans completed
stepCount	The present number of steps completed

Details

Before using this command, use `scan.create()` and `scan.add()` or `scan.addimagestep()` to set up a scan list.

When the scan is run in the background, you must use the `scan.state()` function to check the status of the scan.

Example

`scan.background()`
Runs a scan in the background.

Also see

scan.add() (on page 7-144)
scan.create() (on page 7-148)
scan.execute() (on page 7-149)
scan.list() (on page 7-149)
scan.state() (on page 7-154)
Scan and trigger (on page 2-21)

scan.bypass

This attribute indicates whether the first channel of the scan waits for the channel stimulus event to be satisfied before closing.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	System reset Scan reset	Create configuration script	1 (scan.ON)

Usage

```
bypass = scan.bypass
scan.bypass = bypass
```

bypass	Enable or disable the bypass: ■ Disabled: `scan.OFF` or `0` ■ Enabled: `scan.ON` or `1`

Details

When *bypass* is set to on and the `scan.trigger.arm.stimulus` is set to a nonzero value, the first channel of the scan closes (the `scan.trigger.channel.stimulus` setting is ignored).

For other channels (other than the first), the channel stimulus must be satisfied before the channel action takes place.

When *bypass* is set to off, every channel (including the first) must satisfy the `scan.trigger.channel.stimulus` setting before the channel action occurs for that step.

Example

```
scan.bypass = scan.OFF
print(scan.bypass)
```

Disables the bypass option for scanning and displays the present bypass state.
Output:
`0.000000000e+000`

Also see

scan.trigger.arm.stimulus (on page 7-156)
scan.trigger.channel.stimulus (on page 7-158)
Scan and trigger (on page 2-21)

scan.create()

This function deletes the existing scan list and creates a new list of channels and channel patterns to scan.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.create(channelList)
```

channelList	String specifying channels to add

Details

The existing scan list is lost after calling this function.

The items in the channel list are scanned in the order listed.

If a forbidden channel is included in a range of channels or slot parameter (such as slot 1), the forbidden channel is ignored and no error is generated. If a forbidden channel is individually specified in the channel list, an error is generated.

If an error occurs, the scan list of channels or channel patterns is cleared, even though no new scan list is created.

The function `scan.reset()` clears the list. To clear the scan list without performing a scan reset, send an empty string for the *channelList* parameter.

Example 1

```
scan.create("1A01:1A10")
```

Replaces the active scan list with an empty scan list, and then adds channels `1A01` through `1A10` on slot 1.

Example 2

```
scan.create()
for column = 1, 10 do
   scan.add(channel.createspecifier(1,1,column))
end
```

Replaces the active scan list with an empty scan list.
Loops through column 1 to 10 on row 1 of slot 1 to add ten channels to the scan list. The `channel.createspecifier()` command generates the parameters.
The scan list now has, in order, row 1, columns 1 through 10, on slot 1.

Also see

scan.add() (on page 7-144)
scan.reset() (on page 7-152)
Scan and trigger (on page 2-21)

scan.execute()

This function starts the scan immediately in the foreground with a configured scan list.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scanState, scanCount, stepCount = scan.execute()
```

scanState	The result of scanning:
	`scan.EMPTY` or `0`
	`scan.BUILDING` or `1`
	`scan.RUNNING` or `2`
	`scan.ABORTED` or `3`
	`scan.FAILED` or `4`
	`scan.FAILED_INIT` or `5`
	`scan.SUCCESS` or `6`

scanCount	The present number of scans that have completed
stepCount	The present number of steps have completed

Details

Before using this command, use `scan.create()` and `scan.add()` or `scan.addimagestep()` to set up a scan list.

Execution runs until the scan is complete or until the `abort` command is sent.

Because this function waits for the scan to complete, the `scan.state()` function cannot be used to see the present status of scanning.

Example

`scan.execute()`
Runs a scan immediately.

Also see

scan.add() (on page 7-144)
scan.background() (on page 7-146)
scan.create() (on page 7-148)
scan.list() (on page 7-149)
scan.state() (on page 7-154)
Scan and trigger (on page 2-21)

scan.list()

This function queries the active scan list.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes	Instrument reset Channel reset Scan reset Recall setup Change of channel or scan setting	Create configuration script Save setup	Empty list

Usage

```
scanList = scan.list()
```

scanList	A string that lists the existing scan step information

Details

When you change a channel or scan attribute for an existing scan list item, the scan list is recreated based on this change. If the scan list cannot be rebuilt, the scan list is lost.

To avoid unintentional changes to an existing scan list, configure channel and scan settings before adding or creating a scan or step using the commands `scan.add()`, `scan.addimagestep()`, and `scan.create()`.

If the scan list is empty, the string `"Empty Scan"` is returned. Otherwise, the string lists each step in the scan along with its information for step, open, and close.

Example

```
scan.create("1A07:1B03")
print(scan.list())
```

Populate the scan list with the function `scan.create("1A07:1B03")`, then initiate the scan list to be output.
Outputs the existing scan list.
Output:

```
Init) OPEN...
   1) STEP: 1A07
     CLOSE: 1A07
   2) STEP: 1A08
      OPEN: 1A07
     CLOSE: 1A08
   3) STEP: 1A09
      OPEN: 1A08
     CLOSE: 1A09
   4) STEP: 1A10
      OPEN: 1A09
     CLOSE: 1A10
   5) STEP: 1A11
      OPEN: 1A10
     CLOSE: 1A11
   6) STEP: 1A12
      OPEN: 1A11
     CLOSE: 1A12
   7) STEP: 1B01
      OPEN: 1A12
     CLOSE: 1B01
   8) STEP: 1B02
      OPEN: 1B01
     CLOSE: 1B02
   9) STEP: 1B03
      OPEN: 1B02
     CLOSE: 1B03
```

Also see

scan.add() (on page 7-144)
scan.addimagestep() (on page 7-145)
scan.create() (on page 7-148)
Scan and trigger (on page 2-21)

scan.mode

This attribute sets the relay action when the scan starts.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	System reset Scan reset	Create configuration script	0 (scan.MODE_OPEN_ALL)

Usage

```
scanModeSetting = scan.mode
scan.mode = scanModeSetting
```

scanModeSetting	The present scan mode setting; set to one of the following values: ■ `scan.MODE_OPEN_ALL` or `0` ■ `scan.MODE_OPEN_SELECTIVE` or `1`: See **Details**

Details

When this attribute is set to open all, channels on all slots are opened before a scan starts.

When this attribute is set to `scan.MODE_OPEN_SELECTIVE`, an intelligent open is performed:

- All channels used in scanning are opened
- Closed channels not used in scanning remain closed during the scan

Example

`scan.mode = scan.MODE_OPEN_SELECTIVE`
Sets the scan mode setting to open channels that are used in the scan.

Also see

scan.reset() (on page 7-152)
Scan and trigger (on page 2-21)

scan.reset()

This function resets the trigger model and scan list settings to their factory default settings.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.reset()
```

Details

When `scan.reset()` is sent, the trigger model and scan settings that are reset to the factory defaults are:

- `scan.bypass`
- `scan.mode`
- `scan.scancount`
- `scan.trigger.arm.stimulus`
- `scan.trigger.channel.stimulus`

In addition, the scan list is cleared.

NOTE

Sending this function only affects the trigger model and scan list settings. To reset all instrument settings to factory default settings, use the `reset()` command.

Example

`scan.reset()`
Performs a reset on the trigger model and scan settings.

Also see

channel.reset() (on page 7-42)
reset() (on page 7-142)
scan.bypass (on page 7-147)
scan.mode (on page 7-151)
scan.scancount (on page 7-153)
scan.trigger.arm.stimulus (on page 7-156)
scan.trigger.channel.stimulus (on page 7-158)
Scan and trigger (on page 2-21)

scan.scancount

This attribute sets the number of times the scan is repeated.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Scan reset	Create configuration script	1

Usage

```
scanCount = scan.scancount
scan.scancount = scanCount
```

scanCount	The scan count value: 1 to 2,000,000,000

Details

The scan count attribute setting indicates how many times the scan list is iterated through before the scan completes.

During a scan, the instrument iterates through the arm layer of the trigger model the specified number of times. After performing the specified number of iterations, the instrument returns to an idle state.

Example

```
scan.create("1001:1009")
scan.scancount = 100
scan.scaninterval = 1.0 -- delay between scans
```

Create a scan that includes channels 1 to 9 of slot 1.
Set the scan count to 100.
Set the delay between scans to 1 s.

Also see

Scan and trigger (on page 2-21)
Trigger model (on page 2-21)

scan.state()

This function provides the present state of a running background scan.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`scanState, scanCount, stepCount = scan.state()`

scanState	The present state of the scan running in the background; possible states include: ■ `scan.EMPTY` or `0` ■ `scan.BUILDING` or `1` ■ `scan.RUNNING` or `2` ■ `scan.ABORTED` or `3` ■ `scan.FAILED` or `4` ■ `scan.FAILED_INIT` or `5` ■ `scan.SUCCESS` or `6`
scanCount	The number of scans that have completed
stepCount	The number of steps that have completed

Details

Returns the state of the present scan, the scan count, and the step count.

The scan count is the number of the present iteration through the scan portion of the trigger model. This number does not increment until the scan begins. Therefore, if the instrument is waiting for an input to trigger a scan start, the scan count represents the previous number of scan iterations. If no scan has begun, the scan count is zero.

The step count is the number of times the scan has completed a pass through the channel action portion of the trigger model. This number does not increment until after the action completes. Therefore, if the instrument is waiting for an input to trigger a channel action, the step count represents the previous step. If no step has yet completed, the step count is zero. If the step count has yet to complete the first step in a subsequent pass through a scan, the scan count represents the last step in the previous scan pass.

Example

```
scan.background()
scanState, scanCount, stepCount = scan.state()
print(scanState, scanCount, stepCount)
```

Runs a scan in the background.
Check the present scan state.
View returned values.
Output shows that scan is running:
`2.0000000e+00`

Also see

scan.background() (on page 7-146)
scan.mode (on page 7-151)
Scan and trigger (on page 2-21)

scan.stepcount

This attribute returns the number of steps in the present scan.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
scanStepCount = scan.stepcount
```

scanStepCount	The present step count value

Details

This is set by the number of steps in the active scan list. The value of this attribute is initially determined when the scan is created. When you add steps with the `scan.create()`, `scan.addimagestep()`, and `scan.add()` functions, the value of this attribute is updated.

Example

```
print(scan.stepcount)
```

Responds with the present step count.
Output assuming there are five steps in the scan list:
5.0000000e+00

Also see

scan.add() (on page 7-144)
scan.addimagestep() (on page 7-145)
scan.create() (on page 7-148)
Scan and trigger (on page 2-21)

scan.trigger.arm.clear()

This function clears the arm event detector.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.trigger.arm.clear()
```

Details

This function sets the arm event detector of the trigger model to the undetected state.

Example

```
scan.trigger.arm.clear()
```

Clears the arm event detector.

Also see

scan.trigger.arm.set() (on page 7-156)
scan.trigger.arm.stimulus (on page 7-156)
Trigger model (on page 2-21)
Scan and trigger (on page 2-21)

scan.trigger.arm.set()

This function sets the arm event detector to the detected state.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.trigger.arm.set()
```

Details

This function sets the arm event detector of the trigger model to the detected state.

Example

`scan.trigger.arm.set()`
Sets the arm event detector to the detected state.

Also see

scan.trigger.arm.clear() (on page 7-155)
scan.trigger.arm.stimulus (on page 7-156)
Trigger model (on page 2-21)
Scan and trigger (on page 2-21)

scan.trigger.arm.stimulus

This attribute determines which event starts the scan.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup Scan reset	Create configuration script Save setup	0

Usage

```
eventID = scan.trigger.arm.stimulus
scan.trigger.arm.stimulus = eventID
```

eventID	Trigger stimulus used for the channel action (arm layer); see **Details**

Details

This attribute selects which events cause the arm event detector to enter the detected state.

Set this attribute to 0 to start the scan without waiting for an event.

`eventID` may be one of the following trigger event IDs.

Trigger event IDs

Trigger event ID	Description
`digio.trigger[N].EVENT_ID`	An edge (either rising, falling, or either based on the configuration of the line) on the digital input line.
`display.trigger.EVENT_ID`	The trigger key on the front panel is pressed.
`trigger.EVENT_ID`	A `*trg` message on the active command interface. If GPIB is the active command interface, a GET message also generates this event.
`trigger.blender[N].EVENT_ID`	A combination of events has occurred.
`trigger.timer[N].EVENT_ID`	A delay expired.
`tsplink.trigger[N].EVENT_ID`	An edge (either rising, falling, or either based on the configuration of the line) on the TSP-Link trigger line.
`lan.trigger[N].EVENT_ID`	A LAN trigger event has occurred.
`scan.trigger.EVENT_SCAN_READY`	Scan ready event.
`scan.trigger.EVENT_SCAN_START`	Scan start event.
`scan.trigger.EVENT_CHANNEL_READY`	Channel ready event.
`scan.trigger.EVENT_SCAN_COMP`	Scan complete event.
`scan.trigger.EVENT_IDLE`	Idle event.

Example 1

```
scan.trigger.arm.stimulus = scan.trigger.EVENT_SCAN_READY
```

Sets trigger stimulus of the arm event detector to scan ready event.

Example 2

```
scan.trigger.arm.stimulus = 0
```

The scan begins immediately.

Example 3

```
scan.trigger.arm.stimulus = digio.trigger[3].EVENT_ID
```

The scan begins when the instrument receives a signal from digital I/O line 3.

Also see

scan.trigger.arm.clear() (on page 7-155)
scan.trigger.arm.set() (on page 7-156)
Trigger model (on page 2-21)
Scan and trigger (on page 2-21)

scan.trigger.channel.clear()

This function clears the channel event detector.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.trigger.channel.clear()
```

Details

This function clears the channel event detector of the trigger model (sets it to the undetected state).

Example

`scan.trigger.channel.clear()`
Clears the channel event detector.

Also see

scan.trigger.channel.set() (on page 7-158)
scan.trigger.channel.stimulus (on page 7-158)
Trigger model (on page 2-21)
Scan and trigger (on page 2-21)

scan.trigger.channel.set()

This function sets the channel event detector of the trigger model to the detected state.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.trigger.channel.set()
```

Example

`scan.trigger.channel.set()`
Sets the channel event detector of the trigger model to the detected state.

Also see

scan.trigger.channel.clear() (on page 7-157)
scan.trigger.channel.stimulus (on page 7-158)
Scan and trigger (on page 2-21)
Trigger model (on page 2-21)

scan.trigger.channel.stimulus

This attribute determines which trigger events cause the channel actions to occur.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Scan reset	Create configuration script	scan.trigger.EVENT_CHANNEL_READY

Usage

```
eventID = scan.trigger.channel.stimulus
scan.trigger.channel.stimulus = eventID
```

eventID	Trigger stimulus used for the channel action; see **Details** for possible trigger event IDs

Details

This attribute selects which events cause the channel event detector to enter the detected state. Set this attribute to `0` to start the channel action immediately at the default setting.

Set *`eventID`* to one of the existing trigger event IDs shown in the following table.

Trigger event IDs

Trigger event ID	Description
`digio.trigger[N].EVENT_ID`	An edge (either rising, falling, or either based on the configuration of the line) on the digital input line.
`display.trigger.EVENT_ID`	The trigger key on the front panel is pressed.
`trigger.EVENT_ID`	A `*trg` message on the active command interface. If GPIB is the active command interface, a GET message also generates this event.
`trigger.blender[N].EVENT_ID`	A combination of events has occurred.
`trigger.timer[N].EVENT_ID`	A delay expired.
`tsplink.trigger[N].EVENT_ID`	An edge (either rising, falling, or either based on the configuration of the line) on the TSP-Link trigger line.
`lan.trigger[N].EVENT_ID`	A LAN trigger event has occurred.
`scan.trigger.EVENT_SCAN_READY`	Scan ready event.
`scan.trigger.EVENT_SCAN_START`	Scan start event.
`scan.trigger.EVENT_CHANNEL_READY`	Channel ready event.
`scan.trigger.EVENT_SCAN_COMP`	Scan complete event.
`scan.trigger.EVENT_IDLE`	Idle event.

Example 1

```
scan.trigger.channel.stimulus = scan.trigger.EVENT_SCAN_START
```

Sets the trigger stimulus of the channel event detector to scan start event.

Example 2

```
scan.trigger.channel.stimulus = 0
print(scan.trigger.channel.stimulus)
```

Starts the channel action immediately after the Scan Start Event. This also resets the stimulus to the default.
Output:
`5.000000000e+01`

Also see

scan.trigger.channel.clear() (on page 7-157)
scan.trigger.channel.set() (on page 7-158)
Scan and trigger (on page 2-21)
Trigger model (on page 2-21)

scan.trigger.clear()

This function clears the trigger model.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
scan.trigger.clear()
```

Details

This function sets the arm and channel event detectors of the trigger model to the undetected state.

Example

`scan.trigger.clear()`
Clears the trigger model.

Also see

scan.trigger.channel.set() (on page 7-158)
scan.trigger.channel.stimulus (on page 7-158)
Scan and trigger (on page 2-21)
Trigger model (on page 2-21)

script.anonymous

This is a reference to the anonymous script.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	No	See **Details**	See **Details**	Not applicable

Usage

```
scriptVar = script.anonymous
```

scriptVar	The name of the variable that references the script

Details

You can use the `script.anonymous` script like any other script. To save the anonymous script as a user script, give it a name.

This script is replaced by loading a script with the `loadscript` or `loadandrunscript` commands when they are used without a name.

Example 1

`script.anonymous.list()`
Displays the content of the anonymous script.

Example 2

`print(script.anonymous.source)`
Retrieves the source of the anonymous script.

Also see

Anonymous scripts (on page 6-4)
scriptVar.autorun (on page 7-165)
scriptVar.list() (on page 7-166)
scriptVar.name (on page 7-167)
scriptVar.run() (on page 7-168)
scriptVar.save() (on page 7-168)
scriptVar.source (on page 7-169)

script.delete()

This function deletes a script from nonvolatile memory.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
script.delete("scriptName")
```

scriptName	The string that represents the name of the script

Example

`script.delete("test8")`
Deletes a user script named `test8` from nonvolatile memory.

Also see

Delete user scripts from the instrument (on page 6-44)
scriptVar.save() (on page 7-168)

script.new()

This function creates a script.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
scriptVar = script.new("code")
scriptVar = script.new("code", "name")
```

scriptVar	The name of the variable that references the script
code	A string containing the body of the script
name	The name of the script

Details

The *name* parameter is the name that is added to the `script.user.scripts` table. If *name* is not provided, an empty string is used, and the script is unnamed. If the name already exists in `script.user.scripts`, the `name` attribute of the existing script is set to an empty string before it is replaced by the new script.

The *name* parameter is used for the instrument front-panel display. If this parameter is not defined, the script is not available from the front panel.

You must save the new script into nonvolatile memory to retain it when the instrument is turned off.

Example 1

```
myTest8 = script.new(
   "display.clear() display.settext('Hello from myTest8')", "myTest8")
myTest8()
```

Creates a new script referenced by the variable `myTest8` with the name `myTest8`.
Runs the script. The instrument displays `Hello from myTest8`.

Example 2

```
autoexec = script.new(
   "display.clear() display.settext('Hello from autoexec')", 'autoexec')
```

Creates a new autoexec script that clears the display when the instrument is turned on and displays `Hello from autoexec`.

Also see

Create a script using the script.new() command (on page 6-37)
Global variables and the script.user.scripts table (on page 6-36)
Named scripts (on page 6-4)
scriptVar.save() (on page 7-168)
script.newautorun() (on page 7-162)

script.newautorun()

This function creates a script and enables autorun.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
scriptVar = script.newautorun("code")
scriptVar = script.newautorun("code", "name")
```

scriptVar	The name of the variable that references the script
code	A string that contains the body of the script
name	The name of the script

Details

The *name* parameter is the name that is added to the `script.user.scripts` table. If *name* is not provided, an empty string is used, and the script is unnamed. If the name already exists in `script.user.scripts`, the `name` attribute of the existing script is set to an empty string before it is replaced by the new script.

The *name* parameter is used for the instrument front-panel display. If this parameter is not defined, the script is not available from the front panel.

You must save the new script into nonvolatile memory to retain it when the instrument is turned off.

The script is run automatically immediately after it is created.

This command is the same as the `script.new()` function except that the script is automatically run.

Example

```
NewAuto = script.newautorun("print('Hello from new auto run command')", 'NewAuto')
print(NewAuto.autorun)
print(NewAuto.name)
```

Creates a new script called `NewAuto` that automatically has the autorun attribute set to yes after it is created.
The *name* is set to `NewAuto`.
Output:

```
Hello from new auto run command
yes
NewAuto
```

Also see

Create a script using the script.new() command (on page 6-37)
Global variables and the script.user.scripts table (on page 6-36)
Named scripts (on page 6-4)
script.new() (on page 7-161)
scriptVar.autorun (on page 7-165)
scriptVar.save() (on page 7-168)

script.restore()

This function restores a script that was removed from the runtime environment.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`script.restore(name)`

name	The name of the script to be restored

Details

This command copies the script from nonvolatile memory into the runtime environment. It also creates a global variable with the same name as the name of the script.

Example

`script.restore("test9")`
Restores a script named `test9` from nonvolatile memory.

Also see

script.delete() (on page 7-161)

script.run()

This function runs the anonymous script.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
script.run()
run()
```

Details

Each time the `script.run()` command is given, the anonymous script is executed. This script can be run using this command many times without having to resend it.

Example

`run()`	Runs the anonymous script.

Also see

script.anonymous (on page 7-160)

script.user.catalog()

This function returns an iterator that can be used in a `for` loop to iterate over all the scripts stored in nonvolatile memory.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
for name in script.user.catalog() do body end
```

name	String representing the name of the script
body	Code that implements the body of the `for` loop to process the names in the catalog

Details

This function accesses the catalog of scripts stored in nonvolatile memory, which allows you to process all scripts in nonvolatile memory. The entries are enumerated in no particular order.

Each time the body of the function executes, *name* takes on the name of one of the scripts stored in nonvolatile memory. The `for` loop repeats until all scripts have been iterated.

Example

```
for name in script.user.catalog() do
   print(name)
end
```

Retrieve the catalog listing for user scripts.

Also see

None

scriptVar.autorun

This attribute controls the autorun state of a script.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	No	Not applicable	See **Details**	See **Details**

Usage

scriptVar.autorun = "*state*"
state = *scriptVar*.autorun

scriptVar	The name of the variable that references the script
state	String that indicates whether or not the script runs automatically when powered on: ▪ Script runs automatically: `yes` ▪ Script does not run automatically: `no`

Details

Autorun scripts run automatically when the instrument is turned on. You can set any number of scripts to autorun.

The run order for autorun scripts is arbitrary, so make sure the run order is not important.

The default value for *scriptVar*`.autorun` depends on how the script was loaded. The default is `no` if the script was loaded with `loadscript` or `script.new()`. It is `yes` for scripts loaded with `loadandrunscript` or `script.newautorun()`.

NOTE

Make sure to save the script in nonvolatile memory after setting the `autorun` attribute so that the instrument retains the setting.

Example

```
test5.autorun = "yes"
test5.save()
```

Assume a script named `test5` is in the runtime environment.
The next time the instrument is turned on, `test5` script automatically loads and runs.

Also see

None

scriptVar.list()

This function generates a script listing.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`scriptVar``.list()`

`scriptVar`	The name of the variable that references the script

Details

This function generates output in the form of a sequence of response messages (one message for each line of the script). It also generates output of the script control messages (`loadscript` or `loadandrunscript` and `endscript`).

Example

```
test7 = script.new("display.clear() display.settext('Hello from my test')",
    "test7")
test7()
test7.save()
test7.list()
```

Creates a script named `test7` that displays text on the front panel and lists the script with the following output:

```
loadscript test7
display.clear() display.settext("Hello from my test")
endscript
```

Also see

Retrieve source code one line at a time (on page 6-42)

scriptVar.name

This attribute contains the name of a script in the runtime environment.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	No	Not applicable	Not applicable	Not applicable

Usage

```
scriptVar.name = "scriptName"
scriptName = scriptVar.name
```

scriptVar	Name of the variable that references the script
scriptName	A string that represents the name of the script

Details

When setting the script name, this attribute renames the script that the variable *scriptVar* references.

This attribute must be either a valid Lua identifier or the empty string. Changing the name of a script changes the index that is used to access the script in the `script.user.scripts` table. Setting the attribute to an empty string removes the script from the table completely and the script becomes an unnamed script.

As long as there are variables referencing an unnamed script, the script can be accessed through those variables. When all variables that reference an unnamed script are removed, the script is removed from the runtime environment.

If the new name is the same as a name that is already used for another script, the name of the other script is set to an empty string, and that script becomes unnamed.

NOTE

Changing the name of a script does not change the name of any variables that reference that script. The variables still reference the script, but the names of the script and variables may not match.

Example

```
test7 = script.new("display.clear() display.settext('Hello from my test')", "")
test7()
print(test7.name)

test7.name = "test7"
print(test7.name)

test7.save()
```

This example calls the `script.new()` function to create a script with no name, runs the script, names the script `test7`, and then saves the script in nonvolatile memory.

Also see

Rename a script (on page 6-40)
script.new() (on page 7-161)
scriptVar.save() (on page 7-168)

scriptVar.run()

This function runs a script.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

scriptVar.run()
scriptVar()

scriptVar	The name of the variable that references the script

Details

The *scriptVar*.run() function runs the script referenced by *scriptVar*. You can also run the script by using *scriptVar()*.

Example

test8.run()
Runs the script referenced by the variable test8.

Also see

None

scriptVar.save()

This function saves the script to nonvolatile memory.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

scriptVar.save()

scriptVar	The name of the variable that references the script

Example

test8.save()
Saves the script referenced by the variable test8 to nonvolatile memory.

Also see

Save a user script (on page 6-10)

scriptVar.source

This attribute contains the source code of a script.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW) (see **Details)**	No	Not applicable	Not saved	Not applicable

Usage

```
code = scriptVar.source
scriptVar.source = nil
```

scriptVar	The name of the variable that references the script that contains the source code
code	A string that contains the body of the script

Details

The `loadscript` or `loadandrunscript` and `endscript` keywords are not included in the source code.

The body of the script is a single string with lines separated by the newline character.

The instrument automatically stores the source for all scripts that are loaded on the instrument. To free up memory or to obfuscate the code, assign `nil` to the source attribute of the script. Although this attribute is writable, it can only be set to the `nil` value.

Example

`test7 = script.new("display.clear() display.settext('Hello from my test')", "")` `print(test7.source)`
This example creates a script called `test7` that displays a message on the front panel and retrieves the source code. Output: `display.clear() display.settext('Hello from my test')`

Also see

scriptVar.list() (on page 7-166)

settime()

This function sets the real-time clock (sets the present time of the system).

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
settime(time)
```

time	The time in seconds since January 1, 1970 UTC

Details

This function sets the date and time of the instrument based on the *time* parameter (specified in UTC time). UTC time is specified as the number of seconds since Jan 1, 1970, UTC. You can use UTC time from a local time specification, or you can use UTC time from another source (for example, your computer).

Example

```
systemTime = os.time({year = 2020,
       month = 3,
       day = 31,
       hour = 14,
       min = 25})
settime(systemTime)
```

Sets the date and time to Mar 31, 2020 at 2:25 pm.

Also see

gettimezone() (on page 7-97)
settimezone() (on page 7-170)

settimezone()

This function sets the local time zone.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
settimezone(offset)
settimezone("offset", "dstOffset", "dstStart", "dstEnd")
```

offset	String representing offset from UTC
dstOffset	String representing the daylight savings offset from UTC
dstStart	String representing when daylight savings time starts
dstEnd	String representing when daylight savings time ends

Details

You only need to set the time zone if you use the `os.time()` and `os.date()` functions.

If only one parameter is given, the same time offset is used throughout the year. If four parameters are given, time is adjusted twice during the year for daylight savings time.

offset and *dstOffset* are strings of the form "`[+|-]hh[:mm[:ss]]`" that indicate how much time must be added to the local time to get UTC time:

- `hh` is a number between 0 and 23 that represents hours
- `mm` is a number between 0 and 59 that represents minutes
- `ss` is a number between 0 and 59 that represents seconds

The minute, second, +, and − fields are optional.

For example, to set the UTC-5 time zone, you specify the string `"5"`, because UTC-5 is 5 hours behind UTC and you must add 5 hours to the local time to determine UTC time. To specify the time zone UTC4, you specify `"-4"`, because UTC4 is 4 hours ahead of UTC and 4 hours must be subtracted from the local time to determine UTC.

dstStart and *dstEnd* are strings of the form `"MM.w.dw/hh[:mm[:ss]]"` that indicate when daylight savings time begins and ends respectively:

- `MM` is a number between 1 and 12 that represents the month
- `w` is a number between 1 and 5 that represents the week in the month
- `dw` is a number between 0 and 6 that represents the day of the week (where 0 is Sunday)

The rest of the fields represent the time of day that the change takes effect:

- `hh` represents hours
- `mm` represents minutes
- `ss` represents seconds

The minutes and seconds fields are optional.

The week of the month and day of the week fields are not specific dates.

Example

```
settimezone("8", "1", "3.3.0/02", "11.2.0/02")
   settimezone(offset)
```

Sets `offset` to equal +8 hours, +1 hour for DST, starts on Mar 14 at 2:00 am, ends on Nov 7 at 2:00 am. Sets local time zone to `offset`.

Also see

gettimezone() (on page 7-97)
settime() (on page 7-169)

slot[slot].idn

This attribute returns a string that contains information about the matrix card.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
idnString = slot[slot].idn
```

idnString	The return string
slot	Slot number: `1` for 708B; `1` to `6` for 707B

Details

The information that is returned depends on whether the matrix card in the slot is a physical card or pseudocard.

For physical cards, this returns a comma-separated string that contains the model number, description, firmware revision, and serial number of the matrix card installed in the specified slot.

For pseudocards, the response is `Pseudo`, followed by the model number, description, and `???` for the firmware revision and serial number.

Example

```
print(slot[1].idn)
```

If a 7173 module is installed in slot 1, the response is:
`7173,4x12 Hi Freq Matrix AAAA,02.01a,99999999`

Also see

None

slot[slot].poles.four

This attribute indicates if a four-pole setting is supported for the channels on the card.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
fourPole = slot[slot].poles.four
```

fourPole	The return value
slot	Slot number: `1` for 708B; `1` to `6` for 707B

Details

This attribute only exists if a card is installed and if the card supports four-pole settings for the channels on the card. If not, the value is `nil`. If supported, the value is `1`.

Example

```
fourPole3 = slot[3].poles.four
print(fourPole3)
```

Queries if slot 3 supports four-pole settings for the channels on the card.
Output if card supports four pole:
`1.00000000000e+00`
Output if card does not support four pole:
`nil`

Also see

slot[slot].poles.one (on page 7-173)
slot[slot].poles.two (on page 7-173)

slot[slot].poles.one

This attribute indicates if a one-pole setting is supported for the channels on the specified card.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

`onePole = slot[slot].poles.one`

onePole	The return value
slot	Slot number: `1` for 708B; `1` to `6` for 707B

Details

This attribute only exists if a card is installed and if the card supports one-pole settings for the channels on the card. If not, the value is `nil`. If supported, the value is `1`.

Example

```
print(slot[3].poles.one)
```

Query to see if slot 3 supports one-pole settings for the channels on the card.
Output if card supports one pole:
`1.00000000000e+00`
Output if card does not support one pole:
`nil`

Also see

slot[slot].poles.four (on page 7-172)
slot[slot].poles.two (on page 7-173)

slot[slot].poles.two

This attribute indicates if a two-pole setting is supported for the channels on the card.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

`twoPole = slot[slot].poles.two`

twoPole	The return value
slot	Slot number: `1` for 708B; `1` to `6` for 707B

Details

This attribute only exists if a card is installed and if the card supports a two-pole setting for the channels on the card.

If not, the value is `nil`. If supported, the value is `1`.

Example

`twoPole3 = slot[3].poles.two` `print(twoPole3)`	

Query to see if slot 3 supports two-pole settings for the channels on the card.
Output if card supports two pole:
`1.00000000000e+00`
Output if card does not support two pole:
`nil`	

Also see

slot[slot].poles.one (on page 7-173)
slot[slot].poles.four (on page 7-172)

slot[slot].pseudocard

This attribute specifies a pseudocard to implement.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Restore configuration Instrument reset Power cycle	Configuration script	slot.PSEUDO_NONE

Usage

```
pseudoCard = slot[slot].pseudocard
slot[slot].pseudocard = pseudoCard
```

pseudoCard	The pseudocards available for 707B and 708B are: ■ No pseudocard selection: `slot.PSEUDO_NONE` or `0` ■ 7072 8 × 12 Semiconductor Matrix Card: `7072` ■ 7072-HV 8×12 High-Voltage Semiconductor Matrix Card: `70721` ■ 7173-50 8×12 High-Frequency 2-Pole 4x12 Matrix Card: `7173` ■ 7174A 8×12 Low-Current, High-Speed Matrix Card: `7174`
slot	Slot number: `1` for 708B; `1` to `6` for 707B

Details

Pseudocards allow you to configure your system without having an actual matrix card installed in your system. You can perform open, close, and scan operations and configure your system with pseudocards.

This command is only applicable to a slot that does not have a matrix card or pseudocard installed. If a pseudocard is presently assigned to the slot, you must set the slot to no pseudocard before assigning the new pseudocard.

After assigning a pseudocard, you can use valid commands for the matrix card for that slot.

Changing the pseudocard assignment from a pseudocard to no pseudocard invalidates scan lists that include that slot.

If a card is installed in the slot, this command returns `nil`. If the slot is empty and no pseudocard is installed, the return is `0`.

Example 1

```
myPseudoCard = slot[3].pseudocard
if myPseudoCard == 7072 then
   print("Pseudo-7072 in Slot #3")
end
```

If slot 3 is configured to have a 7072 pseudocard, the following message is output:

```
Pseudo-7072 in Slot #3
```

Example 2

```
slot[1].pseudocard = 0
print(slot[1].idn)
slot[1].pseudocard = 7072
print(slot[1].idn)

slot[1].pseudocard = 0
print(slot[1].idn)
slot[1].pseudocard = 7070
print(slot[1].idn)
```

This example requires an empty slot.
The slot is set to empty and then set to a valid value.
Output:

```
Empty Slot
7072,Pseudo 8x12 SemiMatrix,00.00a,????????
```

To change the pseudocard, set the slot to empty again, then define the new card.
Output:

```
Empty Slot
7070,Universal Adapter Card,00.00a,????????
```

Also see

slot[slot].idn (on page 7-171)

status.condition

This attribute stores the status byte condition register.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not saved	Not applicable

Usage

```
statusByte = status.condition
```

statusByte	The status byte; a zero (0) indicates no bits set; other values indicate bit settings

Details

This attribute is used to read the status byte, which is returned as a numeric value. The binary equivalent of the value of this attribute indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B7. For example, if a value of `1.29000e+02` (which is 129) is read as the value of this register, the binary equivalent is 1000 0001. This value indicates that bit B0 and bit B7 are set.

B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	*
1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

The returned value can indicate one or more status events occurred. When an enabled status event occurs, a summary bit is set in this register to indicate the event occurrence.

The individual bits of this register have the meanings described in the following table.

Bit	Value and description

B0	`status.MEASUREMENT_SUMMARY_BIT`
	`status.MSB`
	Set summary bit indicates that an enabled measurement event has occurred.
	Bit B0 decimal value: 1

B1	`status.SYSTEM_SUMMARY_BIT`
	`status.SSB`
	Set summary bit indicates that an enabled system event has occurred.
	Bit B1 decimal value: 2

B2	`status.ERROR_AVAILABLE`
	`status.EAV`
	Set summary bit indicates that an error or status message is present in the error queue.
	Bit B2 decimal value: 4

B3	`status.QUESTIONABLE_SUMMARY_BIT`
	`status.QSB`
	Set summary bit indicates that an enabled questionable event has occurred.
	Bit B3 decimal value: 8

B4	`status.MESSAGE_AVAILABLE`
	`status.MAV`
	Set summary bit indicates that a response message is present in the output queue.
	Bit B4 decimal value: 16

B5	`status.EVENT_SUMMARY_BIT`
	`status.ESB`
	Set summary bit indicates that an enabled standard event has occurred.
	Bit B5 decimal value: 32

B6	`status.RQS`
	`status.MASTER_SUMMARY_STATUS`
	`status.MSS`
	Request Service (RQS)/Master Summary Status (MSS). Depending on how it is used, bit B6 of the status byte register is either the Request for Service (RQS) bit or the Master Summary Status (MSS) bit:
	■ When using the GPIB, USB, or VXI-11 serial poll sequence of the Models 707B and 708B to obtain the status byte (serial poll byte), B6 is the RQS bit. The set bit indicates that the Request Service (RQS) bit of the status byte (serial poll byte) is set and a serial poll (SRQ) has occurred.
	■ When using the `status.condition` register command or the *STB? common command to read the status byte, B6 is the MSS bit. Set bit indicates that an enabled summary bit of the status byte register is set.
	Bit B6 decimal value: 64

B7	`status.OPERATION_SUMMARY_BIT`
	`status.OSB`
	Set summary bit indicates that an enabled operation event has occurred.
	Bit B7 decimal value: 128

In addition to the above constants, when more than one bit of the register is set, *statusByte* equals the sum of their decimal weights. For example, if 129 is returned, bits B0 and B7 are set (1 + 128).

Bit	**B7**	**B6**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Example

```
statusByte = status.condition
print(statusByte)
```

Returns `statusByte`.
Sample output:
`1.29000e+02`
Converting this output (129) to its binary equivalent yields 1000 0001
Therefore, this output indicates that the set bits of the status byte condition register are presently B0 (MSS) and B7 (OSB).

Also see

Status byte and service request (SRQ) (on page 12-15)

status.node_enable

This attribute stores the system node enable register.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Status reset	Not saved	0

Usage

```
nodeEnableRegister = status.node_enable
status.node_enable = nodeEnableRegister
```

nodeEnableRegister	The status of the system node enable register; a zero (0) indicates no bits set (also send 0 to clear all bits); other values indicate various bit settings

Details

This attribute is used to read or write to the system node enable register. Reading the system node enable register returns a value. The binary equivalent of the value of this attribute indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B7. For example, if a value of `1.29000e+02` (which is 129) is read as the value of this register, the binary equivalent is 1000 0001. This value indicates that bit B0 and bit B7 are set.

B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	*
1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

Assigning a value to this attribute enables one or more status events. When an enabled status event occurs, a summary bit is set in the appropriate system summary register. The register and bit that is set depends on the TSP-Link node number assigned to this instrument.

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value and description

B0	`status.MEASUREMENT_SUMMARY_BIT`
	`status.MSB`
	Set summary bit indicates that an enabled measurement event has occurred.
	Bit B0 decimal value: 1

B1	Not used

B2	`status.ERROR_AVAILABLE`
	`status.EAV`
	Set summary bit indicates that an error or status message is present in the error queue.
	Bit B2 decimal value: 4

B3	`status.QUESTIONABLE_SUMMARY_BIT`
	`status.QSB`
	Set summary bit indicates that an enabled questionable event has occurred.
	Bit B3 decimal value: 8

B4	`status.MESSAGE_AVAILABLE`
	`status.MAV`
	Set summary bit indicates that a response message is present in the output queue.
	Bit B4 decimal value: 16

B5	`status.EVENT_SUMMARY_BIT`
	`status.ESB`
	Set summary bit indicates that an enabled standard event has occurred.
	Bit B5 decimal value: 32

B6	`status.MASTER_SUMMARY_STATUS`
	`status.MSS`
	Set bit indicates that an enabled Master Summary Status (MSS) bit of the Status Byte Register is set.
	Bit B6 decimal value: 64

B7	`status.OPERATION_SUMMARY_BIT`
	`status.OSB`
	Set summary bit indicates that an enabled operation event has occurred.
	Bit B7 decimal value: 128

As an example, to set the B0 bit of the system node enable register, set `status.node_enable = status.MSB`.

In addition to the above values, *nodeEnableRegister* can be set to the numeric equivalent of the bit to set. To set more than one bit of the register, set *nodeEnableRegister* to the sum of their decimal weights. For example, to set bits B0 and B7, set *nodeEnableRegister* to 129 (1 + 128).

Bit	B7	B6	B5	B4	B3	B2	B1	B0
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Example 1

```
nodeEnableRegister = status.MSB + status.OSB
status.node_enable = nodeEnableRegister
```

Use constants to set the MSB and OSB bits of the system node enable register.

Example 2

```
-- decimal 129 = binary 10000001
nodeEnableRegister = 129
status.node_enable = nodeEnableRegister
```

Sets the MSB and OSB bits of the system node enable register using a decimal value.

Also see

status.condition (on page 7-175)
status.system.* (on page 7-195)
Status byte and service request (SRQ) (on page 12-15)

status.node_event

This attribute stores the status node event register.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not saved	0

Usage

```
nodeEventRegister = status.node_event
```

nodeEventRegister	The status of the node event register; a zero (0) indicates no bits set; other values indicate various bit settings

Details

This attribute is used to read the status node event register, which is returned as a numeric value (reading this register returns a value). The binary equivalent of the value of this attribute indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B7. For example, if a value of `1.29000e+02` (which is 129) is read as the value of this register, the binary equivalent is 1000 0001. This value indicates that bit B0 and bit B7 are set.

B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	*
1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

The returned value can indicate one or more status events occurred.

Bit	Value and description

B0	`status.MEASUREMENT_SUMMARY_BIT`
	`status.MSB`
	Set summary bit indicates that an enabled measurement event has occurred.
	`Bit B0 decimal value: 1`

B1	Not used

B2	`status.ERROR_AVAILABLE`
	`status.EAV`
	Set summary bit indicates that an error or status message is present in the error queue.
	Bit B2 decimal value: 4

B3	`status.QUESTIONABLE_SUMMARY_BIT`
	`status.QSB`
	Set summary bit indicates that an enabled questionable event has occurred.
	Bit B3 decimal value: 8

B4	`status.MESSAGE_AVAILABLE`
	`status.MAV`
	Set summary bit indicates that a response message is present in the output queue.
	Bit B4 decimal value: 16

B5	`status.EVENT_SUMMARY_BIT`
	`status.ESB`
	Set summary bit indicates that an enabled standard event has occurred.
	Bit B5 decimal value: 32

B6	`status.MASTER_SUMMARY_STATUS`
	`status.MSS`
	Set bit indicates that an enabled Master Summary Status (MSS) bit of the Status Byte register is set.
	Bit B6 decimal value: 64

B7	`status.OPERATION_SUMMARY_BIT`
	`status.OSB`
	Set summary bit indicates that an enabled operation event has occurred.
	Bit B7 decimal value: 128

In addition to the above constants, *`nodeEventRegister`* can be set to the decimal equivalent of the bits set. When more than one bit of the register is set, *`nodeEventRegister`* contains the sum of their decimal weights. For example, if 129 is returned, bits B0 and B7 are set (1 + 128).

Bit	**B7**	**B6**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Example

```
nodeEventRegister = status.node_event
print(nodeEventRegister)
```

Reads the status node event register.
Sample output:
`1.29000e+02`
Converting this output (129) to its binary equivalent yields `1000 0001`. Therefore, this output indicates that the set bits of the status byte condition register are presently B0 (MSB) and B7 (OSB).

Also see

Status byte and service request (SRQ) (on page 12-15)
status.condition (on page 7-175)
status.system.* (on page 7-195)

status.operation.*

These attributes manage the Operation Status Register set of the status model.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute	- -	- -	- -	- -
.condition (R)	Yes	Not applicable	Not saved	Not applicable
.enable (RW)	Yes	Status reset	Not saved	0
.event (R)	Yes	Status reset	Not saved	0
.ntr (RW)	Yes	Status reset	Not saved	0
.ptr (RW)	Yes	Status reset	Not saved	22,528 (All bits set)

Usage

```
operationRegister = status.operation.condition
operationRegister = status.operation.enable
operationRegister = status.operation.event
operationRegister = status.operation.ntr
operationRegister = status.operation.ptr
status.operation.enable = operationRegister
status.operation.ntr = operationRegister
status.operation.ptr = operationRegister
```

operationRegister	The status of the operation status register; a zero (0) indicates no bits set (also send 0 to clear all bits); other values indicate specific bit settings

Details

These attributes read or write the Operation Status Registers.

Reading a status register returns a value. The binary equivalent of the returned value indicates which register bits are set. The least significant bit of the binary number is bit B0, and the most significant bit is bit B15. For example, if a value of `2.04800e+04` (which is 20,480) is read as the value of the condition register, the binary equivalent is 0101 0000 0000 0000. This value indicates that bit B14 (`PROGRAM_RUNNING`) and bit B12 (USER) are set.

B15 **	B14 >	B13 >	B12 >	B11 >	B10 >	B9 >	B8 >	B7 >	B6 >	B5 >	B4 >	B3 >	B2 >	B1 >	B0 *
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value and description
B0 to B10	Not used

B11	`status.operation.PROMPTS`
	`status.operation.PRMPTS`
	Set bit indicates that command prompts are enabled.
	Bit B11 decimal value: 2,048

B12	`status.USER` Set bit indicates that an enabled bit in the `status.operation.user` register is set. Bit B12 decimal value: 4,096
B13	Not used.

B14	`status.operation.PROGRAM_RUNNING`
	`status.operation.PROG`
	Set bit indicates that a program is running.
	Bit B14 decimal value: 16,384

B15	Not used

As an example, to set bit B12 of the Operation Status Enable Register, set `status.operation.enable = status.operation.USER`.

In addition to the above constants, *`operationRegister`* can be set to the numeric equivalent of the bit to set. To set more than one bit of the register, set *`operationRegister`* to the sum of their decimal weights. For example, to set bits B12 and B14, set *`operationRegister`* to 20,480 (which is the sum of 4,096 + 16,384).

Bit	B7	B6	B5	B4	B3	B2	B1	B0
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Bit	B15	B14	B13	B12	B11	B10	B9	B8
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	32,768	16,384	8,192	4,096	2,048	1,024	512	256
Weights	(2^{15})	(2^{14})	(2^{13})	(2^{12})	(2^{11})	(2^{10})	(2^9)	(2^8)

Example 1

```
operationRegister = status.operation.USER + status.operation.PROG
status.operation.enable = operationRegister
```

Uses constants to set the USER and PROG bits of the Operation Status Enable Register.

Example 2

```
-- decimal 20480 = binary 0101 0000 0000 0000
operationRegister = 20480
status.operation.enable = operationRegister
```

Uses a decimal value to set the USER and PROG bits of the Operation Status Enable Register.

Also see

None

status.operation.user.*

These attributes manage the operation status user register set of the status model.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute	- -	- -	- -	- -
.condition (RW)	Yes	Status reset	Not saved	0
.enable (RW)	Yes	Status reset	Not saved	0
.event (R)	Yes	Status reset	Not saved	0
.ntr (RW)	Yes	Status reset	Not saved	0
.ptr (RW)	Yes	Status reset	Not saved	32,767 (All bits set)

Usage

```
operationRegister = status.operation.user.condition
operationRegister = status.operation.user.enable
operationRegister = status.operation.user.event
operationRegister = status.operation.user.ntr
operationRegister = status.operation.user.ptr
status.operation.user.condition = operationRegister
status.operation.user.enable = operationRegister
status.operation.user.ntr = operationRegister
status.operation.user.ptr = operationRegister
```

operationRegister	The status of the operation status user register; a zero (0) indicates no bits set (also send 0 to clear all bits); other values indicate various bit settings

Details

These attributes are used to read or write to the operation status user registers. Reading a status register returns a value. The binary equivalent of the value indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B15. For example, if a value of `1.29000e+02` (which is 129) is read as the value of the condition register, the binary equivalent is 0000 0000 1000 0001. This value indicates that bits B0 and B7 are set.

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	>	>	>	>	>	>	>	>	*
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value	Decimal value
B0	`status.operation.user.BIT0`	1
B1	`status.operation.user.BIT1`	2
B2	`status.operation.user.BIT2`	4
B3	`status.operation.user.BIT3`	8
B4	`status.operation.user.BIT4`	16
B5	`status.operation.user.BIT5`	32
B6	`status.operation.user.BIT6`	64
B7	`status.operation.user.BIT7`	128
B8	`status.operation.user.BIT8`	256
B9	`status.operation.user.BIT9`	512
B10	`status.operation.user.BIT10`	1,024
B11	`status.operation.user.BIT11`	2,048

Bit	Value	Decimal value
B12	`status.operation.user.BIT12`	4,096
B13	`status.operation.user.BIT13`	8,192
B14	`status.operation.user.BIT14`	16,384
B15	Not used	Not applicable

As an example, to set bit B0 of the operation status user enable register, set `status.operation.user.enable = status.operation.user.BIT0`.

In addition to the above constants, *`operationRegister`* can be set to the decimal value of the bit to set. To set more than one bit of the register, set *`operationRegister`* to the sum of their decimal values. For example, to set bits B11 and B14, set *`operationRegister`* to 18,432 (which is the sum of 2,048 + 16,384).

Bit	B7	B6	B5	B4	B3	B2	B1	B0
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Bit	B15	B14	B13	B12	B11	B10	B9	B8
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	32,768	16,384	8,192	4,096	2,048	1,024	512	256
Weights	(2^{15})	(2^{14})	(2^{13})	(2^{12})	(2^{11})	(2^{10})	(2^9)	(2^8)

Example 1

```
operationRegister = status.operation.user.BIT11 + status.operation.user.BIT14
status.operation.user.enable = operationRegister
```

Uses constants to set bits B11 and B14 of the operation status user enable register.

Example 2

```
-- 18432 = binary 0100 1000 0000 0000
operationRegister = 18432
status.operation.enable = operationRegister
```

Uses the decimal value to set bits B11 and B14 of the operation status user enable register.

Also see

status.operation.* (on page 7-181)

status.questionable.*

These attributes manage the questionable status register set of the status model.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute	- -	- -	- -	- -
.condition (R)	Yes	Not applicable	Not saved	Not applicable
.enable (RW)	Yes	Status reset	Not saved	0
.event (R)	Yes	Status reset	Not saved	0
.ntr (RW)	Yes	Status reset	Not saved	0
.ptr (RW)	Yes	Status reset	Not saved	32,256 (All bits set)

Usage

```
questionableRegister = status.questionable.condition
questionableRegister = status.questionable.enable
questionableRegister = status.questionable.event
questionableRegister = status.questionable.ntr
questionableRegister = status.questionable.ptr
status.questionable.enable = questionableRegister
status.questionable.ntr = questionableRegister
status.questionable.ptr = questionableRegister
```

questionableRegister	The status of the questionable status register; a zero (0) indicates no bits set (also send 0 to clear all bits); other values indicate various bit settings

Details

These attributes are used to read or write to the questionable status registers. Reading a status register returns a value. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B15. For example, if a value of 1.22880e+04 (which is 12,288) is read as the value of the condition register, the binary equivalent is 0011 0000 0000 0000. This value indicates that bits B12 and B13 are set.

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	>	>	>	>	>	>	>	>	*
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value	Decimal value
B0 to B8	Not used	Not available
B9	`status.questionable.S1THR` `status.questionable.SLOT1_THERMAL`	512
B10	`status.questionable.S2THR` `status.questionable.SLOT2_THERMAL`	1,024
B11	`status.questionable.S3THR` `status.questionable.SLOT3_THERMAL`	2,048
B12	`status.questionable.S4THR` `status.questionable.SLOT4_THERMAL`	4,096
B13	`status.questionable.S5THR` `status.questionable.SLOT5_THERMAL`	8,192
B14	`status.questionable.S6THR` `status.questionable.SLOT6_THERMAL`	16,384
B15	Not used	Not available

As an example, to set bit B9 of the questionable status enable register, set `status.questionable.enable = status.questionable.SLOT1_THERMAL`.

In addition to the above constants, *questionableRegister* can be set to the numeric equivalent of the bit to set. To set more than one bit of the register, set *questionableRegister* to the sum of their decimal weights. For example, to set bits B12 and B13, set *questionableRegister* to 12,288 (which is the sum of 4,096 + 8,192).

Bit	B7	B6	B5	B4	B3	B2	B1	B0
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Bit	B15	B14	B13	B12	B11	B10	B9	B8
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	32,768	16,384	8,192	4,096	2,048	1,024	512	256
Weights	(2^{15})	(2^{14})	(2^{13})	(2^{12})	(2^{11})	(2^{10})	(2^9)	(2^8)

Example 1

```
questionableRegister = status.questionable.S1THR + status.questionable.S3THR
status.questionable.enable = questionableRegister
```

Uses constants to set bits B9 and B11 of the status questionable enable register.

Example 2

```
-- decimal 2560 = binary 00001010 0000 0000
questionableRegister = 2560
status.questionable.enable = questionableRegister
```

Uses a decimal value to set bits B9 and B11 of the status questionable enable register.

Also see

None

status.request_enable

This attribute stores the service request (SRQ) enable register.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Status reset	Not saved	0

Usage

```
requestSRQEnableRegister = status.request_enable
status.request_enable = requestSRQEnableRegister
```

requestSRQEnableRegister	The status of the service request (SRQ) enable register; a zero (0) indicates no bits set (also send 0 to clear all bits); other values indicate various bit settings

Details

This attribute is used to read or write to the service request enable register. Reading the service request enable register returns a value. The binary equivalent of the value of this attribute indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B7. For example, if a value of `1.29000e+02` (which is 129) is read as the value of this register, the binary equivalent is 1000 0001. This value indicates that bit B0 and bit B7 are set.

B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	*
1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value

B0	`status.MEASUREMENT_SUMMARY_BIT`
	`status.MSB`
	Set summary bit indicates that an enabled event in the Measurement Event Register has occurred.
	Bit B0 decimal value: 1

B1	`status.SYSTEM_SUMMARY_BIT`
	`status.SSB`
	Set summary bit indicates that an enabled event in the System Summary Register has occurred.
	Bit B1 decimal value: 2

B2	`status.ERROR_AVAILABLE`
	`status.EAV`
	Set summary bit indicates that an error or status message is present in the error queue.
	Bit B2 decimal value: 4

Bit	Value

B3	`status.QUESTIONABLE_SUMMARY_BIT`
	`status.QSB`
	Set summary bit indicates that an enabled event in the Questionable Status Register has occurred.
	Bit B3 decimal value: 8

B4	`status.MESSAGE_AVAILABLE`
	`status.MAV`
	Set summary bit indicates that a response message is present in the output queue.
	Bit B4 decimal value: 16

B5	`status.EVENT_SUMMARY_BIT`
	`status.ESB`
	Set summary bit indicates that an enabled event in the Standard Event Status Register has occurred.
	Bit B5 decimal value: 32

B6	Not used

B7	`status.OPERATION_SUMMARY_BIT`
	`status.OSB`
	Set summary bit indicates that an enabled event in the Operation Status Register has occurred.
	Bit B7 decimal value: 128

As an example, to set bit B0 of the service request enable register, set `status.request_enable = status.MSB`.

In addition to the above values, *requestSRQEnableRegister* can be set to the numeric equivalent of the bit to set. To set more than one bit of the register, set *requestSRQEnableRegister* to the sum of their decimal weights. For example, to set bits B0 and B7, set *requestSRQEnableRegister* to 129 (1 + 128).

Bit	**B7**	**B6**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Example 1

```
requestSRQEnableRegister = status.MSB + status.OSB
status.request_enable = requestSRQEnableRegister
```

Uses constants to set the MSB and OSB bits of the service request (SRQ) enable register.

Example 2

```
-- decimal 129 = binary 10000001
requestSRQEnableRegister = 129
status.request_enable = requestSRQEnableRegister
```

Uses a decimal value to set the MSB and OSB bits of the service request (SRQ) enable register.

Also see

Status byte and service request (SRQ) (on page 12-15)
status.condition (on page 7-175)
status.system.* (on page 7-195)

status.request_event

This attribute stores the service request (SRQ) event register.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not saved	0

Usage

```
requestSRQEventRegister = status.request_event
```

requestSRQEventRegister	The status of the request event register; a zero (0) indicates no bits set; other values indicate various bit settings

Details

This attribute is used to read the service request event register, which is returned as a numeric value. Reading this register returns a value. The binary equivalent of the value of this attribute indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B7. For example, if a value of `1.29000e+02` (which is 129) is read as the value of this register, the binary equivalent is 1000 0001. This value indicates that bit B0 and bit B7 are set.

B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	*
1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

The returned value can indicate one or more status events occurred.

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value and description

B0	`status.MEASUREMENT_SUMMARY_BIT`
	`status.MSB`
	Set summary bit indicates that an enabled event in the Measurement Event Register has occurred.
	Bit B0 decimal value: 1

B1	`status.SYSTEM_SUMMARY_BIT`
	`status.SSB`
	Set summary bit indicates that an enabled event in the System Summary Register has occurred.
	Bit B1 decimal value: 2

B2	`status.ERROR_AVAILABLE`
	`status.EAV`
	Set summary bit indicates that an error or status message is present in the error queue.
	Bit B2 decimal value: 4

Bit	Value and description

B3	`status.QUESTIONABLE_SUMMARY_BIT`
	`status.QSB`
	Set summary bit indicates that an enabled event in the Questionable Status Register has occurred.
	Bit B3 decimal value: 8

B4	`status.MESSAGE_AVAILABLE`
	`status.MAV`
	Set summary bit indicates that a response message is present in the output queue.
	Bit B4 decimal value: 16

B5	`status.EVENT_SUMMARY_BIT`
	`status.ESB`
	Set summary bit indicates that an enabled event in the Standard Event Status Register has occurred.
	Bit B5 decimal value: 32

B6	Not used

B7	`status.OPERATION_SUMMARY_BIT`
	`status.OSB`
	Set summary bit indicates that an enabled event in the Operation Status Register has occurred.
	Bit B7 decimal value: 128

In addition to the above constants, *`requestEventRegister`* can be set to the decimal equivalent of the bits set. When more than one bit of the register is set, *`requestEventRegister`* contains the sum of their decimal weights. For example, if 129 is returned, bits B0 and B7 are set (1 + 128).

Bit	B7	B6	B5	B4	B3	B2	B1	B0
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Example

```
requestEventRegister = status.request_event
print(requestEventRegister)
```

Reads the status request event register.
Sample output:
`1.29000e+02`
Converting this output (129) to its binary equivalent yields 1000 0001.
Therefore, this output indicates that the set bits of the status request event register are presently B0 (MSB) and B7 (OSB).

Also see

status.condition (on page 7-175)
status.system.* (on page 7-195)
Status byte and service request (SRQ) (on page 12-15)

status.reset()

This function resets all bits in the status model.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
status.reset()
```

Details

This function clears all status data structure registers (enable, event, NTR, and PTR) to their default values. For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20).

Example

`status.reset()`
Resets the instrument status model.

Also see

Status model (on page 12-1)

status.standard.*

These attributes manage the standard event status register set of the status model.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute	- -	- -	- -	- -
.condition (R)	Yes	Not applicable	Not saved	Not applicable
.enable (RW)	Yes	Status reset	Not saved	0
.event (R)	Yes	Status reset	Not saved	0
.ntr (RW)	Yes	Status reset	Not saved	0
.ptr (RW)	Yes	Status reset	Not saved	253 (All bits set)

Usage

```
standardRegister = status.standard.condition
standardRegister = status.standard.enable
standardRegister = status.standard.event
standardRegister = status.standard.ntr
standardRegister = status.standard.ptr
status.standard.enable = standardRegister
status.standard.ntr = standardRegister
status.standard.ptr = standardRegister
```

standardRegister	The status of the standard event status register; a zero (0) indicates no bits set (also send 0 to clear all bits); other values indicate various bit settings

Details

These attributes are used to read or write to the standard event status registers. Reading a status register returns a value. The binary equivalent of the returned value indicates which register bits are set. The least significant bit of the binary number is bit B0, and the most significant bit is bit B15. For example, if a value of `1.29000e+02` (which is 129) is read as the value of the condition register, the binary equivalent is 0000 0000 1000 0001. This value indicates that bit B0 and bit B7 are set.

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	>	>	>	>	>	>	>	>	*
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value

B0	`status.standard.OPERATION_COMPLETE`
	`status.standard.OPC`
	Set bit indicates that all pending selected instrument operations are completed and the instrument is ready to accept new commands. The bit is set in response to an `*OPC` command. The `opc()` function can be used in place of the `*OPC` command.
	Bit B0 decimal value: 1

B1	Not used

B2	`status.standard.QUERY_ERROR`
	`status.standard.QYE`
	Set bit indicates that you attempted to read data from an empty Output Queue.
	Bit B2 decimal value: 4

B3	`status.standard.DEVICE_DEPENDENT_ERROR`
	`status.standard.DDE`
	Set bit indicates that an instrument operation did not execute properly due to some internal condition.
	Bit B3 decimal value: 8

B4	`status.standard.EXECUTION_ERROR`
	`status.standard.EXE`
	Set bit indicates that the instrument detected an error while trying to execute a command.
	Bit B4 decimal value: 16

B5	`status.standard.COMMAND_ERROR`
	`status.standard.CME`
	Set bit indicates that a command error has occurred. Command errors include:
	IEEE Std 488.2 syntax error: Instrument received a message that does not follow the defined syntax of the IEEE Std 488.2 standard.
	Semantic error: Instrument received a command that was misspelled or received an optional IEEE Std 488.2 command that is not implemented.
	GET error: The instrument received a Group Execute Trigger (GET) inside a program message.
	Bit B5 decimal value: 32

Bit	Value

B6	`status.standard.USER_REQUEST`
	`status.standard.URQ`
	Set bit indicates that the LOCAL key on the instrument front panel was pressed.
	Bit B6 decimal value: 64

B7	`status.standard.POWER_ON`
	`status.standard.PON`
	Set bit indicates that the instrument has been turned off and turned back on since the last time this register has been read.
	Bit B7 decimal value: 128

B8 to B15	Not used

As an example, to set bit B0 of the standard event status enable register, set `status.standard.enable = status.standard.OPC`.

In addition to the above constants, *`standardRegister`* can be set to the numeric equivalent of the bit to set. To set more than one bit of the register, set *`standardRegister`* to the sum of their decimal weights. For example, to set bits B0 and B4, set *`standardRegister`* to 17 (which is the sum of 1 + 16).

Bit	B7	B6	B5	B4	B3	B2	B1	B0
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Example 1

```
standardRegister = status.standard.OPC + status.standard.EXE
status.standard.enable = standardRegister
```

Uses constants to set the OPC and EXE bits of the standard event status enable register.

Example 2

```
-- decimal 17 = binary 0001 0001
standardRegister = 17
status.standard.enable = standardRegister
```

Uses the decimal value to set the OPC and EXE bits of the standard event status enable register.

Also see

Event summary bit (ESB register) (on page 12-9)

status.system.*

These attributes manage the TSP-Link™ system summary register of the status model for nodes 1 through 14.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute	- -	- -	- -	- -
.condition (R)	Yes	Not applicable	Not saved	Not applicable
.enable (RW)	Yes	Status reset	Not saved	0
.event (R)	Yes	Status reset	Not saved	0
.ntr (RW)	Yes	Status reset	Not saved	0
.ptr (RW)	Yes	Status reset	Not saved	32,767 (All bits set)

Usage

```
enableRegister = status.system.condition
enableRegister = status.system.enable
enableRegister = status.system.event
enableRegister = status.system.ntr
enableRegister = status.system.ptr
status.system.enable = enableRegister
status.system.ntr = enableRegister
status.system.ptr = enableRegister
```

enableRegister	The status of the system summary register; a zero (0) indicates no bits set; other values indicate various bit settings

Details

In an expanded system (TSP-Link), these attributes are used to read or write to the system summary registers. They are set using a constant or a numeric value but are returned as a numeric value. The binary equivalent of the value indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B15. For example, if a value of `1.29000e+02` (which is 129) is read as the value of the condition register, the binary equivalent is 0000 0000 1000 0001. This value indicates that bit B0 and bit B7 are set.

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	>	>	>	>	>	>	>	>	*
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value	Decimal value
B0	`status.system.EXTENSION_BIT` `status.system.EXT`	1
B1	`status.system.NODE1`	2
B2	`status.system.NODE2`	4
B3	`status.system.NODE3`	8
B4	`status.system.NODE4`	16
B5	`status.system.NODE5`	32
B6	`status.system.NODE6`	64
B7	`status.system.NODE7`	128
B8	`status.system.NODE8`	256
B9	`status.system.NODE9`	512
B10	`status.system.NODE10`	1,024
B11	`status.system.NODE11`	2,048
B12	`status.system.NODE12`	4,096
B13	`status.system.NODE13`	8,192
B14	`status.system.NODE14`	16,384
B15	Not used	Not applicable

As an example, to set bit B0 of the system summary status enable register, set `status.system.enable = status.system.enable.EXT`.

In addition to the above constants, *`enableRegister`* can be set to the decimal value of the bit to set. To set more than one bit of the register, set *`enableRegister`* to the sum of their decimal values. For example, to set bits B11 and B14, set *`enableRegister`* to 18,432 (which is the sum of 2,048 + 16,384).

Bit	B7	B6	B5	B4	B3	B2	B1	B0
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Bit	B15	B14	B13	B12	B11	B10	B9	B8
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	32,768	16,384	8,192	4,096	2,048	1,024	512	256
Weights	(2^{15})	(2^{14})	(2^{13})	(2^{12})	(2^{11})	(2^{10})	(2^9)	(2^8)

Example 1

`enableRegister = status.system.NODE11 + status.system.NODE14` `status.system.enable = enableRegister`	
Uses constants to set bits B11 and B14 of the system summary enable register.	

Example 2

`-- decimal 18432 = binary 0100 1000 0000 0000`
`enableRegister = 18432`
`status.system.enable = enableRegister`	

Uses the decimal value to set bits B11 and B14 of the system summary enable register.	

Also see

status.system2.* (on page 7-197)
System summary bit (System register) (on page 12-5)

status.system2.*

These attributes manage the TSP-Link™ system summary register of the status model for nodes 15 through 28.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute	- -	- -	- -	- -
.condition (R)	Yes	Not applicable	Not saved	Not applicable
.enable (RW)	Yes	Status reset	Not saved	0
.event (R)	Yes	Status reset	Not saved	0
.ntr (RW)	Yes	Status reset	Not saved	0
.ptr (RW)	Yes	Status reset	Not saved	32,767 (All bits set)

Usage

```
enableRegister = status.system2.condition
enableRegister = status.system2.enable
enableRegister = status.system2.event
enableRegister = status.system2.ntr
enableRegister = status.system2.ptr
status.system2.enable = enableRegister
status.system2.ntr = enableRegister
status.system2.ptr = enableRegister
```

enableRegister	The status of the system summary 2 register; a zero (0) indicates no bits set; other values indicate various bit settings

Details

In an expanded system (TSP-Link), these attributes are used to read or write to the system summary registers. They are set using a constant or a numeric value but are returned as a numeric value. The binary equivalent of the value indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B15. For example, if a value of `1.29000e+02` (which is 129) is read as the value of the condition register, the binary equivalent is 0000 0000 1000 0001. This value indicates that bit B0 and bit B7 are set.

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	>	>	>	>	>	>	>	>	*
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value	Decimal value
B0	`status.system2.EXTENSION_BIT` `status.system2.EXT`	1
B1	`status.system2.NODE15`	2
B2	`status.system2.NODE16`	4
B3	`status.system2.NODE17`	8
B4	`status.system2.NODE18`	16
B5	`status.system2.NODE19`	32
B6	`status.system2.NODE20`	64
B7	`status.system2.NODE21`	128
B8	`status.system2.NODE22`	256
B9	`status.system2.NODE23`	512
B10	`status.system2.NODE24`	1,024
B11	`status.system2.NODE25`	2,048
B12	`status.system2.NODE26`	4,096
B13	`status.system2.NODE27`	8,192
B14	`status.system2.NODE28`	16,384
B15	Not used	Not applicable

As an example, to set bit B0 of the system summary 2 enable register, set `status.system2.enable = status.system2.EXT`.

In addition to the above constants, *`enableRegister`* can be set to the decimal value of the bit to set. To set more than one bit of the register, set *`enableRegister`* to the sum of their decimal values. For example, to set bits B11 and B14, set *`enableRegister`* to 18,432 (which is the sum of 2,048 + 16,384).

Bit	**B7**	**B6**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Bit	B15	B14	B13	B12	B11	B10	B9	B8
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	32,768	16,384	8,192	4,096	2,048	1,024	512	256
Weights	(2^{15})	(2^{14})	(2^{13})	(2^{12})	(2^{11})	(2^{10})	(2^{9})	(2^{8})

Example 1

```
enableRegister = status.system2.NODE25 + status.system2.NODE28
status.system2.enable = enableRegister
```

Uses constants to set bits B11 and B14 of the system summary 2 enable register.

Example 2

```
-- decimal 18432 = binary 0100 1000 0000 0000
enableRegister = 18432
status.system2.enable = enableRegister
```

Uses the decimal value to set bits B11 and B14 of the system summary 2 enable register.

Also see

status.system.* (on page 7-195)
status.system3.* (on page 7-199)
System summary bit (System register) (on page 12-5)

status.system3.*

These attributes manage the TSP-Link™ system summary register of the status model for nodes 29 through 42.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute	- -	- -	- -	- -
.condition (R)	Yes	Not applicable	Not saved	Not applicable
.enable (RW)	Yes	Status reset	Not saved	0
.event (R)	Yes	Status reset	Not saved	0
.ntr (RW)	Yes	Status reset	Not saved	0
.ptr (RW)	Yes	Status reset	Not saved	32,767 (All bits set)

Usage

```
enableRegister = status.system3.condition
enableRegister = status.system3.enable
enableRegister = status.system3.event
enableRegister = status.system3.ntr
enableRegister = status.system3.ptr
status.system3.enable = enableRegister
status.system3.ntr = enableRegister
status.system3.ptr = enableRegister
```

enableRegister	The status of the system summary 3 register; a zero (0) indicates no bits set; other values indicate various bit settings

Details

In an expanded system (TSP-Link), these attributes are used to read or write to the system summary registers. They are set using a constant or a numeric value but are returned as a numeric value. The binary equivalent of the value indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0 and the most significant bit is bit B15. For example, if a value of `1.29000e+02` (which is 129) is read as the value of the condition register, the binary equivalent is 0000 0000 1000 0001. This value indicates that bit B0 and bit B7 are set.

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	>	>	>	>	>	>	>	>	*
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value	Decimal value
B0	`status.system3.EXTENSION_BIT` `status.system3.EXT`	1
B1	`status.system3.NODE29`	2
B2	`status.system3.NODE30`	4
B3	`status.system3.NODE31`	8
B4	`status.system3.NODE32`	16
B5	`status.system3.NODE33`	32
B6	`status.system3.NODE34`	64
B7	`status.system3.NODE35`	128
B8	`status.system3.NODE36`	256
B9	`status.system3.NODE37`	512
B10	`status.system3.NODE38`	1,024
B11	`status.system3.NODE39`	2,048
B12	`status.system3.NODE40`	4,096
B13	`status.system3.NODE41`	8,192
B14	`status.system3.NODE42`	16,384
B15	Not used	Not applicable

As an example, to set bit B0 of the system summary 3 enable register, set `status.system3.enable = status.system3.EXT`.

In addition to the above constants, `enableRegister` can be set to the decimal value of the bit to set. To set more than one bit of the register, set `enableRegister` to the sum of their decimal values. For example, to set bits B11 and B14, set `enableRegister` to 18,432 (which is the sum of 2,048 + 16,384).

Bit	**B7**	**B6**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Bit	**B15**	**B14**	**B13**	**B12**	**B11**	**B10**	**B9**	**B8**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	32,768	16,384	8,192	4,096	2,048	1,024	512	256
Weights	(2^{15})	(2^{14})	(2^{13})	(2^{12})	(2^{11})	(2^{10})	(2^9)	(2^8)

Example 1

```
enableRegister = status.system3.NODE39 + status.system3.NODE42
status.system3.enable = enableRegister
```

Uses constants to set bits B11 and B14 of the system summary 3 enable register.

Example 2

```
-- decimal 18432 = binary 0100 1000 0000 0000
enableRegister = 18432
status.system3.enable = enableRegister
```

Uses the decimal value to set bits B11 and B14 of the system summary 3 enable register.

Also see

status.system2.* (on page 7-197)
status.system4.* (on page 7-201)
System summary bit (System register) (on page 12-5)

status.system4.*

These attributes manage the TSP-Link™ system summary register of the status model for nodes 43 through 56.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute	- -	- -	- -	- -
.condition (R)	Yes	Not applicable	Not saved	Not applicable
.enable (RW)	Yes	Status reset	Not saved	0
.event (R)	Yes	Status reset	Not saved	0
.ntr (RW)	Yes	Status reset	Not saved	0
.ptr (RW)	Yes	Status reset	Not saved	32,767 (All bits set)

Usage

```
enableRegister = status.system4.condition
enableRegister = status.system4.enable
enableRegister = status.system4.event
enableRegister = status.system4.ntr
enableRegister = status.system4.ptr
status.system4.enable = enableRegister
status.system4.ntr = enableRegister
status.system4.ptr = enableRegister
```

enableRegister	The status of the system summary 4 register; a zero (0) indicates no bits set; other values indicate various bit settings

Details

In an expanded system (TSP-Link), these attributes are used to read or write to the system summary registers. They are set using a constant or a numeric value but are returned as a numeric value. The binary equivalent of the value indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B15. For example, if a value of `1.29000e+02` (which is 129) is read as the value of the condition register, the binary equivalent is 0000 0000 1000 0001. This value indicates that bit B0 and bit B7 are set.

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	>	>	>	>	>	>	>	>	*
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value	Decimal value
B0	`status.system4.EXTENSION_BIT` `status.system4.EXT`	1
B1	`status.system4.NODE43`	2
B2	`status.system4.NODE44`	4
B3	`status.system4.NODE45`	8
B4	`status.system4.NODE46`	16
B5	`status.system4.NODE47`	32
B6	`status.system4.NODE48`	64
B7	`status.system4.NODE49`	128
B8	`status.system4.NODE50`	256
B9	`status.system4.NODE51`	512
B10	`status.system4.NODE52`	1,024
B11	`status.system4.NODE53`	2,048
B12	`status.system4.NODE54`	4,096
B13	`status.system4.NODE55`	8,192
B14	`status.system4.NODE56`	16,384
B15	Not used	Not applicable

As an example, to set bit B0 of the system summary 4 enable register, set `status.system4.enable = status.system4.enable.EXT`.

In addition to the above constants, *enableRegister* can be set to the decimal value of the bit to set. To set more than one bit of the register, set *enableRegister* to the sum of their decimal values. For example, to set bits B11 and B14, set *enableRegister* to 18,432 (which is the sum of 2,048 + 16,384).

Bit	**B7**	**B6**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Bit	**B15**	**B14**	**B13**	**B12**	**B11**	**B10**	**B9**	**B8**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	32,768	16,384	8,192	4,096	2,048	1,024	512	256
Weights	(2^{15})	(2^{14})	(2^{13})	(2^{12})	(2^{11})	(2^{10})	(2^9)	(2^8)

Example 1

```
enableRegister = status.system4.NODE53 + status.system4.NODE56
status.system2.enable = enableRegister
```

Uses constants to set bit B11 and bit B14 of the system summary 4 enable register.

Example 2

```
-- decimal 18432 = binary 0100 1000 0000 0000
enableRegister = 18432
status.system4.enable = enableRegister
```

Uses a decimal value to set bit B11 and bit B14 of the system summary 4 enable register.

Also see

status.system3.* (on page 7-199)
status.system5.* (on page 7-203)
System summary bit (System register) (on page 12-5)

status.system5.*

These attributes manage the TSP-Link™ system summary register of the status model for nodes 57 through 64.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute	- -	- -	- -	- -
.condition (R)	Yes	Not applicable	Not saved	Not applicable
.enable (RW)	Yes	Status reset	Not saved	0
.event (R)	Yes	Status reset	Not saved	0
.ntr (RW)	Yes	Status reset	Not saved	0
.ptr (RW)	Yes	Status reset	Not saved	510 (All bits set)

Usage

```
enableRegister = status.system5.condition
enableRegister = status.system5.enable
enableRegister = status.system5.event
enableRegister = status.system5.ntr
enableRegister = status.system5.ptr
status.system5.enable = enableRegister
status.system5.ntr = enableRegister
status.system5.ptr = enableRegister
```

enableRegister	The status of the system summary 5 register; a zero (0) indicates no bits set; other values indicate various bit settings

Details

In an expanded system (TSP-Link), these attributes are used to read or write to the system summary registers. They are set using a constant or a numeric value, but are returned as a numeric value. The binary equivalent of the value indicates which register bits are set. In the binary equivalent, the least significant bit is bit B0, and the most significant bit is bit B15. For example, if a value of `1.30000e+02` (which is 130) is read as the value of the condition register, the binary equivalent is 0000 0000 1000 0010. This value indicates that bit B1 and bit B7 are set.

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
**	>	>	>	>	>	>	>	>	>	>	>	>	>	>	*
0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0

* Least significant bit
** Most significant bit

For information about .condition, .enable, .event, .ntr, and .ptr registers, refer to Status register set contents (on page 12-1) and Enable and transition registers (on page 12-20). The individual bits of this register are defined in the following table.

Bit	Value	Decimal value
B0	Not used	Not applicable
B1	`status.system5.NODE57`	2
B2	`status.system5.NODE58`	4
B3	`status.system5.NODE59`	8
B4	`status.system5.NODE60`	16
B5	`status.system5.NODE61`	32
B6	`status.system5.NODE62`	64
B7	`status.system5.NODE63`	128
B8	`status.system5.NODE64`	256
B9 to B15	Not used	Not applicable

As an example, to set bit B1 of the system summary 5 enable register, set `status.system5.enable = status.system5.NODE57`.

In addition to the above constants, *enableRegister* can be set to the numeric equivalent of the bit to set. To set more than one bit of the register, set *enableRegister* to the sum of their decimal weights. For example, to set bits B1 and B4, set *enableRegister* to 18 (which is the sum of 2 + 16).

Bit	**B7**	**B6**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Bit	**B15**	**B14**	**B13**	**B12**	**B11**	**B10**	**B9**	**B8**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	32,768	16,384	8,192	4,096	2,048	1,024	512	256
Weights	(2^{15})	(2^{14})	(2^{13})	(2^{12})	(2^{11})	(2^{10})	(2^9)	(2^8)

Example 1

```
enableRegister = status.system5.NODE57 + status.system5.NODE60
status.system2.enable = enableRegister
```

Uses constants to set bits B1 and B4 of the system summary 5 enable register.

Example 2

```
-- decimal 18 = binary 0000 0000 0001 0010
enableRegister = 18
status.system5.enable = enableRegister
```

Uses the decimal value to set bits B1 and B4 of the system summary 5 enable register.

Also see

status.system4.* (on page 7-201)
System summary bit (System register) (on page 12-5)

timer.measure.t()

This function measures the elapsed time since the timer was last reset.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
time = timer.measure.t()
```

time	The elapsed time in seconds (1 µs resolution)

Example 1

```
timer.reset()
-- (intervening code)
time = timer.measure.t()
print(time)
```

This example resets the timer and measures the time since the reset.
Output:
`1.469077e+01`
The output varies. The above output indicates that `timer.measure.t()` was executed 14.69077 seconds after `timer.reset()`.

Example 2

```
beeper.enable = beeper.ON
beeper.beep(0.5, 2400)
print("reset timer")
timer.reset()
delay(0.5)
dt = timer.measure.t()
print("timer after delay:", dt)
beeper.beep(0.5, 2400)
```

Enable the beeper.
Emit a beep and set the beeper.
Reset the timer.
Set a delay.
Verify the duration of the delay before emitting another beep.
Output:
```
reset timer
timer after delay: 5.00e-01
```

Also see

timer.reset() (on page 7-207)

timer.reset()

This function resets the timer to zero (0) seconds.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
timer.reset()
```

Example

```
timer.reset()
-- (intervening code)
time = timer.measure.t()
print(time)
```

Resets the timer and then measures the time since the reset.
Output:
`1.469077e+01`
The above output indicates that `timer.measure.t()` was executed 14.69077 seconds after `timer.reset()`.

Also see

timer.measure.t() (on page 7-206)

trigger.blender[N].clear()

This function clears the blender event detector and resets the overrun indicator of blender *N*.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
trigger.blender[N].clear()
```

N	The blender number: 1 or 2

Details

This command sets the blender event detector to the undetected state and resets the overrun indicator of the event detector.

Example

```
trigger.blender[2].clear()
```

Clears the event detector for blender 2.

Also see

None

trigger.blender[N].EVENT_ID

This constant contains the trigger blender event number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Constant	Yes			

Usage

eventID = trigger.blender[*N*].EVENT_ID

eventID	Trigger event number
N	The blender number: 1 or 2

Details

Set the stimulus of any trigger object to the value of this constant to have the trigger object respond to trigger events from this trigger blender.

Example

```
digio.trigger[1].stimulus = trigger.blender[2].EVENT_ID
```

Set the trigger stimulus of digital I/O trigger 1 to be controlled by the trigger blender 2 event.

Also see

None

trigger.blender[N].orenable

This attribute selects whether the blender performs OR operations or AND operations.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Trigger blender N reset Recall setup	Create configuration script	false (AND mode)

Usage

orenable = trigger.blender[*N*].orenable
trigger.blender[*N*].orenable = *orenable*

orenable	The type of operation: ■ OR operation: `true` ■ AND operation: `false`
N	The blender number: 1 or 2

Details

This command selects whether the blender waits for any one event (`OR`) or waits for all selected events (`AND`) before signaling an output event.

Example

```
trigger.blender[1].orenable = true
trigger.blender[1].stimulus[1] = digio.trigger[3].EVENT_ID
trigger.blender[1].stimulus[2] = digio.trigger[5].EVENT_ID
```

Generate a trigger blender 1 event when a digital I/O trigger happens on line 3 or 5.

Also see

trigger.blender[N].reset() (on page 7-210)

trigger.blender[N].overrun

This attribute indicates whether or not an event was ignored because of the event detector state.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Instrument reset Trigger blender *N* clear Trigger blender *N* reset	Not applicable	Not applicable

Usage

```
overrun = trigger.blender[N].overrun
```

overrun	Trigger blender overrun state: `true` or `false`
N	The blender number: 1 or 2

Details

Indicates if an event was ignored because the event detector was already in the detected state when the event occurred. This is an indication of the state of the event detector that is built into the event blender itself.

This command does not indicate if an overrun occurred in any other part of the trigger model or in any other trigger object that is monitoring the event. It also is not an indication of an action overrun.

Example

```
print(trigger.blender[1].overrun)
```

If an event was ignored, the output is `true`.
If an event was not ignored, the output is `false`.

Also see

trigger.blender[N].reset() (on page 7-210)

trigger.blender[N].reset()

This function resets some of the trigger blender settings to their factory defaults.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
trigger.blender[N].reset()
```

N	The trigger event blender: 1 or 2

Details

The `trigger.blender[N].reset()` function resets the following attributes to their factory defaults:

- `trigger.blender[N].orenable`
- `trigger.blender[N].stimulus[M]`

It also clears `trigger.blender[N].overrun`.

Example

`trigger.blender[1].reset()`
Resets the trigger blender 1 settings to factory defaults.

Also see

trigger.blender[N].orenable (on page 7-208)
trigger.blender[N].overrun (on page 7-209)
trigger.blender[N].stimulus[M] (on page 7-210)

trigger.blender[N].stimulus[M]

This attribute specifies the events that trigger the blender.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Reset Recall setup Trigger blender N reset	Create configuration script	0

Usage

```
eventID = trigger.blender[N].stimulus[M]
trigger.blender[N].stimulus[M] = eventID
```

eventID	The event that triggers the blender action; see **Details**
N	An integer representing the trigger event blender: 1 or 2
M	An integer representing the stimulus index: 1 to 4

Details

There are four acceptors that can each select a different event. The *eventID* parameter can be the event ID of any trigger event. The *eventID* parameter may be one of the existing trigger event IDs shown in the following table.

Trigger event IDs

Trigger event ID	Description
`digio.trigger[N].EVENT_ID`	An edge (rising, falling, or either based on the configuration of the line) on the digital input line
`display.trigger.EVENT_ID`	A press of the trigger key on the front panel
`trigger.EVENT_ID`	A `*trg` message on the active command interface; if GPIB is the active command interface, a `GET` message also generates this event
`trigger.blender[N].EVENT_ID`	A combination of events has occurred
`trigger.timer[N].EVENT_ID`	A delay expired
`tsplink.trigger[N].EVENT_ID`	An edge (rising, falling, or either based on the configuration of the line) on the TSP-Link^{TM} trigger line
`lan.trigger[N].EVENT_ID`	A LAN trigger event occurred
`scan.trigger.EVENT_SCAN_READY`	Scan ready event
`scan.trigger.EVENT_SCAN_START`	Scan start event
`scan.trigger.EVENT_CHANNEL_READY`	Channel ready event
`scan.trigger.EVENT_SCAN_COMP`	Scan complete event
`scan.trigger.EVENT_IDLE`	Idle event

Example

```
digio.trigger[3].mode = digio.TRIG_FALLING
digio.trigger[5].mode = digio.TRIG_FALLING
trigger.blender[1].orenable = true
trigger.blender[1].stimulus[1] = digio.trigger[3].EVENT_ID
trigger.blender[1].stimulus[2] = digio.trigger[5].EVENT_ID
```

Generate a trigger blender 1 event when a digital I/O trigger happens on line 3 or 5.

Also see

trigger.blender[N].reset() (on page 7-210)

trigger.blender[N].wait()

This function waits for a blender trigger event to occur.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`triggered = trigger.blender[N].wait(timeout)`

triggered	Trigger detection indication for blender
N	The trigger blender on which to wait: 1 or 2
timeout	Maximum amount of time in seconds to wait for the trigger blender event

Details

This function waits for an event blender trigger event. If one or more trigger events were detected since the last time `trigger.blender[N].wait()` or `trigger.blender[N].clear()` was called, this function returns immediately.

After detecting a trigger with this function, the event detector automatically resets and rearms. This is true regardless of the number of events detected.

Example

```
digio.trigger[3].mode = digio.TRIG_FALLING
digio.trigger[5].mode = digio.TRIG_FALLING
trigger.blender[1].orenable = true
trigger.blender[1].stimulus[1] = digio.trigger[3].EVENT_ID
trigger.blender[1].stimulus[2] = digio.trigger[5].EVENT_ID
print(trigger.blender[1].wait(3))
```

Generate a trigger blender 1 event when a digital I/O trigger happens either on line 3 or 5.
Wait three seconds while checking if trigger blender 1 event has occurred.
If the blender trigger event has happened, then `true` is output. If the trigger event has not happened, then `false` is output after the timeout expires.

Also see

trigger.blender[N].clear() (on page 7-207)

trigger.clear()

This function clears the command interface trigger event detector.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
trigger.clear()
```

Details

The trigger event detector indicates if a trigger event has been detected since the last `trigger.wait()` call. `trigger.clear()` clears the trigger event detector and discards the history of command interface trigger events.

Also see

trigger.wait() (on page 7-222)

trigger.EVENT_ID

This constant contains the command interface trigger event number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Constant	Yes			

Usage

```
eventID = trigger.EVENT_ID
```

eventID	The event ID for the command interface triggers

Details

You can set the stimulus of any trigger object to the value of this constant to have the trigger object respond to command interface trigger events.

Also see

None

trigger.timer[N].clear()

This function clears the timer event detector and overrun indicator for the specified trigger timer number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
trigger.timer[N].clear()
```

N	Trigger timer number: 1 to 4

Details

This command sets the timer event detector to the undetected state and resets the overrun indicator.

Example

`trigger.timer[1].clear()`
Clears trigger timer 1.

Also see

trigger.timer[N].count (on page 7-214)

trigger.timer[N].count

This attribute sets the number of events to generate each time the timer generates a trigger event.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup Trigger timer *N* reset	Create configuration script	1

Usage

```
count = trigger.timer[N].count
trigger.timer[N].count = count
```

count	Number of times to repeat the trigger: 0 to 1,048,575
N	Trigger timer number: 1 to 4

Details

If the count is set to a number greater than 1, the timer automatically starts the next trigger timer delay at the expiration of the previous delay.

Set the count to zero (0) to cause the timer to generate trigger events indefinitely.

If you use the trigger timer with a trigger model, make sure the count value is the same or more than any count values expected in the trigger model.

Example

`print(trigger.timer[1].count)`	
Read trigger count for timer number 1.	

Also see

trigger.timer[N].clear() (on page 7-214)
trigger.timer[N].delay (on page 7-215)
trigger.timer[N].reset() (on page 7-219)

trigger.timer[N].delay

This attribute sets and reads the timer delay.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup Trigger timer N reset	Create configuration script	10e-6 (10 µs)

Usage

```
interval = trigger.timer[N].delay
trigger.timer[N].delay = interval
```

interval	Delay interval in seconds: 8 µs to 100 ks
N	Trigger timer number: 1 to 4

Details

Once the timer is enabled, each time the timer is triggered, it uses this delay period.

Assigning a value to this attribute is equivalent to:

```
trigger.timer[N].delaylist = {interval}
```

This creates a delay list of one value.

Reading this attribute returns the delay interval that is used the next time the timer is triggered.

Example

`trigger.timer[1].delay = 50e-6`	
Set the trigger timer 1 to delay for 50 µs.	

Also see

trigger.timer[N].reset() (on page 7-219)

trigger.timer[N].delaylist

This attribute sets an array of timer intervals.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup Trigger timer *N* reset	Create configuration script	10e-6 (10 µs)

Usage

```
intervals = trigger.timer[N].delaylist
trigger.timer[N].delaylist = intervals
```

intervals	Table of delay intervals in seconds
N	Trigger timer number: 1 to 4

Details

Each time the timer is triggered after it is enabled, it uses the next delay period from the array. The default value is an array with one value of 10 µs.

After all elements in the array have been used, the delays restart at the beginning of the list.

If the array contains more than one element, the average of the delay intervals in the list must be ≥ 50 µs.

Example

```
trigger.timer[3].delaylist = {50e-6, 100e-6, 150e-6}
DelayList = trigger.timer[3].delaylist
for x = 1, table.getn(DelayList) do
   print(DelayList[x])
end
```

Set a delay list on trigger timer 3 with three delays (50 µs, 100 µs, and 150 µs).
Read the delay list on trigger timer 3.
Output:

```
5e-05
0.0001
0.00015
```

Also see

trigger.timer[N].reset() (on page 7-219)

trigger.timer[N].EVENT_ID

This constant specifies the trigger timer event number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Constant	Yes			

Usage

eventID = trigger.timer[*N*].EVENT_ID

eventID	The trigger event number
N	Trigger timer number: 1 to 4

Details

This constant is an identification number that identifies events generated by this timer.

Set the stimulus of any trigger object to the value of this constant to have the trigger object respond to events from this timer.

Also see

None

trigger.timer[N].overrun

This attribute indicates if an event was ignored because of the event detector state.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Instrument reset Recall setup Trigger timer N clear Trigger timer N reset	Not applicable	false

Usage

overrun = trigger.timer[*N*].overrun

overrun	Trigger overrun state: `true` or `false`
N	Trigger timer number: 1 to 4

Details

This command indicates if an event was ignored because the event detector was already in the detected state when the event occurred.

This is an indication of the state of the event detector built into the timer itself. It does not indicate if an overrun occurred in any other part of the trigger model or in any other construct that is monitoring the delay completion event. It also is not an indication of a delay overrun.

Example

`print(trigger.timer[1].overrun)`
If an event was ignored, the output is `true`. If the event was not ignored, the output is `false`.

Also see

trigger.timer[N].reset() (on page 7-219)

trigger.timer[N].passthrough

This attribute enables or disables the timer trigger pass-through mode.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup Trigger timer N reset	Create configuration script	false (disabled)

Usage

```
passthrough = trigger.timer[N].passthrough
trigger.timer[N].passthrough = passthrough
```

passthrough	The state of pass-through mode; set to one of the following values: ■ Enabled: `true` ■ Disabled: `false`
N	Trigger timer number: 1 to 4

Details

When pass-through mode is enabled, triggers are passed through immediately and initiate the delay. When disabled, a trigger only initiates a delay.

Example

`trigger.timer[1].passthrough = true`
Enables pass-through mode on trigger timer 1.

Also see

trigger.timer[N].reset() (on page 7-219)

trigger.timer[N].reset()

This function resets some of the trigger timer settings to their factory defaults.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
trigger.timer[N].reset()
```

N	Trigger timer number: 1 to 4

Details

The `trigger.timer[N].reset()` function resets the following attributes to their factory defaults:

- `trigger.timer[N].count`
- `trigger.timer[N].delay`
- `trigger.timer[N].delaylist`
- `trigger.timer[N].passthrough`
- `trigger.timer[N].stimulus`

It also clears `trigger.timer[N].overrun`.

Example

`trigger.timer[1].reset()`
Resets the attributes associated with timer 1 to factory default values.

Also see

trigger.timer[N].count (on page 7-214)
trigger.timer[N].delay (on page 7-215)
trigger.timer[N].delaylist (on page 7-216)
trigger.timer[N].overrun (on page 7-217)
trigger.timer[N].passthrough (on page 7-218)
trigger.timer[N].stimulus (on page 7-220)

trigger.timer[N].stimulus

This attribute specifies which event starts the timer.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Reset Recall setup Trigger timer N reset	Create configuration script	0

Usage

```
eventID = trigger.timer[N].stimulus
trigger.timer[N].stimulus = eventID
```

eventID	The event that triggers the timer delay; see **Details**
N	Trigger timer number: 1 to 4

Details

The *eventID* parameter may be one of the trigger event IDs shown in the following table.

Trigger event IDs	
Trigger event ID	**Description**
`digio.trigger[N].EVENT_ID`	An edge (rising, falling, or either based on the configuration of the line) on the digital input line
`display.trigger.EVENT_ID`	A press of the trigger key on the front panel
`trigger.EVENT_ID`	A `*trg` message on the active command interface; if GPIB is the active command interface, a `GET` message also generates this event
`trigger.blender[N].EVENT_ID`	A combination of events has occurred
`trigger.timer[N].EVENT_ID`	A delay expired
`tsplink.trigger[N].EVENT_ID`	An edge (rising, falling, or either based on the configuration of the line) on the TSP-Link™ trigger line
`lan.trigger[N].EVENT_ID`	A LAN trigger event occurred
`scan.trigger.EVENT_SCAN_READY`	Scan ready event
`scan.trigger.EVENT_SCAN_START`	Scan start event
`scan.trigger.EVENT_CHANNEL_READY`	Channel ready event
`scan.trigger.EVENT_SCAN_COMP`	Scan complete event
`scan.trigger.EVENT_IDLE`	Idle event

Set this attribute to the *eventID* of any trigger event to cause the timer to start when that event occurs.

Use zero (0) to disable event processing.

Example

`print(trigger.timer[1].stimulus)`
Prints the event that will start a trigger 1 timer action.

Also see

trigger.timer[N].reset() (on page 7-219)

trigger.timer[N].wait()

This function waits for a trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`triggered = trigger.timer[N].wait(timeout)`

triggered	Trigger detection indication
N	Trigger timer number: 1 to 4
timeout	Maximum amount of time in seconds to wait for the trigger

Details

If one or more trigger events were detected since the last time `trigger.timer[N].wait()` or `trigger.timer[N].clear()` was called, this function returns immediately.

After waiting for a trigger with this function, the event detector is automatically reset and rearmed. This is true regardless of the number of events detected.

Example

```
triggered = trigger.timer[3].wait(10)
print(triggered)
```

Waits up to 10 s for a trigger on timer 3.
If `false` is returned, no trigger was detected during the 10 s timeout.
If `true` is returned, a trigger was detected.

Also see

trigger.timer[N].clear() (on page 7-214)

trigger.wait()

This function waits for a command interface trigger event.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

triggered = trigger.wait(*timeout*)

triggered	A trigger was detected during the timeout period: `true` No triggers were detected during the timeout period: `false`
timeout	Maximum amount of time in seconds to wait for the trigger

Details

This function waits up to *timeout* seconds for a trigger on the active command interface. A command interface trigger occurs when:

- A GPIB GET command is detected (GPIB only)
- A VXI-11 device_trigger method is invoked (VXI-11 only)
- A `*TRG` message is received

If one or more of these trigger events were previously detected, this function returns immediately.

After waiting for a trigger with this function, the event detector is automatically reset and rearmed. This is true regardless of the number of events detected.

Example

```
triggered = trigger.wait(10)
print(triggered)
```

Waits up to 10 seconds for a trigger.
If `false` is returned, no trigger was detected during the 10-second timeout.
If `true` is returned, a trigger was detected.

Also see

trigger.clear() (on page 7-213)

tsplink.group

This attribute contains the group number of a TSP-Link node.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Power cycle	Not applicable	0

Usage

```
groupNumber = tsplink.group
tsplink.group = groupNumber
```

groupNumber	The group number of the TSP-Link node: 0 to 64

Details

To remove the node from all groups, set the attribute value to 0.

When the node is turned off, the group number for that node changes to 0.

The master node can be assigned to any group. You can also include other nodes in the group that includes the master. Any nodes that are set to 0 are automatically included in the group that contains the master node, regardless of the group that is assigned to the master node.

Example

```
tsplink.group = 3
```

Assign the instrument to TSP-Link group number 3.

Also see

Using groups to manage nodes on TSP-Link network (on page 6-52)

tsplink.master

This attribute reads the node number assigned to the master node.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

```
masterNodeNumber = tsplink.master
```

masterNodeNumber	The node number of the master node

Details

After doing a TSP-Link reset (`tsplink.reset()`), use this attribute to access the node number of the master in a set of instruments connected over TSP-Link.

Example

```
LinkMaster = tsplink.master
```

Store the TSP-Link master node number in a variable called `LinkMaster`.

Also see

tsplink.reset() (on page 7-226)

tsplink.node

This attribute defines the node number.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Not applicable	Nonvolatile memory	2

Usage

```
nodeNumber = tsplink.node
tsplink.node = nodeNumber
```

nodeNumber	The node number of the instrument or enclosure: 1 to 64

Details

This command sets the TSP-Link node number and saves the value in nonvolatile memory.

Changes to the node number do not take effect until `tsplink.reset()` from an earlier TSP-Link instrument is executed on any node in the system.

Each node connected to the TSP-Link system must be assigned a different node number.

Example

`tsplink.node = 3`
Sets the TSP-Link node for this instrument to number 3.

Also see

tsplink.reset() (on page 7-226)
tsplink.state (on page 7-227)

tsplink.readbit()

This function reads the state of a TSP-Link synchronization line.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
data = tsplink.readbit(N)
```

data	The state of the synchronization line ▪ Low: 0 ▪ High: 1
N	The trigger line: 1 to 3

Example

`data = tsplink.readbit(3)` `print(data)`	
Assume line 3 is set high and is then read. Output: `1.000000e+00`	

Also see

tsplink.readport() (on page 7-225)
tsplink.writebit() (on page 7-236)

tsplink.readport()

This function reads the TSP-Link trigger lines as a digital I/O port.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
data = tsplink.readport()
```

data	Numeric value that indicates which lines are set

Details

The binary equivalent of the returned value indicates the input pattern on the I/O port. The least significant bit of the binary number corresponds to line 1 and the value of bit 3 corresponds to line 3. For example, a returned value of 2 has a binary equivalent of 010. This indicates that line 2 is high (1), and that the other two lines are low (0).

Example

`data = tsplink.readport()` `print(data)`
Reads state of all three TSP-Link lines. Assuming line 2 is set high, the output is: `2.000000e+00` (binary 010) The format of the output may vary depending on the ASCII precision setting.

Also see

tsplink.readbit() (on page 7-224)
tsplink.writebit() (on page 7-236)

tsplink.reset()

This function initializes (resets) all nodes (instruments) in the TSP-Link system.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
nodesFound = tsplink.reset()
nodesFound = tsplink.reset(expectedNodes)
```

nodesFound	The number of nodes actually found on the system
expectedNodes	The number of nodes expected on the system: 1 to 64

Details

This function erases all information regarding other nodes connected on the TSP-Link system and regenerates the system configuration. This function must be called at least once before any remote nodes can be accessed. If the node number for any instrument is changed, the TSP-Link must be reset again.

If *expectedNodes* is not given, this function generates an error if no other nodes are found on the TSP-Link network.

If *nodesFound* is less than *expectedNodes*, an error is generated. Note that the node on which the command is running is counted as a node. For example, giving an expected node count of 1 does not generate any errors, even if there are no other nodes on the TSP-Link network.

Also returns the number of nodes found.

Example

```
nodesFound = tsplink.reset(2)
print("Nodes found = " .. nodesFound)
```

Perform a TSP-Link reset and indicate how many nodes are found.
Sample output if two nodes are found:
`Nodes found = 2`
Sample output if fewer nodes are found and if `localnode.showerrors = 1`:
```
1219, TSP-Link found fewer nodes than expected
Nodes found = 1
```

Also see

localnode.showerrors
tsplink.node (on page 7-224)
tsplink.state (on page 7-227)

tsplink.state

This attribute describes the TSP-Link online state.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Not applicable	Not applicable	Not applicable

Usage

state = tsplink.state

state	TSP-Link state: `online` or `offline`

Details

When the instrument power is first turned on, the state is `offline`. After `tsplink.reset()` is successful, the state is `online`.

Example

```
state = tsplink.state
print(state)
```

Read the state of the TSP-Link system. If it is online, the output is:
`online`

Also see

tsplink.node (on page 7-224)
tsplink.reset() (on page 7-226)

tsplink.trigger[N].assert()

This function simulates the occurrence of the trigger and generates the corresponding event ID.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

tsplink.trigger[*N*].assert()

N	The trigger line: 1 to 3

Details

The set pulse width determines how long the trigger is asserted.

Example

```
tsplink.trigger[2].assert()
```

Asserts trigger on trigger line 2.

Also see

tsplink.trigger[N].clear() (on page 7-228)
tsplink.trigger[N].mode (on page 7-229)
tsplink.trigger[N].overrun (on page 7-231)
tsplink.trigger[N].pulsewidth (on page 7-232)
tsplink.trigger[N].release() (on page 7-232)
tsplink.trigger[N].stimulus (on page 7-234)
tsplink.trigger[N].wait() (on page 7-235)

tsplink.trigger[N].clear()

This function clears the event detector for a LAN trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
tsplink.trigger[N].clear()
```

N	The trigger line: 1 to 3

Details

The trigger event detector enters the detected state when an event is detected. `tsplink.trigger[N].clear()` clears a trigger event detector, discards the history of the trigger line, and clears the `tsplink.trigger[N].overrun` attribute.

Example

`tsplink.trigger[2].clear()`
Clears trigger event on synchronization line 2.

Also see

tsplink.trigger[N].mode (on page 7-229)
tsplink.trigger[N].overrun (on page 7-231)
tsplink.trigger[N].release() (on page 7-232)
tsplink.trigger[N].stimulus (on page 7-234)
tsplink.trigger[N].wait() (on page 7-235)

tsplink.trigger[N].EVENT_ID

This constant identifies the number that is used for the trigger events.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Constant	Yes			

Usage

```
eventID = tsplink.trigger[N].EVENT_ID
```

eventID	The trigger event number
N	The trigger line: 1 to 3

Details

This number is used by the TSP-Link trigger line when it detects an input trigger.

Set the stimulus of any trigger object to the value of this constant to have the trigger object respond to trigger events from this line.

Example

```
trigger.timer[1].stimulus = tsplink.trigger[2].EVENT_ID
```

Sets the trigger stimulus of trigger timer 1 to the TSP-Link trigger 2 event.

Also see

None

tsplink.trigger[N].mode

This attribute defines the trigger operation and detection mode.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup TSP-Link trigger N reset	Create configuration script	0 (tsplink.TRIG_BYPASS)

Usage

```
mode = tsplink.trigger[N].mode
tsplink.trigger[N].mode = mode
```

mode	The trigger mode
N	The trigger line: 1 to 3

Details

This attribute controls the mode in which the trigger event detector and the output trigger generator operate on the given trigger line.

The setting for the *mode* parameter can be one of the values shown in the following table.

Mode	Number value	Description
tsplink.TRIG_BYPASS	0	Allows direct control of the line as a digital I/O line.
tsplink.TRIG_FALLING	1	Detects falling-edge triggers as input. Asserts a TTL-low pulse for output.
tsplink.TRIG_RISING	2	If the programmed state of the line is high, the tsplink.TRIG_RISING mode behaves similarly to tsplink.TRIG_RISINGA. If the programmed state of the line is low, the tsplink.TRIG_RISING mode behaves similarly to tsplink.TRIG_RISINGM. Use tsplink.TRIG_RISINGA if the line is in the high output state. Use tsplink.TRIG_RISINGM if the line is in the low output state.
tsplink.TRIG_EITHER	3	Detects rising- or falling-edge triggers as input. Asserts a TTL-low pulse for output.
tsplink.TRIG_SYNCHRONOUSA	4	Detects the falling-edge input triggers and automatically latches and drives the trigger line low.
tsplink.TRIG_SYNCHRONOUS	5	Detects the falling-edge input triggers and automatically latches and drives the trigger line low. Asserts a TTL-low pulse as an output trigger.
tsplink.TRIG_SYNCHRONOUSM	6	Detects rising-edge triggers as an input. Asserts a TTL-low pulse for output.
tsplink.TRIG_RISINGA	7	Detects rising-edge triggers as input. Asserts a TTL-low pulse for output.
tsplink.TRIG_RISINGM	8	Edge detection as an input is not available. Generates a TTL-high pulse as an output trigger.

When programmed to any mode except `tsplink.TRIG_BYPASS`, the output state of the I/O line is controlled by the trigger logic and the user-specified output state of the line is ignored.

When the trigger mode is set to `tsplink.TRIG_RISING`, the user-specified output state of the line is examined. If the output state selected when the mode is changed is high, the actual mode that is used is `tsplink.TRIG_RISINGA`. If the output state selected when the mode is changed is low, the actual mode that is used is `tsplink.TRIG_RISINGM`.

The *mode* parameter stores the trigger mode as a numeric value when the attribute is read.

To control the line state, use the `tsplink.TRIG_BYPASS` mode with the `tsplink.writebit()` and the `tsplink.writeport()` commands.

Example

```
tsplink.trigger[3].mode = tsplink.TRIG_RISINGM
```

Sets the trigger mode for synchronization line 3 to `tsplink.TRIG_RISINGM`.

Also see

digio.writebit() (on page 7-65)
digio.writeport() (on page 7-66)
tsplink.trigger[N].assert() (on page 7-227)
tsplink.trigger[N].clear() (on page 7-228)
tsplink.trigger[N].overrun (on page 7-231)
tsplink.trigger[N].release() (on page 7-232)
tsplink.trigger[N].reset() (on page 7-233)
tsplink.trigger[N].stimulus (on page 7-234)
tsplink.trigger[N].wait() (on page 7-235)

tsplink.trigger[N].overrun

This attribute indicates if the event detector ignored an event while in the detected state.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (R)	Yes	Instrument reset Recall setup TSP-Link trigger N clear TSP-Link trigger N reset	Not applicable	Not applicable

Usage

```
overrun = tsplink.trigger[N].overrun
```

overrun	Trigger overrun state
N	The trigger line: 1 to 3

Details

This command indicates whether an event has been ignored because the event detector was already in the detected state when the event occurred.

This is an indication of the state of the event detector built into the synchronization line itself.

It does not indicate if an overrun occurred in any other part of the trigger model, or in any other construct that is monitoring the event. It also is not an indication of an output trigger overrun.

Example

```
print(tsplink.trigger[1].overrun)
```

If an event was ignored, displays `true`; if an event was not ignored, displays `false`.

Also see

tsplink.trigger[N].assert() (on page 7-227)
tsplink.trigger[N].clear() (on page 7-228)
tsplink.trigger[N].mode (on page 7-229)
tsplink.trigger[N].release() (on page 7-232)
tsplink.trigger[N].reset() (on page 7-233)
tsplink.trigger[N].stimulus (on page 7-234)
tsplink.trigger[N].wait() (on page 7-235)

tsplink.trigger[N].pulsewidth

This attribute sets the length of time that the trigger line is asserted for output triggers.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Reset TSP-Link trigger N reset	Create configuration script	10e-6 (10 µs)

Usage

```
width = tsplink.trigger[N].pulsewidth
tsplink.trigger[N].pulsewidth = width
```

width	The pulse width in seconds
N	The trigger line: 1 to 3

Details

Setting the pulse width to 0 (seconds) asserts the trigger indefinitely.

Example

`tsplink.trigger[3].pulsewidth = 20e-6`	
Sets pulse width for trigger line 3 to 20 µs.	

Also see

tsplink.trigger[N].release() (on page 7-232)

tsplink.trigger[N].release()

This function releases a latched trigger on the given TSP-Link trigger line.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
tsplink.trigger[N].release()
```

N	The trigger line: 1 to 3

Details

Releases a trigger that was asserted with an indefinite pulse width. It also releases a trigger that was latched in response to receiving a synchronous mode trigger.

Example

`tsplink.trigger[3].release()`	
Releases trigger line 3.	

Also see

tsplink.trigger[N].assert() (on page 7-227)
tsplink.trigger[N].clear() (on page 7-228)
tsplink.trigger[N].mode (on page 7-229)
tsplink.trigger[N].overrun (on page 7-231)
tsplink.trigger[N].pulsewidth (on page 7-232)
tsplink.trigger[N].stimulus (on page 7-234)
tsplink.trigger[N].wait() (on page 7-235)

tsplink.trigger[N].reset()

This function resets some of the TSP-Link trigger attributes to their factory defaults.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`tsplink.trigger[N].reset()`

N	The trigger line: 1 to 3

Details

The `tsplink.trigger[N].reset()` function resets the following attributes to their factory defaults:

- `tsplink.trigger[N].mode`
- `tsplink.trigger[N].stimulus`
- `tsplink.trigger[N].pulsewidth`

This also clears `tsplink.trigger[N].overrun`.

Example

`tsplink.trigger[3].reset()`
Resets TSP-Link trigger line 3 attributes to factory default values.

Also see

tsplink.trigger[N].mode (on page 7-229)
tsplink.trigger[N].overrun (on page 7-231)
tsplink.trigger[N].pulsewidth (on page 7-232)
tsplink.trigger[N].stimulus (on page 7-234)

tsplink.trigger[N].stimulus

This attribute specifies the event that causes the synchronization line to assert a trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Reset Recall setup TSP-Link trigger N reset	Create configuration script	0

Usage

```
eventID = tsplink.trigger[N].stimulus
tsplink.trigger[N].stimulus = eventID
```

eventID	The event identifier for the triggering event
N	The trigger line (1 to 3)

Details

To disable automatic trigger assertion on the synchronization line, set this attribute to zero (0).

Do not use this attribute when triggering under script control. Use `tsplink.trigger[N].assert()` instead.

The *eventID* parameter may be one of the existing trigger event IDs shown in the following table.

Trigger event IDs	
Trigger event ID	**Description**
`digio.trigger[N].EVENT_ID`	An edge (rising, falling, or either based on the configuration of the line) on the digital input line
`display.trigger.EVENT_ID`	A press of the trigger key on the front panel
`trigger.EVENT_ID`	A `*trg` message on the active command interface; if GPIB is the active command interface, a `GET` message also generates this event
`trigger.blender[N].EVENT_ID`	A combination of events has occurred
`trigger.timer[N].EVENT_ID`	A delay expired
`tsplink.trigger[N].EVENT_ID`	An edge (rising, falling, or either based on the configuration of the line) on the TSP-Link™ trigger line
`lan.trigger[N].EVENT_ID`	A LAN trigger event occurred
`scan.trigger.EVENT_SCAN_READY`	Scan ready event
`scan.trigger.EVENT_SCAN_START`	Scan start event
`scan.trigger.EVENT_CHANNEL_READY`	Channel ready event
`scan.trigger.EVENT_SCAN_COMP`	Scan complete event
`scan.trigger.EVENT_IDLE`	Idle event

Example

```
print(tsplink.trigger[3].stimulus)
```

Prints the event that will start TSP-Link trigger line 3 action.

Also see

tsplink.trigger[N].assert() (on page 7-227)
tsplink.trigger[N].reset() (on page 7-233)

tsplink.trigger[N].wait()

This function waits for a trigger.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
triggered = tsplink.trigger[N].wait(timeout)
```

triggered	Trigger detection indication; set to one of the following values: ■ A trigger is detected during the timeout period: `true` ■ A trigger is not detected during the timeout period: `false`
N	The trigger line: 1 to 3
timeout	The timeout value in seconds

Details

This function waits up to the timeout value for an input trigger. If one or more trigger events were detected since the last time `tsplink.trigger[N].wait()` or `tsplink.trigger[N].clear()` was called, this function returns immediately.

After waiting for a trigger with this function, the event detector is automatically reset and rearmed. This is true regardless of the number of events detected.

Example

```
triggered = tsplink.trigger[3].wait(10)
print(triggered)
```

Waits up to 10 seconds for a trigger on TSP-Link line 3.
If `false` is returned, no trigger was detected during the 10-second timeout.
If `true` is returned, a trigger was detected.

Also see

tsplink.trigger[N].clear() (on page 7-228)

tsplink.writebit()

This function sets a TSP-Link trigger line high or low.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`tsplink.writebit(N, data)`

N	The trigger line: 1 to 3
data	The value to write to the bit: ■ Low: `0` ■ High: `1`

Details

Use `tsplink.writebit()` and `tsplink.writeport()` to control the output state of the trigger line when trigger operation is set to `tsplink.TRIG_BYPASS`.

If the output line is write-protected by the `tsplink.writeprotect` attribute, this command is ignored.

The reset function does not affect the present states of the TSP-Link trigger lines.

Example

`tsplink.writebit(3, 0)`
Sets trigger line 3 low (0).

Also see

tsplink.readbit() (on page 7-224)
tsplink.readport() (on page 7-225)
tsplink.writeport() (on page 7-236)
tsplink.writeprotect (on page 7-237)

tsplink.writeport()

This function writes to all TSP-Link synchronization lines.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

`tsplink.writeport(data)`

data	Value to write to the port: 0 to 7

Details

The binary representation of `data` indicates the output pattern that is written to the I/O port. For example, a data value of 2 has a binary equivalent of 010. Line 2 is set high (1) and the other two lines are set low (0).

Write-protected lines are not changed.

Use the `tsplink.writebit()` and `tsplink.writeport()` commands to control the output state of the synchronization line when trigger operation is set to `tsplink.TRIG_BYPASS`.

The `reset()` function does not affect the present states of the trigger lines.

Example

`tsplink.writeport(3)`
Sets the synchronization lines 1 and 2 high (binary 011).

Also see

tsplink.readbit() (on page 7-224)
tsplink.writebit() (on page 7-236)
tsplink.writeprotect (on page 7-237)

tsplink.writeprotect

This attribute contains the write-protect mask that protects bits from changes by the `tsplink.writebit()` and `tsplink.writeport()` functions.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	Yes	Instrument reset Recall setup	Create configuration script	0

Usage

```
mask = tsplink.writeprotect
tsplink.writeprotect = mask
```

mask	An integer that specifies the value of the bit pattern for write-protect; set bits to `1` to write-protect the corresponding TSP-Link trigger line

Details

The binary equivalent of *mask* indicates the mask to be set for the TSP-Link trigger line. For example, a *mask* value of `5` has a binary equivalent of 101. This *mask* write-protects TSP-Link trigger lines 1 and 3.

Example

`tsplink.writeprotect = 5`
Write-protects TSP-Link trigger lines 1 and 3.

Also see

tsplink.readbit() (on page 7-224)
tsplink.readport() (on page 7-225)
tsplink.writebit() (on page 7-236)
tsplink.writeport() (on page 7-236)

tspnet.clear()

This function clears any pending output data from the instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`tspnet.clear(connectionID)`

connectionID	The connection ID returned from `tspnet.connect()`

Details

This function clears any pending output data from the device. No data is returned to the caller and no data is processed.

Example

```
tspnet.write(testdevice, "print([[hello]])")
print(tspnet.readavailable(testdevice))
tspnet.clear(testdevice)
print(tspnet.readavailable(testdevice))
```

Write data to a device, then print how much is available.
Output:
`6.0000000e+00`

Clear data and print how much data is available again.
Output:
`0.0000000e+00`

Also see

tspnet.connect() (on page 7-239)
tspnet.readavailable() (on page 7-244)
tspnet.write() (on page 7-250)

tspnet.connect()

This function establishes a network connection with another LAN instrument or device through the LAN interface.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
connectionID = tspnet.connect("ipAddress")
connectionID = tspnet.connect("ipAddress", portNumber, "initString")
```

connectionID	The connection ID to be used as a handle in all other `tspnet` function calls
ipAddress	A string that contains the IP address to which to connect; accepts IP address or host name when trying to connect
portNumber	Port number (default `5025`)
initString	Initialization string to send to *ipAddress*

Details

This command connects a device to another device through the LAN interface. If the *portNumber* is 23, the interface uses the telnet protocol and sets appropriate termination characters to communicate with the device.

If a *portNumber* and *initString* are provided, it is assumed that the remote device is not TSP-enabled. The Model 707B or 708B does not perform any extra processing, prompt handling, error handling, or sending of commands. In addition, the `tspnet.tsp.*` commands cannot be used on devices that are not TSP-enabled.

If neither a *portNumber* nor an *initString* is provided, the remote device is assumed to be a Keithley TSP-enabled device. Depending on the state of the `tspnet.tsp.abortonconnect` attribute, the Model 707B or 708B sends an `abort` command to the remote device on connection.

The Model 707B or 708B also enables TSP prompts on the remote device and error management. The Model 707B or 708B places remote errors from the TSP-enabled device in its own error queue and prefaces these errors with `Remote Error`, followed by an error description.

Do not manually change either the prompt functionality (`localnode.prompts`) or show errors by changing `localnode.showerrors` on the remote TSP-enabled device. If you do this, subsequent `tspnet.tsp.*` commands using the connection may fail.

You can simultaneously connect to a maximum of 32 remote devices.

Example 1

```
instrumentID = tspnet.connect("192.0.2.1")
if instrumentID then
   -- Use instrumentID as needed here
   tspnet.disconnect(instrumentID)
end
```

Connect to a TSP-enabled device.

Example 2

```
instrumentID = tspnet.connect("192.0.2.1", 1394, "*rst\r\n")
if instrumentID then
   -- Use instrumentID as needed here
   tspnet.disconnect(instrumentID)
end
```

Connect to a device that is not TSP-enabled.

Also see

localnode.prompts (on page 7-129)
localnode.showerrors (on page 7-133)
tspnet.tsp.abortonconnect (on page 7-247)
tspnet.disconnect() (on page 7-240)

tspnet.disconnect()

This function disconnects a specified TSP-Net session.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`tspnet.disconnect(`*`connectionID`*`)`

connectionID	The connection ID returned from `tspnet.connect()`

Details

This function disconnects the two devices by closing the connection. The *connectionID* is the session handle returned by `tspnet.connect()`.

For TSP-enabled devices, this aborts any remotely running commands or scripts.

Example

```
testID = tspnet.connect("192.0.2.0")
-- Use the connection
tspnet.disconnect(testID)
```

Create a TSP-Net session.
Close the session.

Also see

tspnet.connect() (on page 7-239)

tspnet.execute()

This function sends a command string to the remote device.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
tspnet.execute("connectionID", "commandString")
value1 = tspnet.execute("connectionID", "commandString", formatString)
value1, value2 = tspnet.execute("connectionID", "commandString", formatString)
value1, ..., valueN = tspnet.execute("connectionID", "commandString", formatString)
```

connectionID	The connection ID returned from `tspnet.connect()`
commandString	The command to send to the remote device
value1	The first value decoded from the response message
value2	The second value decoded from the response message
valueN	The *N* th value decoded from the response message; there is one return value for each format specifier in the format string
...	One or more values separated with commas
formatString	Format string for the output

Details

This command sends a command string to the remote instrument. A termination is added to the command string when it is sent to the remote instrument (`tspnet.termination()`). You can also specify a format string, which causes the command to wait for a response from the remote instrument. The Model 707B or 708B decodes the response message according to the format specified in the format string and returns the message as return values from the function (see `tspnet.read()` for format specifiers).

When this command is sent to a TSP-enabled instrument, the Model 707B or 708B suspends operation until a timeout error is generated or until the instrument responds. The TSP prompt from the remote instrument is read and discarded. The Model 707B or 708B places any remotely generated errors into its error queue. When the optional format string is not specified, this command is equivalent to `tspnet.write()`, except that a termination is automatically added to the end of the command.

Example 1

```
tspnet.execute(instrumentID, "runScript()")
```

Command the remote device to run a script named `runScript`.

Example 2

```
tspnet.timeout = 5
id_instr = tspnet.connect("192.0.2.23", 23, "*rst\r\n")
tspnet.termination(id_instr, tspnet.TERM_CRLF)
tspnet.execute(id_instr, "*idn?")
print("tspnet.execute returns:", tspnet.read(id_instr))
```

Print the `*idn?` string from the remote device.

Also see

tspnet.connect() (on page 7-239)
tspnet.read() (on page 7-243)
tspnet.termination() (on page 7-245)
tspnet.write() (on page 7-250)

tspnet.idn()

This function retrieves the response of the remote device to `*IDN?`.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`idnString = tspnet.idn(connectionID)`

idnString	The returned `*IDN?` string
connectionID	The connection ID returned from `tspnet.connect()`

Details

This function retrieves the response of the remote device to `*IDN?`.

Example

```
deviceID = tspnet.connect("192.0.2.1")
print(tspnet.idn(deviceID))
tspnet.disconnect(deviceID)
```

Assume the instrument is at IP address 192.0.2.1.
The output that is produced when you connect to the instrument and read the identification string may appear as:
`KEITHLEY INSTRUMENTS INC.,MODEL 707B,00000170,01.10h`

Also see

tspnet.connect() (on page 7-239)

tspnet.read()

This function reads data from a remote device.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
value1 = tspnet.read(connectionID)
value1 = tspnet.read(connectionID, formatString)
value1, value2 = tspnet.read(connectionID, formatString)
value1, ..., valueN = tspnet.read(connectionID, formatString)
```

value1	The first value decoded from the response message
value2	The second value decoded from the response message
valueN	The *N* th value decoded from the response message; there is one return value for each format specifier in the format string
...	One or more values separated with commas
connectionID	The connection ID returned from `tspnet.connect()`
formatString	Format string for the output, maximum of 10 specifiers

Details

This command reads available data from the remote instrument and returns responses for the specified number of arguments.

The format string can contain the following specifiers:

`%[width]s`	Read data until the specified length
`%[max width]t`	Read data until the specified length or until punctuation is found, whichever comes first
`%[max width]n`	Read data until a newline or carriage return
`%d`	Read a number (delimited by punctuation)

A maximum of 10 format specifiers can be used for a maximum of 10 return values.

If *formatString* is not provided, the command returns a string that contains the data until a new line is reached. If no data is available, the Model 707B or 708B pauses operation until the requested data is available or until a timeout error is generated. Use `tspnet.timeout` to specify the timeout period.

When the Model 707B or 708B reads from a TSP-enabled remote instrument, the Model 707B or 708B removes Test Script Processor (TSP™) prompts and places any errors it receives from the remote instrument into its own error queue. Errors from the remote device are prefaced with "`Remote Error`," followed by the error number and error description.

Example

```
tspnet.write(deviceID, "*idn?\r\n")

print("write/read returns:", tspnet.read(deviceID))
```

Send the `"*idn?\r\n"` message to the instrument connected as `deviceID`.
Display the response that is read from `deviceID` (based on the `*idn?` message).

Also see

tspnet.connect() (on page 7-239)
tspnet.readavailable() (on page 7-244)
tspnet.timeout (on page 7-246)
tspnet.write() (on page 7-250)

tspnet.readavailable()

This function checks if output data is available from the remote device.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

bytesAvailable = tspnet.readavailable(*connectionID*)

bytesAvailable	The number of bytes available to be read from the connection
connectionID	The connection ID returned from `tspnet.connect()`

Details

No data is read from the instrument. This allows TSP scripts to continue to run without waiting on a remote command to finish.

Example

```
ID = tspnet.connect("192.0.2.1")
tspnet.write(ID, "*idn?\r\n")
repeat bytes = tspnet.readavailable(ID) until bytes > 0
print(tspnet.read(ID))
tspnet.disconnect(ID)
```

Send commands that create data.
Wait for data to be available.

Also see

tspnet.connect() (on page 7-239)
tspnet.read() (on page 7-243)

tspnet.reset()

This function disconnects all TSP-Net sessions.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
tspnet.reset()
```

Details

This command disconnects all remote instruments connected through TSP-Net. For TSP-enabled devices, this causes any commands or scripts running remotely to be terminated.

Also see

None

tspnet.termination()

This function sets the device line termination sequence.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
type = tspnet.termination(connectionID)
type = tspnet.termination(connectionID, termSequence)
```

type	An enumerated value indicating the termination type: ▪ `1` or `tspnet.TERM_LF` ▪ `4` or `tspnet.TERM_CR` ▪ `2` or `tspnet.TERM_CRLF` ▪ `3` or `tspnet.TERM_LFCR`
connectionID	The connection ID returned from `tspnet.connect()`
termSequence	The termination sequence

Details

This function sets and gets the termination character sequence that is used to indicate the end of a line for a TSP-Net connection.

Using the *termSequence* parameter sets the termination sequence. The present termination sequence is always returned.

For the *termSequence* parameter, use the same values as *type*. There are four possible combinations, all of which are made up of line feeds (LF or 0x10) and carriage returns (CR or 0x13). For TSP-enabled devices, the default is `tspnet.TERM_LF`. For devices that are not TSP-enabled, the default is `tspnet.TERM_CRLF`.

Example

```
deviceID = tspnet.connect("192.0.2.1")
if deviceID then
   tspnet.termination(deviceID, tspnet.TERM_LF)
end
```

Sets termination type for IP address 192.0.2.1 to `TERM_LF`.

Also see

tspnet.connect() (on page 7-239)
tspnet.disconnect() (on page 7-240)

tspnet.timeout

This attribute sets the timeout value for the `tspnet.connect()`, `tspnet.execute()`, and `tspnet.read()` commands.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	No	Instrument reset Recall setup	Create configuration script Save setup	20.0 (20 s)

Usage

```
value = tspnet.timeout
tspnet.timeout = value
```

value	The timeout duration in seconds: 0.001 to 30.0

Details

This attribute sets the amount of time the `tspnet.connect()`, `tspnet.execute()`, and `tspnet.read()` commands wait for a response.

The time is specified in seconds. The timeout may be specified to millisecond resolution but is only accurate to the nearest 10 ms.

Example

`tspnet.timeout = 2.0`
Sets the timeout duration to 2 s.

Also see

tspnet.connect() (on page 7-239)
tspnet.execute() (on page 7-241)
tspnet.read() (on page 7-243)

tspnet.tsp.abort()

This function causes the TSP-enabled instrument to stop executing any of the commands that were sent to it.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
tspnet.tsp.abort(connectionID)
```

connectionID	Integer value used as a handle for other `tspnet` commands

Details

This function is appropriate only for TSP-enabled instruments.

Sends an abort command to the remote instrument.

Example

`tspnet.tsp.abort(testConnection)`
Stops remote instrument execution on `testConnection`.

Also see

None

tspnet.tsp.abortonconnect

This attribute contains the setting for abort on connect to a TSP-enabled instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Attribute (RW)	No	Instrument reset Recall setup	Create configuration script	1 (enable)

Usage

```
tspnet.tsp.abortonconnect = value
value = tspnet.tsp.abortonconnect
```

value	■ Enable: `1` ■ Disable: `0`

Details

This setting determines if the instrument sends an abort message when it attempts to connect to a TSP-enabled instrument using the `tspnet.connect()` function.

When you send the abort command on an interface, it causes any other active interface on that instrument to close. If you do not send an abort command (or if `tspnet.tsp.abortonconnect` is set to 0) and another interface is active, connecting to a TSP-enabled remote instrument results in a connection. However, the instrument does not respond to subsequent reads or executes because control of the instrument is not obtained until an abort command has been sent.

Example

`tspnet.tsp.abortonconnect = 0`
Configure the instrument so that it does not send an abort command when connecting to a TSP-enabled instrument.

Also see

tspnet.connect() (on page 7-239)

tspnet.tsp.rbtablecopy()

This function copies a reading buffer synchronous table from a remote instrument to a TSP-enabled instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
table = tspnet.tsp.rbtablecopy(connectionID, "name")
table = tspnet.tsp.rbtablecopy(connectionID, "name", startIndex, endIndex)
```

table	A copy of the synchronous table or a string
connectionID	Integer value used as a handle for other `tspnet` commands
name	The full name of the reading buffer name and synchronous table to copy
startIndex	Integer start value
endIndex	Integer end value

Details

This function is only appropriate for TSP-enabled instruments.

This function reads the data from a reading buffer on a remote instrument and returns an array of numbers or a string representing the data. The *startIndex* and *endIndex* parameters specify the portion of the reading buffer to read. If no index is specified, the entire buffer is copied.

The function returns a table if the table is an array of numbers; otherwise, a comma-delimited string is returned.

This command is limited to transferring 50,000 readings at a time.

Example 1

```
t = tspnet.tsp.rbtablecopy(testConnection, "testRemotebuffername.readings", 1, 3)
print(t[1], t[2], t[3])
```

Copies the specified readings table for buffer items 1 through 3, then displays the first three readings. Example output:

```
4.5653423423e-1
4.5267523423e-1
4.5753543423e-1
```

Example 2

```
times = tspnet.tsp.rbtablecopy(testTspdevice,
          "testRemotebuffername.timestamps", 1, 3)
print(times)
```

Copy the specified timestamps table for items 1 through 3, then display the table.
Example output:

```
01/01/2011 10:10:10.0000013,01/01/2011 10:10:10.0000233,01/01/2011
   10:10:10.0000576
```

Also see

None

tspnet.tsp.runscript()

This function loads and runs a script on a remote TSP-enabled instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	Yes			

Usage

```
tspnet.tsp.runscript(connectionID, "script")
tspnet.tsp.runscript(connectionID, "name", "script")
```

connectionID	Integer value used as an identifier for other `tspnet` commands
name	The name that is assigned to the script
script	The body of the script as a string

Details

This function is appropriate only for TSP-enabled instruments.

This function downloads a script to a remote instrument and runs it. It automatically adds the appropriate `loadscript` and `endscript` commands around the script, captures any errors, and reads back any prompts. No additional substitutions are done on the text.

The script is automatically loaded, compiled, and run.

Any output from previous commands is discarded.

This command does not wait for the script to complete.

If you do not want the script to do anything immediately, make sure the script only defines functions for later use. Use the `tspnet.execute()` function to execute those functions later.

If no name is specified, the script is loaded as the anonymous script.

Example

```
tspnet.tsp.runscript(myconnection, "mytest",
"print([[start]]) for d = 1, 10 do print([[work]]) end print([[end]])")
```

Load and run a script entitled `mytest` on the TSP-enabled instrument connected with `myconnection`.

Also see

tspnet.execute() (on page 7-241)

tspnet.write()

This function writes a string to the remote instrument.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
tspnet.write(connectionID, "inputString")
```

connectionID	The connection ID returned from `tspnet.connect()`
inputString	The string to be written

Details

The `tspnet.write()` function sends *`inputString`* to the remote instrument. It does not wait for command completion on the remote instrument.

The Model 707B or 708B sends *`inputString`* to the remote instrument exactly as indicated. The *`inputString`* must contain any necessary new lines, termination, or other syntax elements needed to complete properly.

Because `tspnet.write()` does not process output from the remote instrument, do not send commands that generate too much output without processing the output. This command can stop executing if there is too much unprocessed output from previous commands.

Example

`tspnet.write(myID, "runscript()\r\n")`
Commands the remote instrument to execute a command or script named `runscript()` on a remote device identified in the system as `myID`.

Also see

tspnet.connect() (on page 7-239)
tspnet.read() (on page 7-243)

userstring.add()

This function adds a user-defined string to nonvolatile memory.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
userstring.add("name", "value")
```

name	The name of the string; the key of the key-value pair
value	The string to associate with *`name`*; the value of the key-value pair

Details

This function associates the string *value* with the string *name* and stores this key-value pair in nonvolatile memory.

Use the `userstring.get()` function to retrieve the *value* associated with the specified *name*.

You can use the `userstring` functions to store custom, instrument-specific information in the instrument, such as department number, asset number, or manufacturing plant location.

Example

```
userstring.add("assetnumber", "236")
userstring.add("product", "Widgets")
userstring.add("contact", "John Doe")
for name in userstring.catalog() do
   print(name .. " = " ..
      userstring.get(name))
end
```

Stores user-defined strings in nonvolatile memory and recalls them from the instrument using a for loop.
Example output:

```
assetnumber = 236
contact = John Doe
product = Widgets
```

Also see

userstring.catalog() (on page 7-251)
userstring.delete() (on page 7-252)
userstring.get() (on page 7-253)

userstring.catalog()

This function creates an iterator for the user-defined string catalog.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

`for` *name* `in userstring.catalog() do` *body* `end`

name	The name of the string; the key of the key-value pair
body	Code to execute in the body of the for loop

Details

The catalog provides access for user-defined string pairs, allowing you to manipulate all the key-value pairs in nonvolatile memory. The entries are enumerated in no particular order.

Example 1

```
for name in userstring.catalog() do
   userstring.delete(name)
end
```

Deletes all user-defined strings in nonvolatile memory.

Example 2

```
userstring.add("assetnumber", "236")
userstring.add("product", "Widgets")
userstring.add("contact", "John Doe")
for name in userstring.catalog() do
   print(name .. " = " ..
      userstring.get(name))
end
```

Prints all `userstring` key-value pairs.

Output:

```
product = Widgets
assetnumber = 236
contact = John Doe
```

Notice the key-value pairs are not listed in the order they were added.

Also see

userstring.add() (on page 7-250)
userstring.delete() (on page 7-252)
userstring.get() (on page 7-253)

userstring.delete()

This function deletes a user-defined string from nonvolatile memory.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
userstring.delete("name")
```

name	The name (key) of the key-value pair of the user-defined string to delete

Details

This function deletes the string that is associated with *name* from nonvolatile memory.

Example

```
userstring.delete("assetnumber")
userstring.delete("product")
userstring.delete("contact")
```

Deletes the user-defined strings associated with the `assetnumber`, `product`, and `contact` names.

Also see

userstring.add() (on page 7-250)
userstring.catalog() (on page 7-251)
userstring.get() (on page 7-253)

userstring.get()

This function retrieves a user-defined string from nonvolatile memory.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
value = userstring.get("name")
```

value	The value of the user-defined string key-value pair
name	The name (key) of the user-defined string

Details

This function retrieves the string that is associated with *name* from nonvolatile memory.

Example

```
userstring.add("assetnumber", "236")
value = userstring.get("assetnumber")
print(value)
```

Create the user-defined string `assetnumber`, set to a value of `236`.
Read the value associated with the user-defined string named `assetnumber`.
Store it in a variable called `value`, then print the variable `value`.
Output:
`236`

Also see

userstring.add() (on page 7-250)
userstring.catalog() (on page 7-251)
userstring.delete() (on page 7-252)

waitcomplete()

This function waits for all previously started overlapped commands to complete.

Type	TSP-Link accessible	Affected by	Where saved	Default value
Function	No			

Usage

```
waitcomplete()
waitcomplete(group)
```

group	Specifies which TSP-Link group on which to wait

Details

There are two types of instrument commands:

- **Overlapped commands:** Commands that allow the execution of subsequent commands while instrument operations of the overlapped command are still in progress.
- **Sequential commands:** Commands whose operations must finish before the next command is executed.

The `waitcomplete()` command suspends the execution of commands until the instrument operations of all previous overlapped commands are finished. This command is not needed for sequential commands.

Presently, the 707B and 708B have no overlapped commands implemented. However, other TSP-enabled products, such as the Series 2600B System SourceMeter™ Instruments, have overlapped commands. Therefore, when the 707B or 708B is a TSP master to a subordinate device with overlapped commands, use this function to wait until all overlapped operations are completed.

A group number may only be specified when this node is the master node.

If no *group* is specified, the local group is used.

If zero (0) is specified for the *group*, this function waits for all nodes in the system.

NOTE

Any nodes that are not assigned to a group (group number is 0) are part of the master node group.

Example 1

`waitcomplete()`
Waits for all nodes in the local group.

Example 2

`waitcomplete(G)`
Waits for all nodes in group G.

Example 3

`waitcomplete(0)`
Waits for all nodes on the TSP-Link network.

Also see

None

Work with the 4200-SCS or 4200A-SCS

In this section:

Introduction

You can connect the Model 4200 or 4200A Parameter Analyzer to a Model 707B or 708B. This enables the 4200-SCS or 4200A-SCS to automate measurements to multiple devices for DC, CV, and pulse I-V characterization.

This section describes:

- How to configure the 707B or 708B for use with a 4200-SCS or 4200A-SCS
- How to add the Model 707B or 708B to a 4200-SCS or 4200A-SCS
- Where to look for more information

Equipment required to run the example

To run this test, you will need the following equipment:

- One 4200-SCS or 4200A-SCS instrument
- GPIB cable to connect the Model 707B or 708B to the 4200-SCS or 4200A-SCS
- Model 707B or 708B and associated matrix cards

Configure Model 707B or 708B for 4200-SCS or 4200A-SCS

To use the Model 707B or 708B with the 4200-SCS or 4200A-SCS, you need to configure the Model 707B or 708B to use compatible 707B and 708B commands.

To configure the Model 707B or 708B:

1. Press **MENU**.
2. Select **DDC**.
3. Select **ENABLE**.

Figure 82: Front-panel DDC enable menu

4. Select **70xB-VERSION**. The instrument displays the message `Must Power Cycle for DDC to enable`.
5. Press **ENTER**.
6. Cycle power to the instrument.

NOTE

When DDC mode is enabled (70xA-VERSION or 70xB-VERSION), the Model 707B or 708B sends a different identification string in response to the `*IDN?` query. With DDC mode enabled, the response of the 707B to `*IDN?` is:
`707A03`

With DDC mode disabled (OFF), the response to `*IDN?` is:
`KEITHLEY INSTRUMENTS INC, MODEL 707B, <serial #>, <firmware rev>`

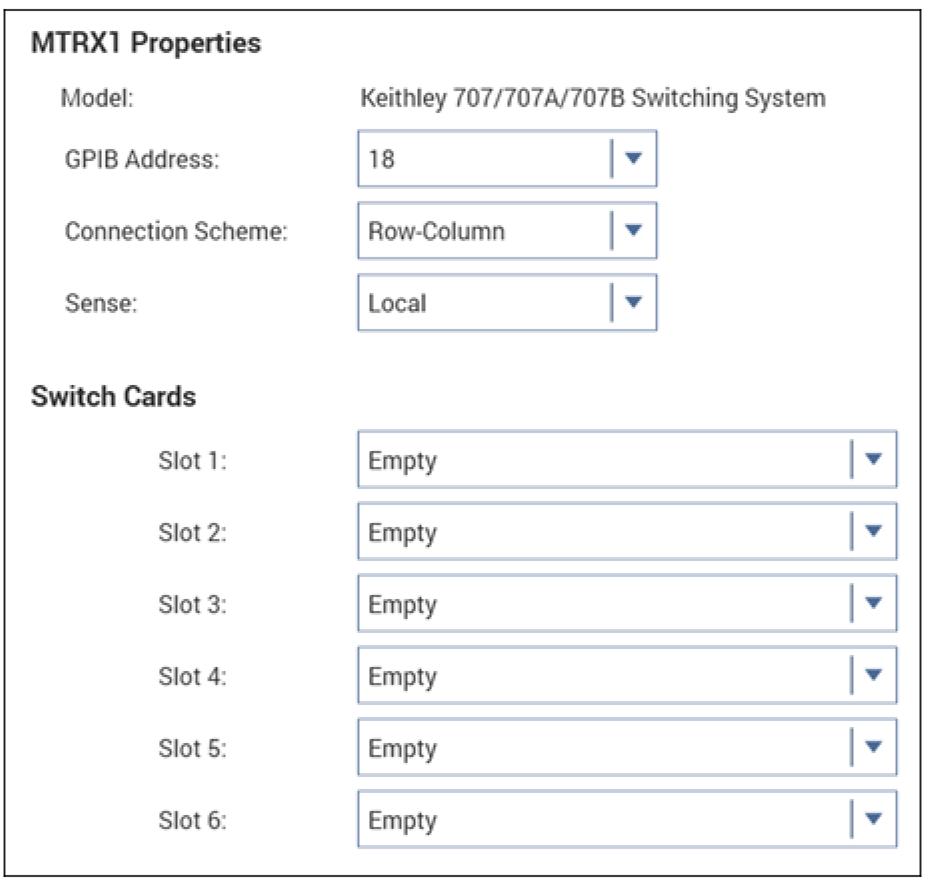
Add Model 707B or 708B to 4200A-SCS

You use the Keithley Configuration Utility (KCon) to manage the configuration of all instrumentation controlled by the 4200A-SCS software. To use the 4200A-SCS to control a switch matrix, you must add the switch matrix to the system configuration using KCon.

To add the 707B and 708B to a 4200A-SCS:

1. Connect the Model 707B or 708B to the GPIB controller on the 4200A-SCS.
2. Ensure that the Model 707B or 708B has a GPIB address that is different than any other instrument connected to the 4200A-SCS and is different than the address of the GPIB controller.
3. On the 4200A-SCS, exit Clarius.
4. On the Windows desktop, select the **KCon** icon.
5. Select **Add External Instrument**.
6. Select **Keithley 707/707A/707B Switching Matrix** or **Keithley 708/708A/708B Switching Matrix**.
7. Select **OK**.
8. In the System Configuration list, select the switching matrix. The properties are displayed. The following figure shows the properties for the 707/707A/707B. If the 708/708A/708B mainframe is selected, there is only one Switch Card slot.

Figure 83: KCon MTRX1 Properties

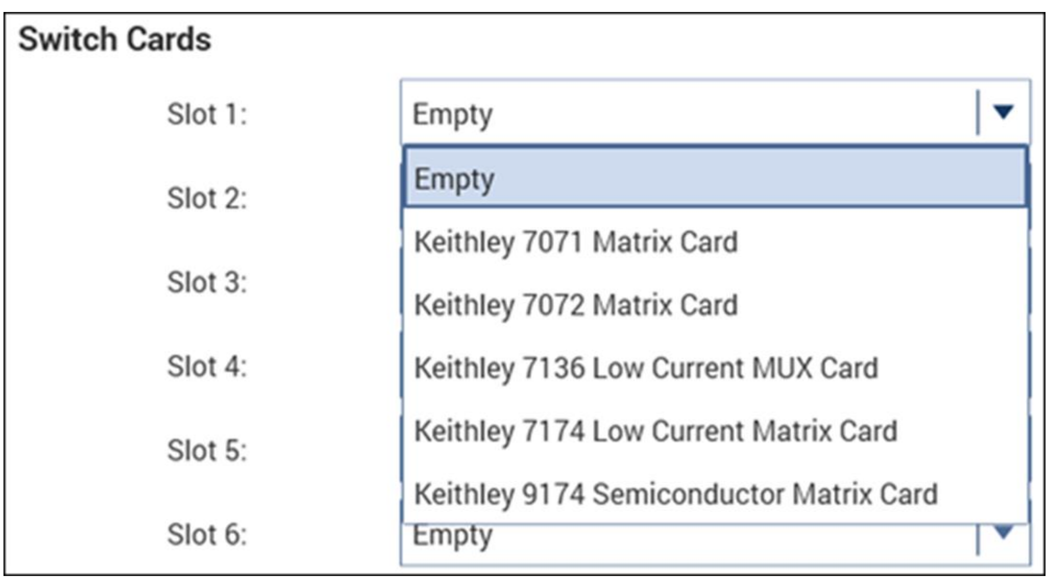


9. Select the GPIB Address from the list. Addresses that are in use are displayed with asterisks (*) next to them. The range of addresses is 0 to 30 (GPIB address 31 is reserved as the Models 707B and 708B controller address). If the selected GPIB address conflicts with the GPIB address of another system component, a red exclamation-point symbol (!) is displayed next to the selected address.
10. Select **Save** to save the changes.
11. Select the **Connection Scheme** from the list:
    - If you are connecting the instrumentation to matrix rows and the device under test (DUT) to matrix columns, select **Row-Column**.
    - If all connections (instrumentation and DUT) are made to matrix columns only, select **Instrument Card**.
12. Select **Local Sense** or **Remote Sense**:
    - For 2-wire connections to the DUT, select **Local Sense**.
    - For 4-wire connections to the DUT, select **Remote Sense**.
13. For each slot that contains a matrix card, select the model number of the matrix card.
14. For each slot that is empty, select **Empty**.

NOTE

You cannot mix matrix card models. For example, if you set slot 1 to Keithley 7174 Low Current Matrix Card, all other slots can only be set to the 7174 or Empty. To select a different model, you must set all slots to Empty and then make the new selection.

Figure 84: Assign switch cards to slots



15. In the System Configuration list, expand the switching matrix.
16. Select the card. The properties are displayed. Each row and column has a list to set the card properties. If the row-column connection scheme is selected, instruments are assigned to the rows and the test fixture pins or probe pins are assigned to the columns. If the instrument card connection scheme is selected, both instrumentation and test fixture/probe pins are assigned to columns.

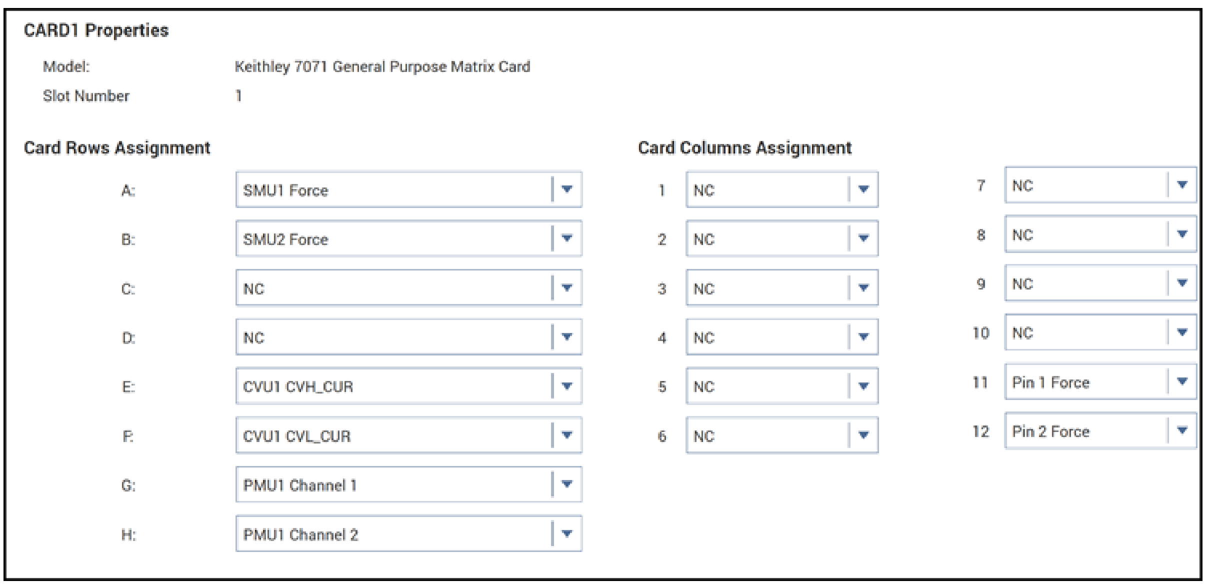
    The following figure shows the matrix card **Properties** configuration that is required to support the physical connection configuration.
17. Select from the lists to connect the rows and columns to instrument terminals and prober or test fixture pins. Card properties must match the actual physical connections to the matrix card.

    In the following figure, the lists labeled **A** to **H** correspond to the eight rows of all Keithley Series 700 compatible matrix cards. The lists labeled **1** to **12** correspond to the 12 columns of Series 700 matrix cards.

NOTE

Prober or test-fixture pins are always connected to matrix card columns.

Figure 85: Keithley 7071 Matrix Card Properties



18. Select **Validate**.
19. Select **Save** to save the configuration.

You can now use the Model 707B or 708B with the 4200A-SCS.

Add Model 707B or 708B to 4200-SCS

Use these instructions to add a Model 707B or 708B to a 4200-SCS.

To add the 707B or 708B to the 4200-SCS:

1. Connect the Model 707B or 708B to the GPIB controller on the 4200-SCS.
2. Ensure that the Model 707B or 708B has a GPIB address that is different than any other instrument connected to the 4200-SCS and is different than the address of the controller.
3. On the 4200-SCS, open KCon.
4. In KCon, open the **Tools** menu.
5. Select **Add External Instrument**.
6. Select **Switch Matrix**.
7. Select **Model 707/707A Switching Matrix** or **Model 708/708A Switching Matrix**. Note that the instrument is added to the Configuration Navigator. In the KCon Workspace, the Switch System Properties dialog is displayed.
8. Enter the appropriate GPIB Address for the Model 707B or 708B.
9. Select an instrument connection scheme.
10. Select the matrix cards that you have already installed in the Model 707B or 708B.
11. From the Tools menu, check the configuration by selecting **Validate Configuration**.
12. When the 4200-SCS finds the Model 707B or 708B, a green check mark is displayed next to “Verifying Keithley 707A Switching Matrix” or “Verifying Keithley 708A Switching Matrix.” If there is an error, a red X is displayed. An error description is provided. Generally, these errors are associated with an incorrect GPIB address.
13. From the File menu, save the current configuration by selecting **Save Configuration**.

You can now use the Model 707B or 708B with the 4200-SCS.

Additional information

For the 4200A-SCS, refer to the *Clarius Learning Center* “Using switch matrices” topics and the video “Configuring a Series 700 switch matrix in KCon.”

For the 4200-SCS, refer to the *Model 4200-SCS Reference Manual*, “Appendix B: Using Switch Matrices,” “Key Concepts” section, which includes:

- Switch matrix connection scheme settings for row-column and instrument cards
- Configuring matrix card property configurations in KCon
- Connecting the Model 4200-SCS instrument to the Model 707B or 708B
- Accessing and using the user module library to control the switch matrix

Also see “Section 15: Multi-Frequency C-V measurements,” which includes information on using the 4200-CVU card with a switch matrix.

To customize the switch user library of the 4200-SCS or 4200A-SCS, see Models 707A and 708A compatibility mode (on page 14-1).

Work with the Series 2600B

In this section:

About the Series 2600B examples

This section shows you how to use the Model 707B or 708B with a Series 2600B System SourceMeter™ instrument (SMU) to perform an I-V sweep across a diode. In each example, the Model 707B or 708B connects a diode across the output of SMUA of the SMU. The SMU then performs an I-V sweep across the diode.

There are four examples in this section, progressing from using direct control over the remote communications interface to automating the sweep using the scan and trigger model.

Example 1 shows you how to control the Series 2600B and Model 707B or 708B instruments using direct control over the remote communications interface. This example:

- Demonstrates how to use the computer to send commands to each instrument separately and cannot be run using TSB Embedded. It requires a separate application development environment.
- Requires two communications identifiers, such as the GPIB address or LAN IP address. One is required for the Model 707B or 708B and the other for the Series 2600B.
- Does not require TSP-Link.
- Does not use hardware triggering or the scan and trigger model of the Model 707B or 708B.

Example 2 shows you how to use TSP-Link to configure the Series 2600B and Model 707B or 708B instruments. This example:

- Can be run using TSB Embedded.
- Demonstrates how to configure the TSP-Link interface for communications between the instruments.
- Requires unique TSP-Link node numbers for each instrument.
- Demonstrates the benefit of using the TSP-Link interface to send commands to another instrument.
- Does not require hardware triggering or the scan and trigger model of the Model 707B or 708B.

Example 3 shows you how to use the TSP-Link interface and interactive triggering with Series 2600B and Model 707B or 708B instruments. This example:

- Can be run using TSB Embedded.
- Demonstrates how to create, load, and execute a TSP script that includes functions.
- Demonstrates how to use the functions in the script to run the test.
- Demonstrates the benefit of using TSP scripts to run the test independently of computer control after initiating a test to run.
- Uses the TSP-Link interface for communications and interactive hardware triggering.
- Does not use the scan and trigger model of the Model 707B or 708B.

Example 4 shows you how to use the scan and trigger model of the Model 707B or 708B with the SourceMeter instrument. This example:

- Can be run using TSB Embedded.
- Demonstrates how to create, load, and execute a TSP script that includes functions, and how to use those functions to run the test.
- Demonstrates the benefit of using TSP scripts to run the test independently of computer control after initiating a test to run.
- Uses the TSP-Link interface for communications and interactive hardware triggering.
- Demonstrates how to use the scan and trigger model to achieve hardware synchronization during a test.
- Illustrates the speed advantage of preconfiguring the sequence of crosspoints used during the test.
- Includes instructions on how to create a scan list, how to execute a foreground scan, and how to configure the trigger model to automatically wait for and assert trigger signals.

Equipment required to run the examples

To run this test, you need the following equipment:

- Six diodes with triaxial connections.
- One Model 707B or 708B.
- One switch matrix card (Model 7174A, 7072, or 7072-HV).
- One Model 2635B SourceMeter Instrument or equivalent source measurement unit (SMU). You can use a Model 2601B or 2611B SMU if an adaptation is made from the screw terminals on the instrument to the triaxial connections on the switch card.

Connect the equipment

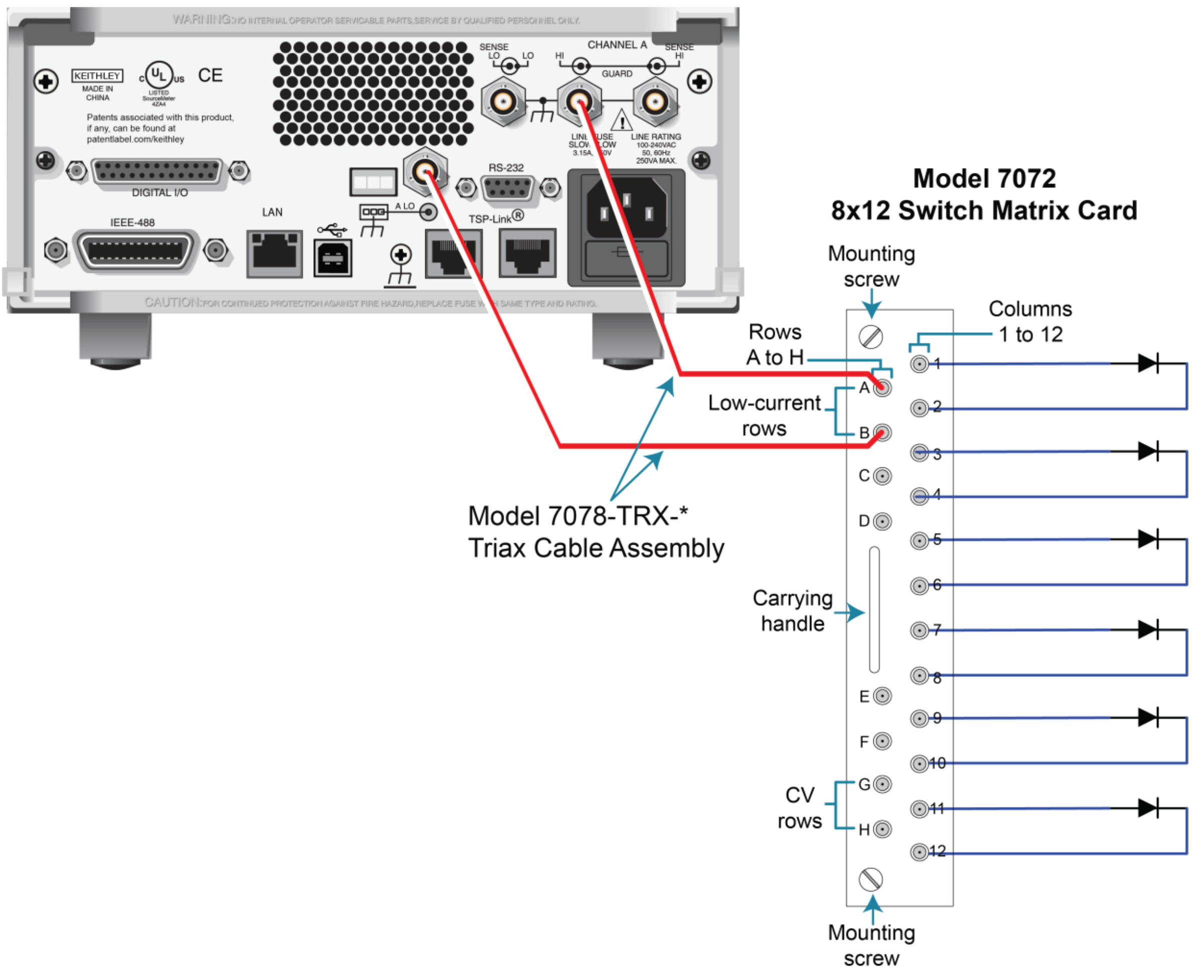
All the examples use the same test equipment setup, as shown in the following figure.

NOTE

This figure shows connections to the Model 7072 8×12 matrix card. You can make similar connections to the other switch cards listed for this example.

All triaxial connections for both the SMU and the diodes are made to the center conductor (pin) of the triaxial connector.

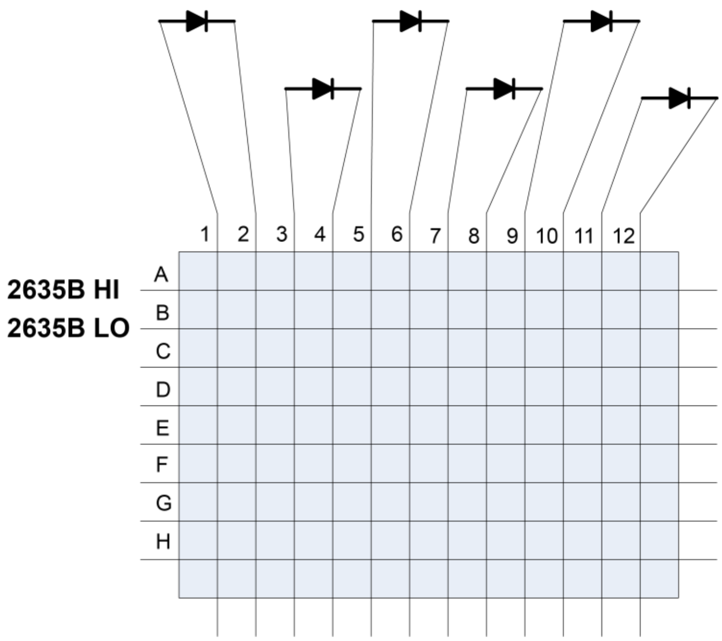
Figure 86: Connections to matrix card



Logical block diagram of test connections

The following figure shows the logical block diagram of the test connections.

Figure 87: Logical block diagram of test connections



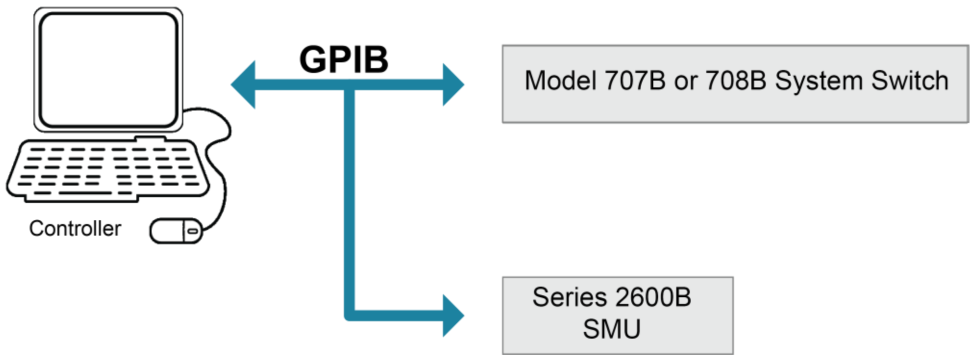
Example 1: Control with separate interfaces

This example controls Series 2600B and Model 707B or 708B instruments using separate communications interfaces. To run this test, you create a table on the Model 707B or 708B that contains the crosspoints that are needed to connect the SMU to each diode under test, one at a time. The SMU then performs an I-V sweep on the diode. Data from the SMU is returned to the controlling computer after all six diodes have been tested.

Set up communications

The communications setup is shown in the following figure. GPIB is used as an example, but this application can be run using any of the supported communications interfaces for the instruments.

Figure 88: GPIB communications setup



Example program code for the Model 707B or 708B

NOTE

The example code does not include programming syntax to issue commands from the host computer over GPIB or LAN communications interfaces. It cannot be run using TSB Embedded. You are expected to send the instrument commands by means of a separate application development environment, such as Microsoft™ Visual Studio™ or NI™ LabVIEW™.

Send the following commands to configure the Model 707B or 708B.

Example program code for the Model 707B or 708B	
Code	**Notes and comments**
`reset()` `errorqueue.clear()`	Reset the instrument to factory defaults. Clear all errors.
`matrixChannels = {}`	Define the table that details crosspoints to close to connect each diode to the source-measure unit (SMU).

`matrixChannels[1] = "1A01,1B02"`
`matrixChannels[2] = "1A03,1B04"`
`matrixChannels[3] = "1A05,1B06"`
`matrixChannels[4] = "1A07,1B08"`
`matrixChannels[5] = "1A09,1B10"`
`matrixChannels[6] = "1A11,1B12"`
	Add entries to the table.

Example program code for the Model 2635B

Send the following commands to configure the Model 2635B System SourceMeter Instrument.

Example program code for the Model 2635B System SourceMeter Instrument	
Code	**Notes and comments**

`reset()`
`errorqueue.clear()`
`smua.nvbuffer1.clear()`
	Reset the instrument.
	Clear all errors and reset status bits.
	Clear existing data from the data buffer.

`smua.source.func = smua.OUTPUT_DCAMPS` `smua.source.rangei = 0.01`	Set source function to current source. Set source current range to 10 mA.

`smua.measure.rangev = 6`
`smua.measure.nplc = 1`
`smua.nvbuffer1.appendmode = 1`
	Set voltage measurement range to 6 V.
	Set the integration rate.
	Enable buffer append mode.

`smua.trigger.source.limitv = 10`
`smua.trigger.source.lineari(0, 0.01, 11)`
`smua.trigger.source.action = smua.ENABLE`
	Set the voltage compliance during the sweep.
	Configure linear staircase sweep.
	Enable the source action during the sweep.

`smua.trigger.measure.v(smua.nvbuffer1)` `smua.trigger.measure.action = smua.ENABLE`	Configure sweep voltage measurements. Enable voltage measurements.
`smua.trigger.count = 11` `smua.trigger.arm.count = 1`	Set trigger count to number of steps in the sweep. Set arm count to number of sweeps to perform.

Example code to run the test

To run the test, send the following commands to the corresponding instrument.

Example program code to run the test		
Model	**Code**	**Notes and comments**
2635B	`smua.source.output = smua.OUTPUT_ON`	Turn on the 2635B output.
707B or 708B	`channel.exclusiveclose(matrixChannels[1])`	Connect first diode.
2635B	`smua.trigger.initiate()` `waitcomplete()`	Initiate sweep. Wait until sweep is complete.
707B or 708B	`channel.exclusiveclose(matrixChannels[2])`	Connect second diode.
2635B	`smua.trigger.initiate()` `waitcomplete()`	Initiate sweep. Wait until sweep is complete.
707B or 708B	`channel.exclusiveclose(matrixChannels[3])`	Connect third diode.
2635B	`smua.trigger.initiate()` `waitcomplete()`	Initiate sweep. Wait until sweep is complete.
Model 707B or 708B	`channel.exclusiveclose(matrixChannels[4])`	Connect fourth diode.
2635B	`smua.trigger.initiate()` `waitcomplete()`	Initiate sweep. Wait until sweep is complete.
707B or 708B	`channel.exclusiveclose(matrixChannels[5])`	Connect fifth diode.
2635B	`smua.trigger.initiate()` `waitcomplete()`	Initiate sweep Wait until sweep is complete.
707B or 708B	`channel.exclusiveclose(matrixChannels[6])`	Connect sixth diode.

Example program code to run the test		
Model	**Code**	**Notes and comments**

2635B	`smua.trigger.initiate()`
	`waitcomplete()`
	`smua.source.output = smua.OUTPUT.OFF`
		Initiate sweep.
		Wait until sweep is complete, then turn off the output.

707B or 708B	`channel.open('allslots')`	Open all crosspoints.
2635B	`printbuffer(1,smua.nvbuffer1.n, smua.nvbuffer1.readings)`	Retrieve data from the 2635B.

Example 2: Use TSP-Link to configure instruments

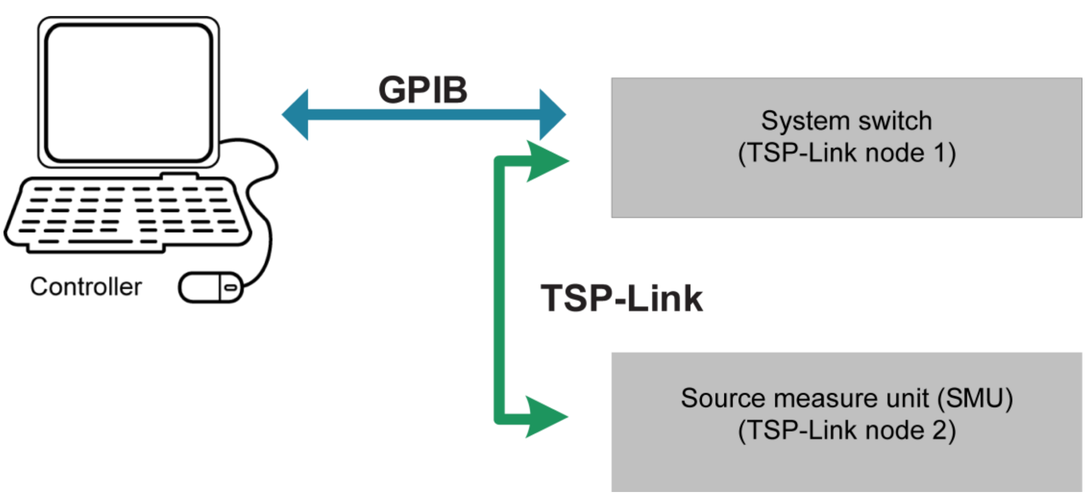
This example uses TSP-Link to configure the Series 2600B and Model 707B or 708B. This example is identical to the test in example 1, except that TSP-Link is used to communicate to the Model 2635B SMU from the Model 707B or 708B.

To run this test, you create a table on the Model 707B or 708B that contains the crosspoints that are needed to connect the SMU to each diode under test, one at a time. The SMU then performs an I-V sweep on the diode. Data from the SMU is returned to the controlling computer after all six diodes have been tested.

Set up communications

The communications setup is shown in the following figure. GPIB is used as an example, but you can run this application using any of the supported communications interfaces for the instruments. A TSP-Link connection enables communications between the two instruments and commands for the SourceMeter (on TSP-Link node 2) are sent over the TSP-Link interface.

Figure 89: GPIB communications interface with TSP-Link



To configure the TSP-Link communications interface, each instrument must have a unique TSP-Link node number. Configure the node number for the Model 707B or 708B to 1 and the SourceMeter Instrument to 2.

To set the TSP-Link node number using the front-panel interface of either instrument:

1. Press **MENU**.
2. Select **TSPLink**.
3. Select **NODE**.
4. Use the navigation wheel ◷ to adjust the node number.
5. Press **ENTER** to save the TSP-Link node number.

On the Model 707B or 708B, perform a TSP-Link reset to alert the Model 707B or 708B to the presence of the SMU:

1. Press **MENU**.
2. Select **TSPLink**.
3. Select **RESET**.

NOTE

You can also perform a TSP-Link reset from the remote command interface by sending `tsplink.reset()` to the Model 707B or 708B. This command is included in the following example program code.

NOTE

If error code 1205, "TSP-Link initialization failed (no remote nodes found)," is generated during the TSP-Link reset, ensure that the SMU has a unique TSP-Link node number.

Example program code

The following example code to configure the Model 707B or 708B and System SourceMeter Instrument is identical to the code used in Example 1. However, in this example, all commands are sent to the Model 707B or 708B. This example can be run using TSB Embedded. Commands for the Model 2635B System SourceMeter have the TSP-Link node number appended to the beginning of the command (`node[2]`).

In the following table, the Model 707B or 708B sends the commands in the shaded rows to the SMU through the TSP-Link interface. Commands in unshaded rows are executed by the Model 707B or 708B.

Example program code for the Model 707B or 708B

Code	Notes and comments

`tsplink.reset()`
`localnode.reset()`
`errorqueue.clear()`
	Reset TSP-Link.
	Reset Model 707B or 708B.
	Clear errors.

`matrixChannels = {}`	Define table that details crosspoints to close to connect each diode to the SMU.

`matrixChannels[1] = "1A01, 1B02"`
`matrixChannels[2] = "1A03, 1B04"`
`matrixChannels[3] = "1A05, 1B06"`
`matrixChannels[4] = "1A07, 1B08"`
`matrixChannels[5] = "1A09, 1B10"`
`matrixChannels[6] = "1A11, 1B12"`
	Add entries to the table.

`node[2].reset()`
`node[2].errorqueue.clear()`
`node[2].smua.nvbuffer1.clear()`
	Reset the instrument.
	Clear all errors and reset status bits.
	Clear existing data from data buffer.

`node[2].smua.source.func = node[2].smua.OUTPUT_DCAMPS` `node[2].smua.source.rangei = 0.01`	Set source function to current source. Set source current range to 10 mA.

`node[2].smua.measure.rangev = 6`
`node[2].smua.measure.nplc = 1`
`node[2].smua.nvbuffer1.appendmode = 1`
	Set voltage measurement range to 6 V.
	Set the integration rate.
	Enable buffer append mode.

`node[2].smua.trigger.source.limitv = 10`
`node[2].smua.trigger.source.lineari (0, 0.01, 11)`
`node[2].smua.trigger.source.action = node[2].smua.ENABLE`
	Set the voltage compliance during the sweep.
	Configure the linear staircase sweep.
	Enable the source action during the sweep.

`node[2].smua.trigger.measure.v (node[2].smua.nvbuffer1)` `node[2].smua.trigger.measure.action = node[2].smua.ENABLE`	Configure the sweep voltage measurements. Enable voltage measurements during the sweep.
`node[2].smua.trigger.count = 11` `node[2].smua.trigger.arm.count = 1`	Set the trigger count to the number of steps in the sweep. Set the arm count to the number of sweeps to perform each time the sweep is initiated.

Example code to run the test

To run the test, send the following commands to the Model 707B or 708B.

Example program code to run the test	
Code	**Notes and comments**
`node[2].smua.source.output = node[2].smua.OUTPUT_ON`	Turn on the output of the Model 2635B.
`channel.exclusiveclose(matrixChannels[1])`	Connect the first diode.
`node[2].smua.trigger.initiate()` `waitcomplete()`	Initiate the sweep. Both instruments wait until the sweep is complete.
`channel.exclusiveclose(matrixChannels[2])`	Connect the second diode.
`node[2].smua.trigger.initiate()` `waitcomplete()`	Initiate the sweep. Both instruments wait until the sweep is complete
`channel.exclusiveclose(matrixChannels[3])`	Connect the third diode.
`node[2].smua.trigger.initiate()` `waitcomplete()`	Initiate the sweep. Both instruments wait until the sweep is complete.
`channel.exclusiveclose(matrixChannels[4])`	Connect the fourth diode.
`node[2].smua.trigger.initiate()` `waitcomplete()`	Initiate the sweep. Both instruments wait until the sweep is complete.
`channel.exclusiveclose (matrixChannels[5])`	Connect the fifth diode.
`node[2].smua.trigger.initiate()` `waitcomplete()`	Initiate the sweep. Both instruments wait until the sweep is complete.
`channel.exclusiveclose(matrixChannels[6])`	Connect the sixth diode.

`node[2].smua.trigger.initiate()`
`waitcomplete()`
`node[2].smua.source.output = node[2].smua.OUTPUT_OFF`
	Initiate the sweep.
	Both instruments wait until the sweep is complete.
	The SMU output is turned off.

`channel.open('allslots')`	Open all crosspoints.

To retrieve data from the SMU through the TSP-Link interface, use the `printbuffer()` command to print the data to the computer communications interface. An example is shown below.

Example: Retrieve buffer data	
Code	**Notes and comments**
`printbuffer(1, node[2].smua.nvbuffer1.n, node[2].smua.nvbuffer1.readings)`	Print all readings from the SMU data buffer. Note that `readings` is a buffer attribute.

Example 3: Configuration using TSP-Link and interactive triggers

This example uses a TSP script to configure and run the test. The script includes two separate functions for configuring the Model 707B or 708B and SMU and a third function for running the test. Configuration for both instruments is similar to examples 1 and 2 except that TSP-Link trigger lines coordinate the actions between the two instruments.

When the test is run, the instruments assert trigger signals after completing their respective actions. After connecting the diode, the Model 707B or 708B sends a trigger signal to the SMU. The SMU then performs a single I-V sweep on the diode and sends a trigger signal to the 707B and 708B to indicate that the test is complete. Hardware handshaking continues in this fashion until all six diodes have been tested. Data from the SMU is returned to the controlling computer at the end of all the tests.

NOTE

A TSP script is a sequence of instrument commands and programming statements. The Model 707B or 708B can store the TSP script for later use. Using scripts allows the Model 707B or 708B to perform simple and complex tasks with less interaction with the remote interface controller (the controlling computer). For more information on using scripts, refer to Instrument programming (on page 6-1).

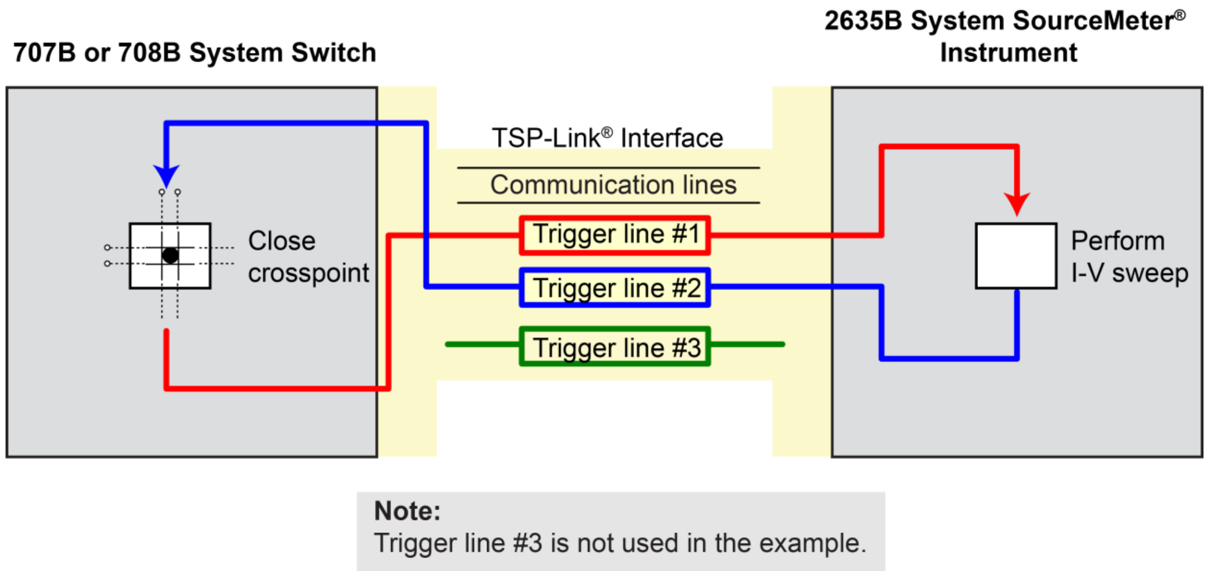
Set up communications

In this example, in addition to the TSP-Link interface serving as a communications bus, it will also serve as triggering bus to enable hardware handshaking between the System SourceMeter instrument and Model 707B or 708B.

The communication setup is identical to the setup in Example 2; see Set up communications (on page 9-8).

There are three digital lines in the TSP-Link cable that may be used for triggering. This example uses trigger lines 1 and 2. The Model 707B or 708B outputs a trigger signal on TSP-Link line 1 and waits for a trigger signal on TSP-Link line 2 before proceeding with the next set of crosspoint closures. The SMU outputs a trigger signal on TSP-Link line 2 after completing its test and waits for a trigger signal on TSP-Link line #1 before testing. The triggering setup is shown in the following figure.

Figure 90: TSP-Link interface



Example program code

In the following table, all commands are sent from the computer to the Model 707B or 708B. This example can be run using TSB Embedded. The Model 707B or 708B sends all commands in the shaded rows to the SMU through the TSP-Link interface. Commands in unshaded rows are executed by the 707B or 708B.

Example program code for the Model 707B or 708B

Code	Notes and comments
`loadscript diodeTest`	Create a script `diodeTest` to contain test configuration and execution instructions.
`function configSwitch()`	Create a function to configure the Model 707B or 708B.

`tsplink.reset()`
`localnode.reset()`
`errorqueue.clear()`
	Reset TSP-Link.
	Reset Model 707B or 708B.
	Clear errors.

`matrixChannels = {}`	Define a table that details crosspoints to close to connect each diode to the SMU.

`matrixChannels[1] = "1A01, 1B02"`
`matrixChannels[2] = "1A03, 1B04"`
`matrixChannels[3] = "1A05, 1B06"`
`matrixChannels[4] = "1A07, 1B08"`
`matrixChannels[5] = "1A09, 1B10"`
`matrixChannels[6] = "1A11, 1B12"`
	Add entries to the table.

Example program code for the Model 707B or 708B	
Code	**Notes and comments**

`tsplink.trigger[1].mode = tsplink.TRIG_FALLING`
`tsplink.trigger[2].mode = tsplink.TRIG_FALLING`
 	
`tsplink.trigger[1].clear()`
`tsplink.trigger[2].clear()`
 	
`return matrixChannels`
`end`
	Set TSP-Link lines 1 and 2 to send and receive falling-edge triggers.
	
	Clear any latched triggers on TSP-Link trigger lines 1 and 2.
	
	Return the table as an output of the function.

`function configSMU()`	Create a function to configure the SMU.

`node[2].reset()`
`node[2].errorqueue.clear()`
`node[2].smua.nvbuffer1.clear()`
	Reset the instrument.
	Clear all errors and reset status bits.
	Clear existing data from data buffer.

`node[2].smua.source.func = node[2].smua.OUTPUT_DCAMPS`
 	
`node[2].smua.source.rangei = 0.01`
	Set source function to current source.
	
	Set source current range to 10 mA.

`node[2].smua.measure.rangev = 6`
 	
`node[2].smua.measure.nplc = 1`
 	
`node[2].smua.nvbuffer1.appendmode = 1`
	Set voltage measurement range to 6 V.
	Set the integration rate.
	
	Enable buffer append mode.

`node[2].smua.trigger.source.limitv = 10`
 	
`node[2].smua.trigger.source.lineari(0, 0.01, 11)`
 	
`node[2].smua.trigger.source.action = node[2].smua.ENABLE`
	Set the voltage compliance during the sweep.
	Configure linear staircase sweep.
	
	Enable the source action during the sweep.

`node[2].smua.trigger.measure.v(node[2].smua.nvbuffer1)`
 	
`node[2].smua.trigger.measure.action = node[2].smua.ENABLE`
	Configure sweep voltage measurements.
	Enable voltage measurements during the sweep.

`node[2].smua.trigger.count = 11`
 	
`node[2].smua.trigger.arm.count = 6`
	Set trigger count to the number of steps in the sweep.
	
	Set arm count to 6, the total number of sweeps that will be run for all diodes.

`node[2].tsplink.trigger[1].mode = node[2].tsplink.TRIG_FALLING` `node[2].tsplink.trigger[2].mode = node[2].tsplink.TRIG_FALLING`	Set TSP-Link trigger lines 1 and 2 to send and receive falling edge trigger signals.
`node[2].tsplink.trigger[1].clear()` `node[2].tsplink.trigger[2].clear()`	Clear any latched triggers on TSP-Link trigger lines 1 and 2.
`node[2].smua.trigger.arm.stimulus = node[2].tsplink.trigger[1].EVENT_ID`	Configure sweep to start when signal received on TSP-Link line 1.
`node[2].tsplink.trigger[2].stimulus = node[2].smua.trigger.SWEEP_COMPLETE_EVENT_ID`	Configure System SourceMeter to output trigger signal on TSP-Link line 2 when sweep is complete.
`end`	

<table>
<tr><th colspan="2">Example program code for the Model 707B or 708B</th></tr>
<tr><th>Code</th><th>Notes and comments</th></tr>
<tr><td><pre>function runDiodeTest()</pre></td><td>This function sequences the crosspoint closures and triggering.</td></tr>
<tr><td><pre>   xptTable = configSwitch()</pre></td><td>Call the function to configure the switch storing the result in the variable <code>xptTable</code>.</td></tr>
<tr><td><pre>   configSMU()
   node[2].smua.source.output = node[2].smua.OUTPUT_ON
   node[2].smua.trigger.initiate()</pre></td><td>Call function that configures the SMU and turns on its output. Initiate the SMU so that it is ready to sweep once a trigger signal is received.</td></tr>
<tr><td><pre>   for i = 1, table.getn(xptTable) do
      if i > 1 then
         triggered = tsplink.trigger[2].wait(3)
         if triggered == true then
            channel.exclusiveclose(xptTable[i])
            tsplink.trigger[1].assert()
         else
            print("Trigger not received")
      end
         else
         channel.exclusiveclose(xptTable[i])
         tsplink.trigger[1].assert()
      end
   end</pre></td><td>Watch for a trigger signal for 3 seconds.</td></tr>
<tr><td><pre>waitcomplete()
node[2].smua.source.output = node[2].smua.OUTPUT_OFF
channel.open('allslots')
print("done")</pre></td><td>After all switch and measure actions have completed, turn off the SMU output, open all crosspoints, and print <code>done</code>; the computer controller can query for <code>done</code> to determine when the test is complete.</td></tr>
<tr><td><pre>end</pre></td><td>Ends the <code>runDiodeTest</code> function.</td></tr>
<tr><td><pre>endscript</pre></td><td>Ends the <code>diodeTest</code> script.</td></tr>
</table>

Program code to run the test

The example program code stores the program in the instrument for later use. This code runs the script to define the functions, then executes the test by calling the function that configures the instruments and sequences their actions.

Program code example

```
diodeTest()
runDiodeTest()
```

To retrieve data from the SMU through TSP-Link, use the `printbuffer()` command to print the data to the computer communications interface. The following example code prints all readings from the SMU data buffer. `readings` is a buffer attribute.

Program code example

```
printbuffer(1, node[2].smua.nvbuffer1.n, node[2].smua.nvbuffer1.readings)
```

Example 4: Use the scan and trigger model

This example demonstrates how to use the scan and trigger model of the Model 707B or 708B to maximize the speed of the test by preconfiguring the sequence of crosspoints used.

This example uses a TSP script to configure the instruments for testing. Once the instruments are configured, the Model 707B or 708B internally manages all close and open actions and any triggering signals to interface with the SMU. No separate script is required to sequence these actions.

As with the previous examples, a full I-V sweep is performed on each of the six diodes. Data from these tests is available from the System SourceMeter SMU after all tests have been completed.

Set up communications

In this example, the TSP-Link interface serves as a communications bus and as a triggering bus to enable hardware handshaking between the System SourceMeter instrument and Model 707B or 708B.

The communications setup is identical to the setup in Example 2; see Set up communications (on page 9-8).

There are three digital lines in the TSP-Link cable that you can use for triggering. This example uses trigger lines 1 and 2. The Model 707B or 708B outputs a trigger signal on TSP-Link line 1 and waits for a trigger signal on TSP-Link line 2 before proceeding with the next set of crosspoint closures. The SMU outputs a trigger signal on TSP-Link line 2 after completing its test and waits for a trigger signal on TSP-Link line 1 before testing. The triggering setup is shown in the figure named “TSP-Link interface” in Configuration using TSP-Link and interactive triggers (on page 9-12).

Program code

In the following table, all commands are sent from the computer to the Model 707B or 708B. This example can be run using TSB Embedded. The Model 707B or 708B sends all commands in the shaded rows to the System SourceMeter instrument through the TSP-Link interface. Commands in unshaded rows are executed by the Model 707B or 708B.

<table>
<tr><th colspan="2">Example program code for the Model 707B or 708B</th></tr>
<tr><th>Code</th><th>Notes and comments</th></tr>
<tr><td><pre>loadscript diodeTest</pre></td><td>Create a script <code>diodeTest</code> to contain test configuration and execution instructions.</td></tr>
<tr><td><pre>function configSwitch()</pre></td><td>Create a function to configure the Model 707B or 708B.</td></tr>
<tr><td><pre>   tsplink.reset()</pre></td><td>Reset TSP-Link.</td></tr>
<tr><td><pre>   localnode.reset()</pre></td><td>Reset Model 707B or 708B.</td></tr>
<tr><td><pre>   errorqueue.clear()</pre></td><td>Clear errors.</td></tr>
<tr><td><pre>   matrixChannels = {}</pre></td><td>Define table that details crosspoints to close to connect each diode to the SMU.</td></tr>
<tr><td><pre>   matrixChannels[1] = "1A01, 1B02"
   matrixChannels[2] = "1A03, 1B04"
   matrixChannels[3] = "1A05, 1B06"
   matrixChannels[4] = "1A07, 1B08"
   matrixChannels[5] = "1A09, 1B10"
   matrixChannels[6] = "1A11, 1B12"</pre></td><td>Add entries to the table.</td></tr>
<tr><td><pre>   for i = 1, 6 do
      scan.addimagestep(matrixChannels[i])
   end</pre></td><td>Use the values in the table to create a scan list.</td></tr>
<tr><td><pre>   tsplink.trigger[1].mode = tsplink.TRIG_FALLING
   tsplink.trigger[2].mode = tsplink.TRIG_FALLING</pre></td><td>Set TSP-Link lines 1 and 2 to send and receive falling-edge triggers.</td></tr>
<tr><td><pre>   tsplink.trigger[1].clear()
   tsplink.trigger[2].clear()</pre></td><td>Clear any latched triggers on TSP-Link trigger lines 1 and 2.</td></tr>
<tr><td><pre>   scan.bypass = scan.ON</pre></td><td>Configure the scan and trigger mode of the Model 707B or 708B. The bypass is needed for the trigger signal to start the scan.</td></tr>
<tr><td><pre>   scan.trigger.arm.stimulus = 0</pre></td><td>Arm the scan immediately.</td></tr>
<tr><td><pre>   scan.trigger.channel.stimulus =
      tsplink.trigger[2].EVENT_ID</pre></td><td>Trigger channel close action with receipt of trigger on TSP-Link line 2.</td></tr>
<tr><td><pre>   tsplink.trigger[1].stimulus =
      scan.trigger.EVENT_CHANNEL_READY</pre></td><td>Output trigger on TSP-Link line 1 after channel close action occurs.</td></tr>
<tr><td><pre>   scan.scancount = 1</pre></td><td>Loop through scan list once.</td></tr>
<tr><td><pre>end</pre></td><td>End of the <code>configSwitch</code> function.</td></tr>
</table>

Example program code for the Model 707B or 708B	
Code	**Notes and comments**
`function configSMU()`	Create a function to configure the SMU.
`node[2].reset()`	Reset the instrument.
`node[2].errorqueue.clear()`	Clear all errors and reset status bits.
`node[2].smua.nvbuffer1.clear()`	Clear existing data from data buffer.
`node[2].smua.source.func = node[2].smua.OUTPUT_DCAMPS`	Set source function to current source.
`node[2].smua.source.rangei = 0.01`	Set source current range to 10 mA.
`node[2].smua.measure.rangev = 6`	Set voltage measurement range to 6 V.
`node[2].smua.measure.nplc = 1`	Set the integration rate.
`node[2].smua.nvbuffer1.appendmode = 1`	Enable buffer append mode.
`node[2].smua.trigger.source.limitv = 10`	Set the voltage compliance during the sweep.
`node[2].smua.trigger.source.lineari(0, 0.01, 11)`	Configure a linear staircase sweep.
`node[2].smua.trigger.source.action = node[2].smua.ENABLE`	Enable the source action during the sweep.
`node[2].smua.trigger.measure.v(node[2].smua.nvbuffer1)`	Configure sweep voltage measurements.
`node[2].smua.trigger.measure.action = node[2].smua.ENABLE`	Enable voltage measurements during the sweep.
`node[2].smua.trigger.count = 11`	Set the trigger count to the number of steps in the sweep.
`node[2].smua.trigger.arm.count = 6`	Set the arm count to 6, the total number of sweeps that will be run for all diodes.
`node[2].tsplink.trigger[1].mode = node[2].tsplink.TRIG_FALLING` `node[2].tsplink.trigger[2].mode = node[2].tsplink.TRIG_FALLING`	Set TSP-Link trigger lines 1 and 2 to send and receive falling edge trigger signals.
`node[2].tsplink.trigger[1].clear()` `node[2].tsplink.trigger[2].clear()`	Clear any latched triggers on TSP-Link trigger lines 1 and 2.
`node[2].smua.trigger.arm.stimulus = node[2].tsplink.trigger[1].EVENT_ID`	Configure sweep to start when signal received on TSP-Link line 1.
`node[2].tsplink.trigger[2].stimulus = node[2].smua.trigger.SWEEP_COMPLETE_EVENT_ID`	Configure the SMU to output trigger signal on TSP-Link line 2 when the sweep is complete.
`end`	End of the function `ConfigSMU`.

Example program code for the Model 707B or 708B	
Code	**Notes and comments**
`function runDiodeTest()`	This function sequences the crosspoint closures and triggering.
`configSwitch()`	Call function to configure the Model 707B or 708B.

`configSMU()`
`node[2].smua.source.output = node[2].smua.OUTPUT_ON`
 	
`node[2].smua.trigger.initiate()`
 	
`scan.execute()`
`waitcomplete()`
	Call function that configures the SMU and turns on its output.
	Initiate the SMU so that it is ready to sweep once a trigger signal is received. The output turns on but no power is applied.
	Initiate the 707B or 708B to start a foreground scan. Waits until the 2635B finishes all measurements. Remote operation pauses until all six tests are complete. For other options when running a scan, see Using background scans for longer scan lists (on page 9-20).

`node[2].smua.source.output = node[2].smua.OUTPUT_OFF` `channel.open('allslots')`	Turn off the SMU output. Open all crosspoints.
`end`	Ends the `runDiodeTest` function.
`endscript`	Ends the `diodeTest` script.

Program code to run the test

The example program code stores the program in the instrument for later use. This code runs the script to define the functions, then executes the test by calling the function that configures the instruments and sequences their actions.

Program code example

```
diodeTest()
runDiodeTest()
```

To retrieve data from the SMU through TSP-Link, use the `printbuffer()` command to print the data to the computer communications interface. The following example code prints all readings from the SMU data buffer. `readings` is a buffer attribute.

Program code example

```
printbuffer(1, node[2].smua.nvbuffer1.n, node[2].smua.nvbuffer1.readings)
```

Use background scans for longer scan lists

The previous example demonstrates use of the scan and trigger model to create a foreground scan.

While running a foreground scan, you must wait for the scan to complete or you must abort the scan before you can query the instrument state or any reading buffers. For a scan with a few channels or crosspoints in the scan list, this might not be a problem. However, when there are many channels in the scan list, or when scans are run over long periods, it can be useful to determine instrument state. In these situations, you can run a background scan. A background scan allows you to query settings during a scan.

To change Example 4 from a foreground scan to a background scan, in the `runDiodeTest()` function, change `scan.execute()` to `scan.background()`. You can then use the `scan.state()` command to query the status of the scan, including the number of steps in the scan, the number of steps performed, and the number of entire scans performed.

The following code shows a revised example of the `runDiodeTest` function in Example 4.

Program code description

Create the script `runDiodeTest` to contain test configuration and execution instructions.

Call the `configSwitch()` function to configure the Model 707B or 708B.

Call the `configSMU()` function to configure the SMU and turn on its output.

Initiate the SMU so that it is ready to sweep once a trigger signal is received. Note that the output turns on but no power is applied.

Initiate the System Switch to start a background scan.

Delay 2 seconds to allow the scan to start.

Use variables to hold the scan state response.

Print the response to scan state, which you can use to determine when the scan is complete.

Query the scan state every second to determine when the scan completes.

When the scan is complete (scan state = 6), exit the loop.

Turn off the SMU output.

Open all crosspoints.

End the `runDiodeTest` function.

Program code example

```
function runDiodeTest()
   configSwitch()
   configSMU()
   node[2].smua.source.output = node[2].smua.OUTPUT_ON
   node[2].smua.trigger.initiate()
   scan.background()
   delay(2)
   scanState, scanCount, stepCount = scan.state()
   print(scanState, " ", scanCount, " ", stepCount, " ")
   while scanState ~=6 do
      delay(1)
   end
   node[2].smua.source.output = node[2].smua.OUTPUT_OFF
   channel.open('allslots')
end
```

Section 10

Work with the Series 2400

In this section:

About the Series 2400 examples

This section provides examples that demonstrate the basic interaction of the Model 707B or 708B with a Model 2400 SourceMeter Instrument (SMU). One of the examples also shows how to use digital I/O triggering to synchronize source and measure actions between the Model 707B or 708B and the SMU.

The concepts in these examples can be applied when using the Model 707B or 708B with other SCPI-based instruments. TSB Embedded cannot be used to program the Model 2400.

Equipment required to run the examples

To run this test, you need the following equipment:

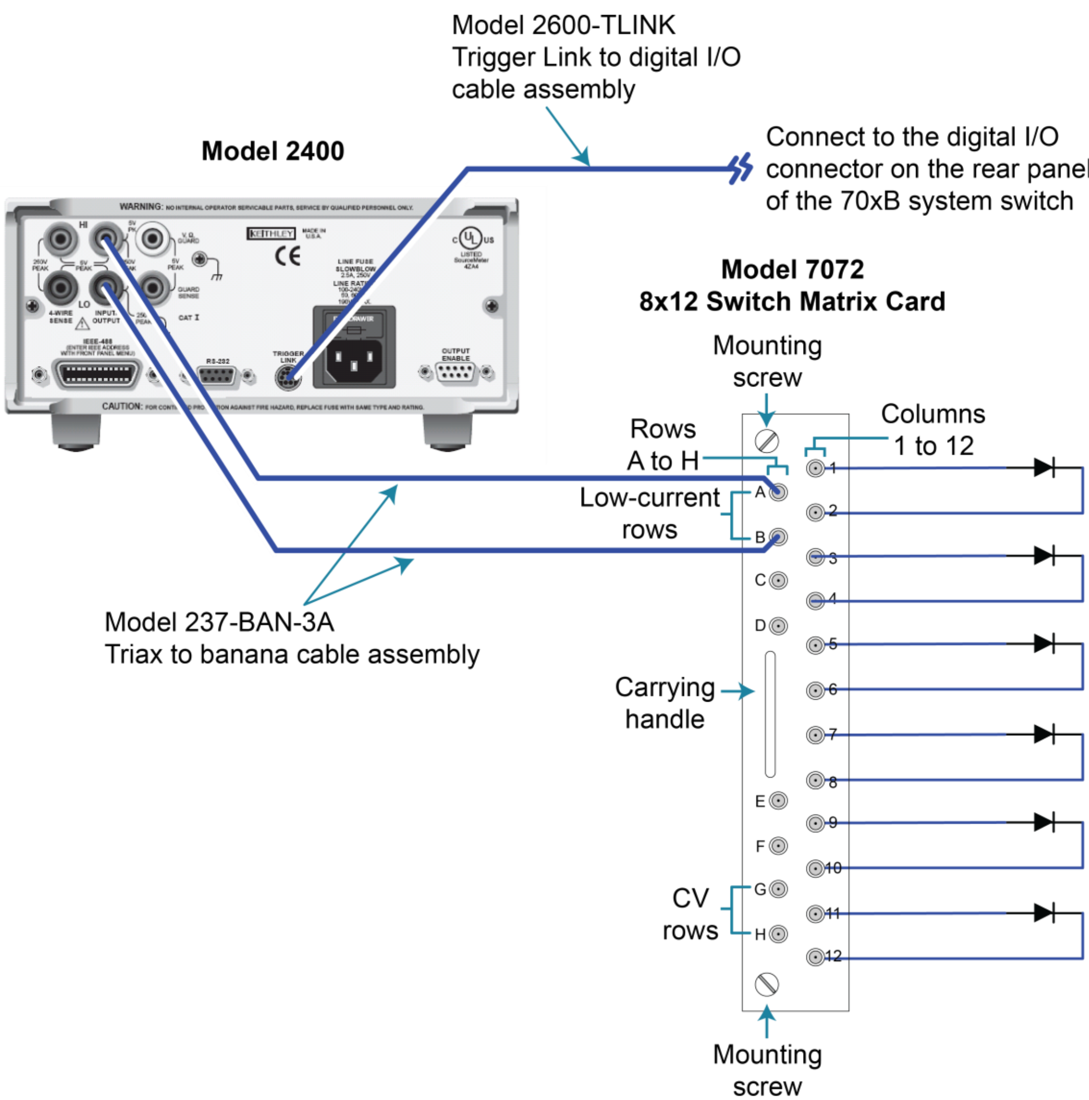
- Six diodes with triaxial connections
- One Model 707B or 708B
- One switch matrix card (Model 7174A, 7072, or 7072-HV)
- One Model 2400 SourceMeter instrument or equivalent source measurement unit (SMU)
- Example 2 only: One Model 2600-TLINK trigger link to digital I/O cable assembly
- Two Model 237-BAN-3A banana jack to triaxial cable assemblies

Connect the equipment

To run the test, make the connections as shown in the following figure. All triaxial connections for the SMU and the diodes are made to the center conductor (pin) of the triaxial connector.

NOTE

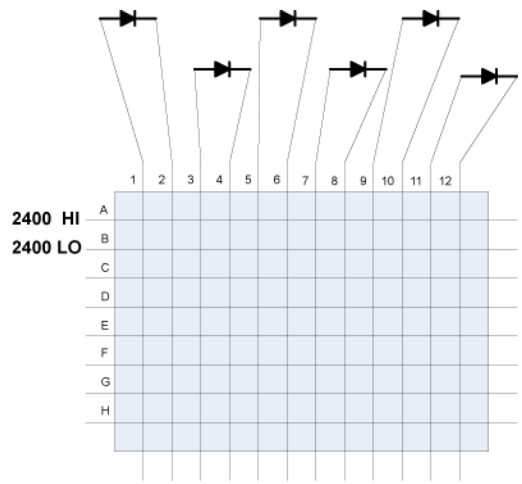
This figure shows connections to the Model 7072 8×12 Switch Matrix Card. You can make similar connections to the other matrix cards listed for this example.



Logical block diagram of test connections

The following figure shows the logical block diagram of the test connections.

Figure 91: Test system block diagram



Example 1: Individual commands for switch-measure

In this example, individual commands are sent to the Model 707B or 708B and SMU to perform switch closure and measure actions.

Example program code for the Model 2400

NOTE

TSB and TSB Embedded cannot be used to control the Model 2400 SourceMeter instrument. You are expected to send commands to the Model 2400 using a separate application development environment, such as Microsoft™ Visual Studio™ or NI™ LabVIEW™.

Send the following commands to configure the 2400.

Example program code for the Model 2400 SourceMeter Instrument	
Code	**Notes and comments**

`*RST`
`*CLS`
`:FORM:ELEM VOLT, CURR`
	Reset the instrument.
	Clear all errors and reset status bits.
	Format return data string to return current and voltage values.

`:SOUR:FUNC CURR"`
`:SOUR:CURR:MODE SWE`
`:SOUR:CURR:STAR 0`
`:SOUR:CURR:STOP 10E-3`
`:SOUR:CURR:STEP 1e-3`
`:SOUR:SWE:SPAC LIN`
	Configure current source to perform a linear sweep from 0 to 10 mA in 1 mA steps.

`:SENS:VOLT:PROT 10`
`:SENS:FUNC 'VOLT'`
`:SENS:VOLT:RANG 5`
`:SENS:VOLT:NPLC 1`
	Configure voltage limit.
	Configure SMU to measure voltage.
	Set a fixed voltage range of 5 V.
	Set integration rate to 1 PLC.

`:ARM:COUN 1` `:TRIG:COUN 11`	Complete each sweep once. Number of steps in sweep = 11.

Example program code for the Model 707B or 708B

Send the following commands to configure the Model 707B or 708B.

Example program code for the Model 707B or 708B	
Code	**Notes and comments**
`reset` `errorqueue.clear()`	Reset the instrument to factory defaults. Clear all errors.

Example code to run the test

To run the test, send the following commands to the corresponding instrument.

NOTE

After each `:READ?` query to the Model 2400, you must retrieve the response from the instrument. If you do not retrieve the response, the Model 2400 generates an error.

Example program code to run the test		
Model	**Code**	**Notes and comments**
2400	`:OUTP:STAT ON`	Turn on the output of the SMU.
707B or 708B	`channel.exclusiveclose("1A01,1B02")`	Connect first diode.
2400	`:READ?`	Initiate sweep and return data.
707B or 708B	`channel.exclusiveclose("1A03,1B04")`	Connect second diode.
2400	`:READ?`	Initiate sweep and return data.
707B or 708B	`channel.exclusiveclose("1A05,1B06")`	Connect third diode.
2400	`:READ?`	Initiate sweep and return data.
707B or 708B	`channel.exclusiveclose("1A07,1B08")`	Connect fourth diode.
2400	`:READ?`	Initiate sweep and return data.
707B or 708B	`channel.exclusiveclose("1A09,1B10")`	Connect fifth diode.
2400	`:READ?`	Initiate sweep and return data.
707B or 708B	`channel.exclusiveclose("1A11,1B12")`	Connect sixth diode.
2400	`:READ?`	Initiate sweep and return data.
2400	`:OUTP:STAT OFF`	Turn off the output of the SMU.
707B or 708B	`channel.open("allslots")`	Open all crosspoints.

Example 2: Digital I/O triggering for switch-measure synchronization

In this example, you configure both the Model 707B or 708B and the 2400 to wait for and send digital trigger signals after each instrument performs the requested switch closure or measure action.

Example program code for the Model 2400

NOTE

TSB and TSB Embedded cannot be used to control the Model 2400 SourceMeter. You are expected to send commands to the Model 2400 using a separate application development environment, such as Microsoft Visual Studio or NI LabVIEW™.

Send the following commands to configure the Model 2400 SourceMeter Instrument.

Example program code for the Model 2400 SourceMeter Instrument	
Code	**Notes and comments**

`*RST`
`*CLS`
`:FORM:ELEM VOLT, CURR`
	Reset the instrument.
	Clear all errors and reset status bits.
	Format return data string to return current and voltage values.

`:SOUR:FUNC CURR`
`:SOUR:CURR:MODE SWE`
`:SOUR:CURR:STAR 0`
`:SOUR:CURR:STOP 10E-3`
`:SOUR:CURR:STEP 1e-3`
`:SOUR:SWE:SPAC LIN`
	Configure current source to perform a linear sweep from 0 to 10 mA in 1 mA steps.

`:SENS:VOLT:PROT 10`
`:SENS:FUNC 'VOLT'`
`:SENS:VOLT:RANG 5`
`:SENS:VOLT:NPLC 1`
	Configure voltage limit.
	Configure the SMU to measure voltage.
	Set a fixed voltage range of 5 V.
	Set integration rate to 1 PLC.

`:ARM:DIR ACC`
`:ARM:SOUR TLINK`
`:ARM:ILIN 1`
`:ARM:OLIN 2`
`:ARM:OUTP TEXIT`
`:TRIG:DIR ACC`
`:TRIG:SOUR IMM`
`:TRIG:COUN 11`
`:ARM:COUN 6`
	Configure the SMU to trigger entire sweep on receipt of hardware trigger on the Trigger Link connector.
	Configure the SMU to receive a trigger on line 1 and to generate a trigger on line 2.
	
	Configure the counts of the trigger model so that the 11-point sweep is performed six times (once for each diode).

`:TRAC:POIN 66`
`:TRAC:FEED SENS`
`:TRAC:FEED:CONT NEXT`
	Configure the nonvolatile data buffer of the SMU to store all 66 sets of readings generated for sweeps across all diodes.

Example program code for the Model 707B or 708B

The following code loads a script on the 70xB that contains two functions that:

- Set up the table of crosspoints to close
- Execute the test

Example program code for the Model 707B or 708B	
Code	**Notes and comments**
`loadscript forScan`	This script creates two functions on the Model 707B or 708B to configure and run the test. This script should be stored in the Model 707B or 708B. Remember that `loadscript` and `endscript` commands are not necessary if this script is created in the instrument using TSB or TSB Embedded.

<table>
<tr><th colspan="2">Example program code for the Model 707B or 708B</th></tr>
<tr><td><pre>function setup70xChanList()

   reset()
   errorqueue.clear()

   chanTable = {}
   chanTable[1] = "1A01,1B02"
   chanTable[2] = "1A03,1B05"
   chanTable[3] = "1A05,1B06"
   chanTable[4] = "1A07,1B08"
   chanTable[5] = "1A09,1B10"
   chanTable[6] = "1A11,1B12"

   digio.trigger[1].mode = digio.TRIG_FALLING
   digio.trigger[2].mode = digio.TRIG_FALLING

   digio.trigger[1].clear()
   digio.trigger[2].clear()

   return chanTable

end</pre></td><td>Create a function that sets up the Model 707B or 708B and the list of channels to be used in the test.
Reset the instrument and clear errors.

Define and populate a table that holds the crosspoints to be used to connect the SMU to the diodes under test (also see notes below this example).

Configure the digital I/O trigger lines to respond to and generate falling edge triggers.

Clear any latched triggers.

Return the table as an output of the function</td></tr>
<tr><td><pre>function run70xForScan(cTable)

   for i = 1, table.getn(cTable) do
      if i > 1 then
         triggered = digio.trigger[2].wait(3)
         if triggered == true then
            channel.exclusiveclose(cTable[i])
            waitcomplete()
            digio.trigger[1].assert()
         else
            print("Trigger not received")
         end
      else
         channel.exclusiveclose(cTable[i])
         waitcomplete()
         digio.trigger[1].assert()
      end
   end
   print("done")
end</pre></td><td>This function sequences the crosspoint closures and hardware triggering. A table of crosspoints to close must be input to this function.

The <code>for</code> loop iterates through the channel list and waits for a trigger on a digital I/O line to occur before closing a channel.

The first channel of the list does not wait for a trigger.

After channel closure is complete, an output trigger is asserted on digital I/O line.</td></tr>
<tr><td><code>endscript</code></td><td></td></tr>
<tr><td><code>forScan()</code></td><td>Run the script to generate the functions.</td></tr>
<tr><td><code>crossptTable = setup70xChanList()</code></td><td>Call the function that sets up the Model 707B or 708B and stores the output of the function in the variable <code>crossptTable</code>.</td></tr>
</table>

Alternate method to create a list of crosspoints

The following code creates a list of crosspoints that can be used in a test. This code replaces `chanTable` in the previous example code.

```
x = 1
-- The following nested for loop creates channel list for the scan.
-- This list is compiled in the table "jscTable".
for i = 1,11,2 do
   j = i + 1
   chan1 = channel.createspecifier(1,1,i)
   chan2 = channel.createspecifier(1,2,j)
   chanTable[x] = chan1 .. "," .. chan2
   -- Crosspoints are printed to show output of example.
   print(chanTable[x])
x = x+1
end

Output:
1A01,1B02
1A03,1B05
1A05,1B06
1A07,1B08
1A09,1B10
1A11,1B12
```

Run the test

The Model 707B or 708B closes the crosspoints necessary to connect the SMU to each diode under test one-at-a-time. The SMU then performs an I-V sweep on each diode. This process continues until all six diodes are tested.

Example code to run the test

To run the test, send the following commands to the corresponding instrument.

Example program code to run the test		
Model	**Code**	**Notes and comments**
2400	`:OUTP ON`	Turn on the output of the SMU.
2400	`:INIT`	Initiate the sweep. After processing this command, the Model 2400 waits for a trigger from the Model 707B or 708B.
707B or 708B	`run70xForScan(crossptTable)`	Call the function that executes the sequence of crosspoints and generates hardware trigger signals to trigger sweeps on the SMU.
2400	`:TRAC:DATA?`	After the test is complete, send this command to retrieve the data from the Model 2400. Retrieve the data from the Model 2400 after sending `:TRAC:DATA?`.

Section 11

Troubleshooting guide

In this section:

Introduction

This section provides information to help you troubleshoot problems with your instrument.

Error and status messages

This section includes information on error levels and how to read errors.

Error and status messages are assigned a level of severity, as listed in the following table.

Severity level descriptions		
Number	**Level**	**Description**
0	Informational	Indicates that there are no entries in the queue
10	Informational	Indicates a status message or minor error
20	Recoverable	Indicates possible invalid user input; operation continues but operator should take action to correct the error
30	Serious	Indicates a serious error that may require technical assistance, such as corrupted data
40	Fatal	Instrument is not operational

Effects of errors on scripts

Most errors do not abort a running script. The only time a script is aborted is when a Lua runtime error (error number −286) is detected. Runtime errors are caused by actions such as trying to index into a variable that is not a table.

Syntax errors (error number −285) in a script or command do not abort the script, but do prevent the script or command from being executed.

Retrieve errors

When errors occur, the error messages are placed in the error queue. Use `errorqueue` commands to request error message information. For example, the following commands request the complete set of information about the next message in the error queue. They return the code, message, severity, and node for that error:

```
errorCode, message, severity, errorNode = errorqueue.next()
print(errorCode, message, severity, errorNode)
```

The following table lists the commands associated with the error queue.

Remote commands associated with the error queue

Command	Description
errorqueue.clear() (on page 7-87)	Clear error queue of all errors
errorqueue.count (on page 7-87)	Number of messages in the error queue
errorqueue.next() (on page 7-88)	Request next error message from queue

Identify that an error occurred

If you are using TSB Embedded, error messages are displayed in the Instrument Output box when they occur.

If you are using another remote interface, you might need to use commands to retrieve the error messages. You can use the commands errorqueue.count (on page 7-87) and errorqueue.next() (on page 7-88) to retrieve the number of messages and the text of the messages.

To set the instrument to automatically send generated errors, set localnode.showerrors (on page 7-133) to 1 (enabled).

To set the instrument to automatically send prompts after each command message, set localnode.prompts (on page 7-129) to 1 (enabled).

Troubleshoot USB communications

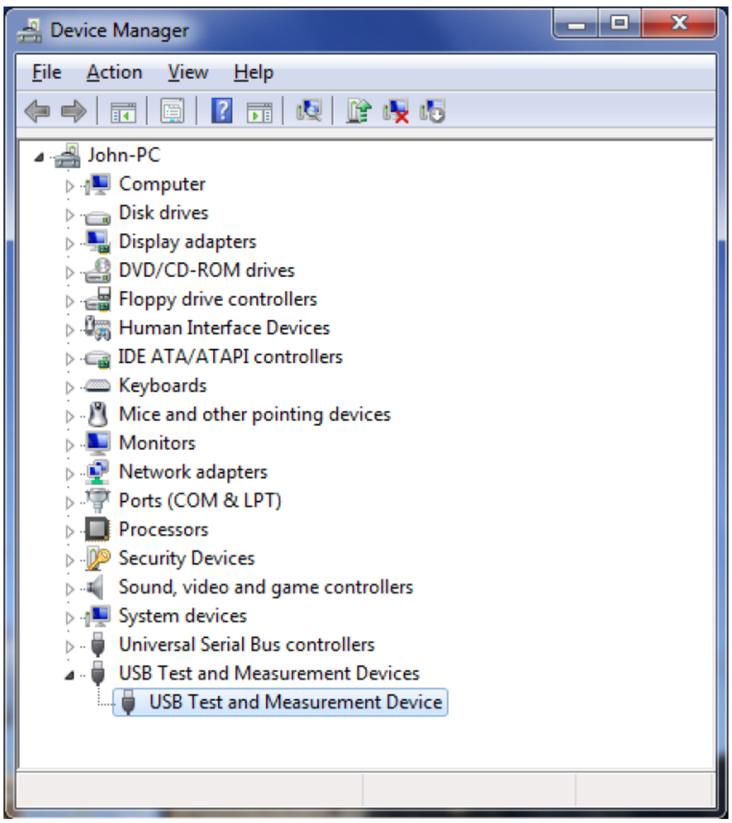
This section provides information on checks you can perform if USB communications with the instrument are not working.

Check driver for the USB Test and Measurement Device

1. Open Device Manager. From the **Start** menu, you can enter `devmgmt.msc` in the Run box or the Windows search box to start Device Manager.
2. Under USB Test and Measurement Devices, look for USB Test and Measurement Device.

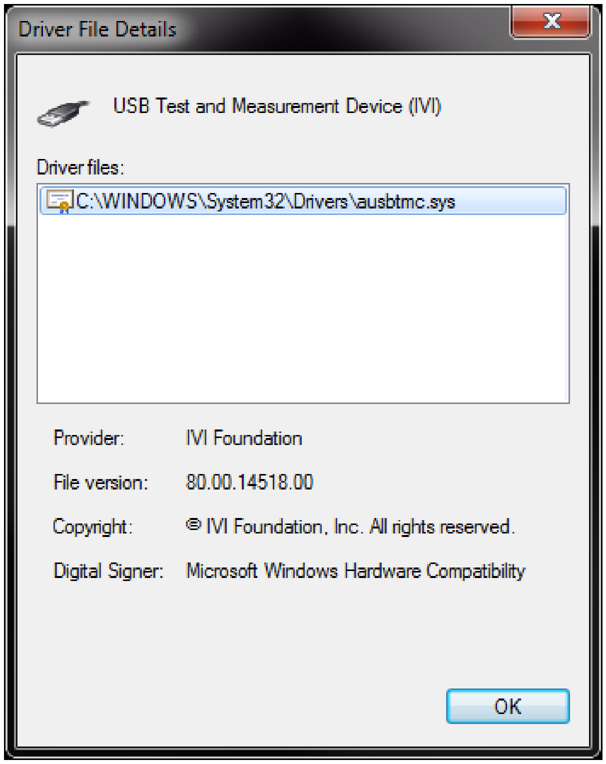
   If the device is not there, either VISA is not installed or the instrument is not plugged in and switched on.

Figure 92: Device Manager dialog showing USB Test and Measurement Device



3. Right-click the device.
4. Select **Properties**.
5. Select the **Driver** tab.
6. Select **Driver Details**.
7. Verify that the device driver is the `ausbtmc.sys` driver from IVI Foundation.

Figure 93: Driver File Details dialog



8. If the incorrect driver is installed, select **OK**.
9. If this does not work, uninstall VISA, unplug the instrument, and follow the steps to reinstall VISA in the section Modify, repair, or remove Keithley I/O Layer software.

Troubleshoot GPIB interfaces

Controller (hardware) not recognized

If the hardware is not recognized by the computer:

1. Check the website of the vendor for newer drivers.
2. Check that the drivers are valid for the operating system you have and any updates that might be necessary. This information is typically found in the readme file that comes with the drivers.
3. Follow the instructions from the vendor on updating drivers.

If the hardware is still not recognized, you can try a different computer using a different operating system to rule out operating system issues.

If this does not resolve the issue, contact the vendor of the GPIB controller for assistance.

Timeout errors

If your GPIB controller is recognized by the operating system, but you get a timeout error when you try to communicate with the instrument, check the following:

1. Confirm that the GPIB address you assigned to the instrument is unique and between 0 to 30. Do not use 0 or 21 because they are common controller addresses.
2. Check cabling connections. GPIB cables are heavy and can fall out of the connectors if they are not screwed in securely.
3. Substitute cables to verify cable integrity. For example, if you can send and receive ASCII text, but you cannot do a binary transfer, check your program and the decoding of the binary data. If that does not resolve the problem, try another cable. ASCII text only uses seven data lines in the cable; the binary transfer requires all eight lines.

Troubleshoot LAN interfaces

This section provides information on troubleshooting LAN interfaces.

For detailed information on setting up remote interfaces see Remote communications interfaces (on page 4-1).

Verify connections and settings

If you are unable to connect to the internal web page of the instrument, check the following items:

- Verify that the crossover cable is in the correct LAN port on the instrument. Do not connect to one of the TSP-Link ports.
- Verify that the crossover cable is in the correct port on the computer. The ethernet port of a laptop may be disabled while the computer is in a docking station.
- Verify that the correct ethernet card configuration information was used during the setup procedure.
- Verify that the network card in the computer is enabled.
- Verify that the instrument IP address is compatible with the IP address on the computer.
- Verify that the subnet mask address of the instrument is the same as the subnet mask address of the computer.
- Turn the instrument power off, and then on.
- Reboot the computer.

Use ping to test the connection

Ping is a computer network administration utility that you can use to test whether a particular host can be reached across an Internet Protocol (IP) network. It also measures the round-trip time for packets sent from the local host to a destination computer, including the interfaces of the local host.

To run ping:

1. From the Windows Start menu, type `cmd` in the Run box or Search box. The Command Prompt window is displayed.
2. At the > prompt, type `ping` followed by the IP address. For example:

```
ping 169.254.52.51
```

Beware that some network devices, especially LXI instruments, can disable the ping response to prevent denial of service attacks. This prevents hackers from pinging your instrument indefinitely, which causes the instrument to become so busy it cannot respond to a web browser or instrument driver.

If you cannot ping an instrument from the computer, you cannot communicate with the instrument. You need to check the LAN settings from the front panel of the instrument to see if they match the configuration of your network.

If you can ping your instrument, you can open the web page in the instrument from a browser by typing the IP address in the address (URL) field.

Open ports on firewalls

A firewall is a part of a computer system or network that is designed to block unauthorized access while permitting authorized communications. It is a device or set of devices that are configured to permit or deny applications based on a set of rules and other criteria.

If you have a firewall in the network between your computer and the instrument, you need to make sure the following ports are opened for UDP and TCP packets:

- **Port 80:** Web server. This is normally open.
- **Port 1024:** VXI-11 connection for sending and receiving commands from the instrument.
- **Port 5025:** Raw socket connection for sending and receiving commands from the instrument.

Web page problems

All LXI instruments have a web server. The LAN configuration information on these pages is mandated by the LXI consortium. For Keithley instruments that are compliant with LXI, the standard LXI pages use standard HTML.

The added value pages that Keithley has added to control the instrument use Java. If Java is not installed when you select one of these instrument-specific web pages, the web page prompts you to install it. To do this, your computer must have access to the internet so it can access the web browser plug-in Sun Java Runtime Environment Version 6 or higher. If you do not have an internet connection, you must download it elsewhere and install it on the computer that it connected to the instrument. Installation files are available at the Java download site (java.com/download/manual.jsp).

When you connect to the instrument web page for the first time, several things can happen:

- If the security settings are high, scripting might be disabled and the browser will prompt you to enable ActiveX and scripting. To resolve the problem, enable ActiveX and scripting.
- When the Java applet from the instrument gets downloaded into the browser it will ask you if you trust this active content from Keithley. Select **Yes**.

If you have resolved the problems, the instrument control pages should work and if you try to perform an action, such as closing a relay, you are prompted for the password (the default is `admin`).

NOTE

If you update the firmware for the instrument using the web page (not available for all instruments), you need to flush the browser cache so that a fresh Java Applet gets downloaded the next time you access the web page.

LXI LAN status indicator

Most LAN network interface cards have two LEDs, one that indicates LAN traffic and one that designates the LAN speed (10 megabits, 100 megabits, 1 gigabits) through the color of the LED. LXI also requires a LAN status indicator. This can be an LED or an indicator on a display. It shows if the instrument has a valid IP address or is in a fault state.

When diagnosing a LAN connection issue with an LXI instrument, see if the LAN status indicator is signaling a valid or fault condition. If there is an error, you cannot communicate with the instrument through the LAN connection. In this case, you need to check the LAN parameter settings from the front panel of the instrument. Make sure if you change a LAN setting through the front panel that you select **Apply LAN Settings**.

Initialize the LAN configuration

If you cannot communicate with your instrument using LAN, perform this reset. When you perform a reset, the instrument is returned to DHCP and Auto-IP enabled. If you set your computer to match, you should be able to use a discovery tool to determine the IP address and communicate with the instrument again. Also check the LXI LAN status indicator to verify that there are no faults.

To reset the 707B or 708B, from the front panel, select **MENU**, then select **LAN > RESET**.

Use the LXI Discovery Tool

To find the IP address of the Models 707B and 708B from a computer, use the LXI Discovery Tool, a utility that is available from the Resources tab of the website for the LXI Consortium (lxistandard.org).

Communicate using VISA communicator

There are several interactive communications utilities that you can use to communicate with LAN instruments:

- The Keithley I/O layer installs the Keithley Communicator.
- NI-VISA (full version) installs the NI-VISA Interactive Control utility, which can also be launched from NI-MAX.
- Keysight has a similar utility called Interactive IO that gets installed with their IO Libraries Suite.

All these utilities require you to enter the VISA resource string for your instrument. See Communicate with the instrument (on page 4-30) for more information on the VISA resource string formats.

WireShark

WireShark is an open source LAN packet sniffer. You can run it to spy on all the packets going across a network. It allows you to filter what you spy on so that you can narrow the content down to just what you are interested in. For example, you could check only web page packets (http) or all packets being sent by a device on a certain IP address.

Refer to the WireShark documentation for information. WireShark can be downloaded from www.wireshark.org (wireshark.org).

Test the display, keys, and channel matrix

You can test operation of the keys, display, and crosspoint display (707B only) from the front panel of the instrument.

Verify front-panel key operation

You can verify that the instrument is properly reading front-panel key presses.

To verify key operation:

1. From the front panel, select **MAIN MENU > DISPLAY > TEST > KEYS**. The message "No keys pressed" is displayed.
2. Press a key. The name of the key is displayed. For a list of key values, see display.sendkey() (on page 7-82).
3. Press **EXIT (LOCAL)** twice to return to the menu.

Verify display operation

You can verify that all the pixels on the vacuum fluorescent display (VFD) are working.

To verify VFD operation:

1. From the front panel, select **MAIN MENU > DISPLAY > TEST > DISPLAY-PATTERNS**. A pattern is displayed.
2. Press the navigation wheel to display the next pattern.
3. When you have viewed the patterns, press **EXIT** to return to the menu.

Verify crosspoint display operation (707B only)

You can verify that the LEDs and displays on the crosspoint display are working properly.

To verify crosspoint display operation:

1. From the front panel, select **MAIN MENU > DISPLAY > TEST > LED-PATTERNS**. The `ALPH NUMERIC COL LEDS` is displayed.
2. Press the navigation wheel ◷ to display first test. The name of the test is displayed on the bottom display.
3. After each test, press the navigation wheel ◷ to move to the next test.
4. On the last test (STEP COL GRIP LED WHEEL TO DIAL), use the navigation wheel ◷ to check the matrix LEDs and the slot LEDs.
5. When you have verified the patterns, press **EXIT** to return to the menu.

Update drivers

For the latest drivers and additional support information, see tek.com/support.

To see what drivers are available for your instrument:

1. Go to the Product Support and Downloads web page (tek.com/support/product-support).
2. Enter the model number of your instrument.
3. Select **Software** from the list.
4. Select and download the file.

For LabVIEW™, you can also go to the NI website and search their instrument driver database.

Support contact information

If you have any questions after reviewing this information, please contact your local Tektronix office, sales partner, or distributor. You can also call the Tektronix corporate headquarters (toll-free inside the U.S. and Canada only) at 1-800-833-9200. For worldwide contact numbers, visit tek.com/contact-tek.

When contacting Keithley, have the following information ready:

- The serial number of the instrument.
- The firmware revision of the instrument.
- The model and firmware revision of all installed cards.

When you call, have the information available, and, if possible, be near the instrument.

Section 12

Status model

In this section:

Overview

Each Keithley 707B and 708B provides status registers and queues that are collectively referred to as the status model. Through manipulation and monitoring of these registers and queues, you can view and control various instrument events. You can include commands in your test program that can determine if a service request (SRQ) event has occurred and the cause of the event.

The heart of the status model is the Status Byte Register. All status model registers and queues flow into the Status Byte Register.

The entire status model is shown in Status model diagrams (on page 12-3).

Status register set contents

Typically, a status register set contains the following registers:

- **Condition** (`.condition`): A read-only register that is constantly updated to reflect the present operating conditions of the instrument.
- **Enable Register** (`.enable`): A read-write register that allows a summary bit to be set when an enabled event occurs.
- **Event Register** (`.event`): A read-only register that sets a bit to 1 when the applicable event occurs. If the enable register bit for that event is also set, the summary bit of the register is set to 1.

- **Negative Transition Register (NTR)** (`.ntr`): When a bit is set in this read-write register, it enables a 1 to 0 change in the corresponding bit of the condition register to cause the corresponding bit in the event register to be set.
- **Positive Transition Register (PTR)** (`.ptr`): When a bit is set in this read-write register, it enables a 0 to 1 change in the corresponding bit of the condition register to cause the corresponding bit in the event register to be set.

An event is represented by a condition register bit changing from a 1 to 0 or 0 to 1. When an event occurs and the appropriate NTR or PTR bit is set, the corresponding event register bit is set to 1. The event bit remains latched to 1 until the event register is read or the status model is reset. When an event register bit is set and its corresponding enable bit is set, the summary bit of the register is set to 1. This, in turn, sets a bit in a higher-level condition register, potentially cascading to the associated summary bit of the Status Byte Register.

Summary bit

The summary bit of each register is either set (1) or clear (0). A set summary bit indicates that one (or more) of the enabled events in that register has occurred.

Queues

The Models 707B and 708B uses queues to store messages. The queues include:

- **Command queue:** Holds commands that are available for execution.
- **Output queue:** Holds response messages.
- **Error queue:** Holds error and status messages.

When a queue contains data, it sets the condition bit for that queue in one of the registers. The condition bits are:

- **Command queue:** CAV in the Operation Status Remote Summary Register.
- **Output queue:** MAV in the Status Byte Register.
- **Error queue:** EAV in the Status Byte Register.

The CAV, MAV, and EAV bits in the registers are cleared when the queue is empty. Queues empty when:

- Commands are executed.
- Errors are read from the error queue.
- Response messages are read from the instrument.

All Models 707B and 708B queues are first-in, first-out (FIFO).

The Status Byte Register overview (on page 12-4) shows how the queues are structured with the other registers.

Output queue

When the instrument is in the remote state, the output queue holds data that pertains to the normal operation of the instrument. For example, when a `print()` command is sent, the response message is placed in the output queue.

When data is placed in the output queue, the Message Available (MAV) bit in the status byte register is set. A response message is cleared from the output queue when it is read. The output queue is considered cleared when it is empty. An empty output queue clears the MAV bit in the status byte register.

A message is read from the output queue by addressing the instrument to talk.

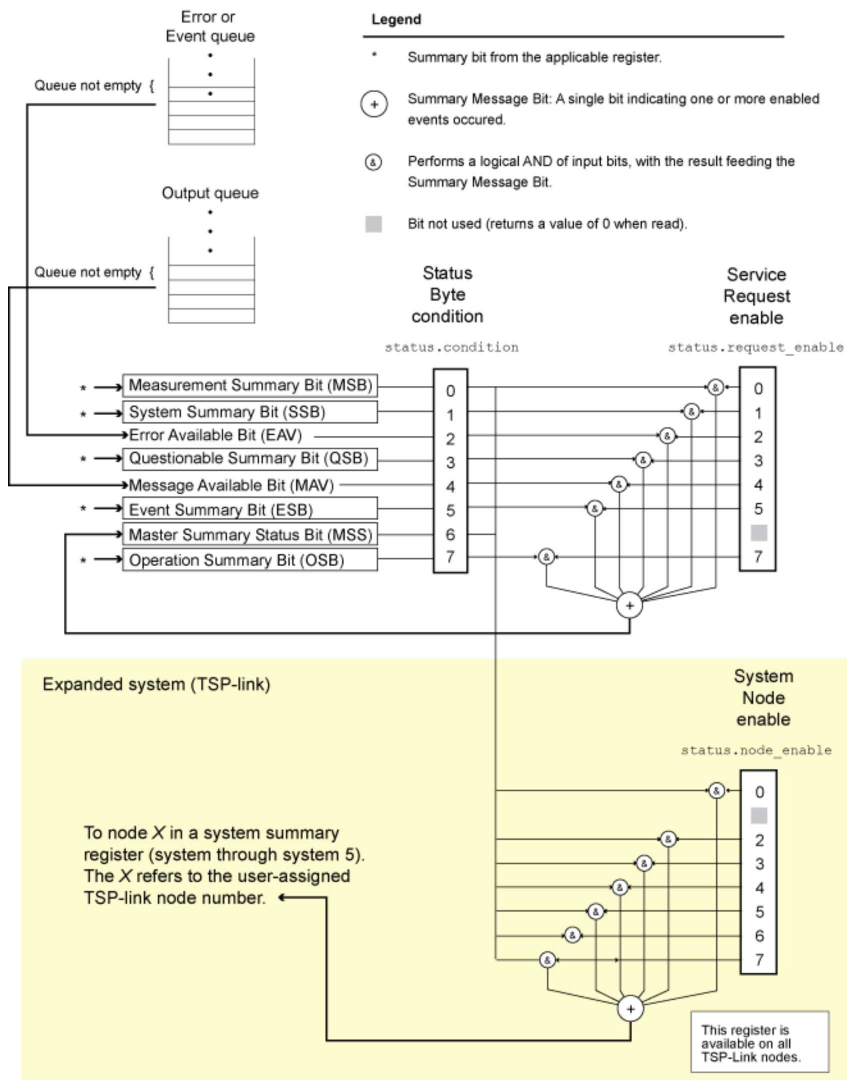
Status model diagrams

The register sets (and queues) monitor various instrument events. When an enabled event occurs in one of the five registers, it sets the associated summary bit in the Status Byte register. When a summary bit of the Status Byte is set and its corresponding enable bit is set (as programmed using status.request_enable (on page 7-188)), the MSS bit will set to indicate that an SRQ has occurred. View the master summary bit using status.condition (on page 7-175). In an expanded system (TSP-Link), setting the status.node_enable (on page 7-177) attribute allows the System registers to be shared by all nodes in the TSP-Link system. The following figures and topics describe the relationships of the individual registers and queues with the Status Byte register.

Status Byte Register diagram

The Status Byte Register receives summary bits from the other status register sets and queues, and also from itself (which sets the Master Summary Status, or MSS, bit).

Figure 94: Status Byte Register



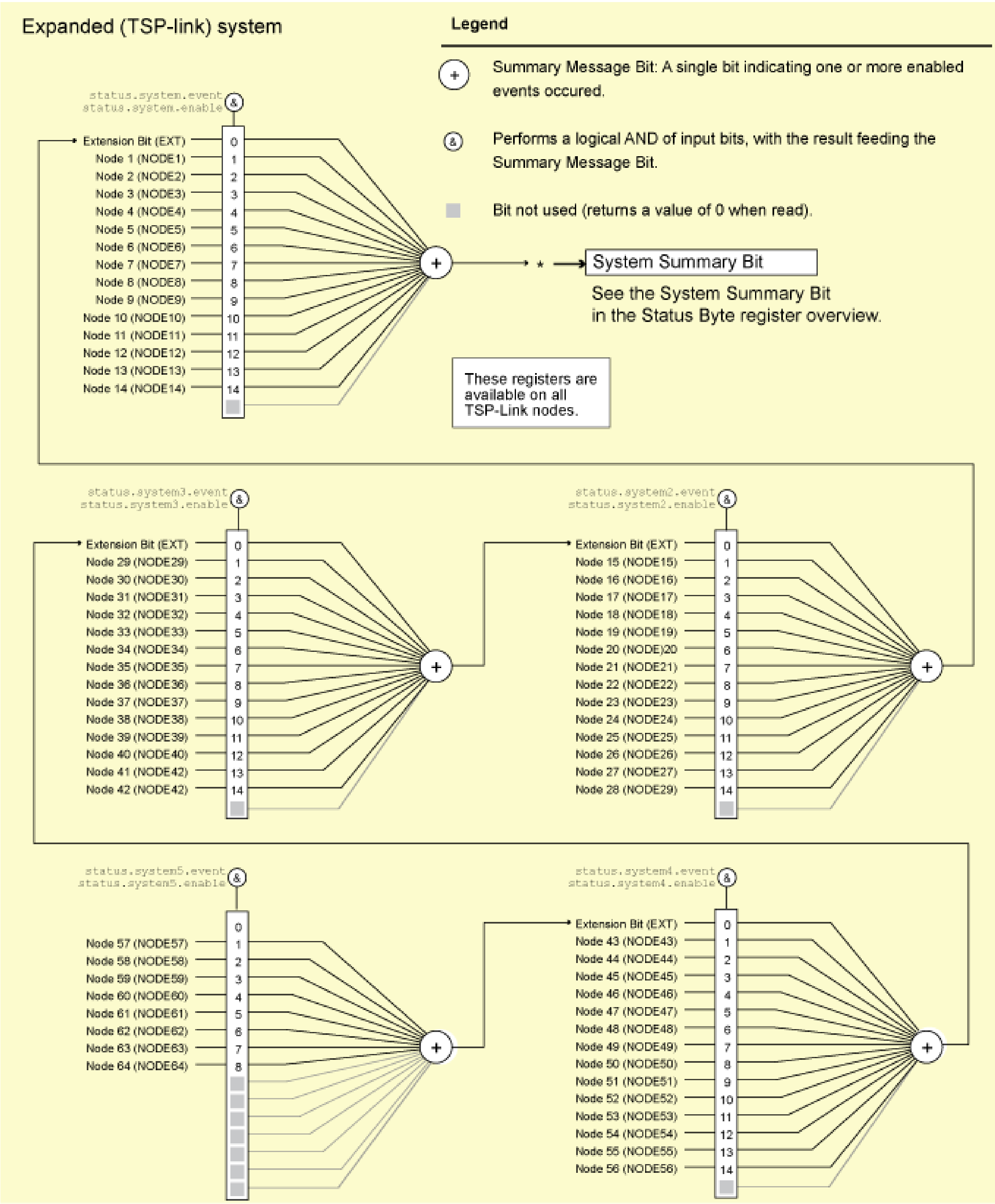
Measurement summary bit (Measurement event register)

The summary bit of the measurement event register provides enabled summary information to Bit B0 (MSB) of the status byte. Since the 707B and 708B has no internal measurement capabilities, none of the bits in the measurement registers are defined. Therefore, this bit is always 0.

System summary bit (System register)

The summary bit of the system register provides enabled summary information to Bit B1 (SSB) of the status byte.

Figure 95: System summary bit (System Register)



As shown above, there are five register sets associated with System Event Status. These registers summarize system status for the nodes connected to TSP-Link. All nodes on TSP-Link share a copy of the system summary registers once TSP-Link has been initialized. This feature allows all nodes to access the status models of other nodes, including SRQ.

In a TSP-Link system, the status model can be configured such that a status event in any node in the system can set the RQS (Request for Service) bit of the Master Node Status Byte. See TSP-Link system status (on page 12-20) for details on using the status model in a TSP-Link system.

Attributes are summarized in status.system.* (on page 7-195), status.system2.* (on page 7-197), status.system3.* (on page 7-199), status.system4.* (on page 7-201), and status.system5.* (on page 7-203).

For example, any of the following commands set the EXT enable bit:

```
status.system.enable = status.system.EXT
status.system.enable = status.system.EXTENSION_BIT
status.system.enable = 1
```

When reading a register, a numeric value is returned. The binary equivalent of this value indicates which bits in the register are set. For details, see Read registers (on page 12-15). For example, the following command will read the system enable register:

```
print(status.system.enable)
```

The bits used in the system register sets are described as follows:

- **Bit B0, Extension Bit (EXT):** Set bit indicates that an extension bit from another system status register is set.
- **Bits B1-B14* NODE*N*:** Indicates a bit on TSP-Link node n has been set (*N* = 1 to 64).
- **Bits B15:** Not used.

*`status.system5` does not use bits B9 through B15.

Refer to the following table for available *N* values.

Command	*N* value
`status.system.*`	1 to 14
`status.system2.*`	15 to 28
`status.system3.*`	29 to 42
`status.system4.*`	43 to 56
`status.system5.*`	57 to 64

Error available bit (Error or Event queue)

The summary bit of the error or event queue provides enabled summary information to Bit B2, Error Available (EAV) bit, of the status byte.

The EAV bit is set when a message defining an error or status is placed in the error or event queue. The error or event queue is one of the two Model 707B or 708B queues associated with the status model. The other queue sets the Message Available (MAV) bit (output queue). Both queues are first-in, first-out (FIFO) queues. The error queue holds error and status messages. The status model shows how these queues are structured with regard to the other registers.

The following sequence outlines typical events associated with this queue:

1. When an error or status event occurs, a message defining the error (or status) is placed in the error queue.
2. The EAV bit in the Status Byte Register is set.
3. Through programming, the error (or status) message is read. This clears the error (or status) from the error queue. The error queue is considered cleared when it is empty.
4. An empty error queue clears the EAV bit in the Status Byte Register.

The commands to control the error queue are listed in the following table. When you read a single message in the error queue, the oldest message is read and then removed from the queue. On power-up, the error queue is initially empty. If there are problems detected during power-on, entries will be placed in the queue. If no problems are detected, the error number `0` and `No Error` is returned.

Error queue command	**Description**
`errorqueue.clear()`	Clear error queue of all errors.
`errorqueue.count`	Number of messages in the error/event queue.
`errorqueue.next()`	Request error message.

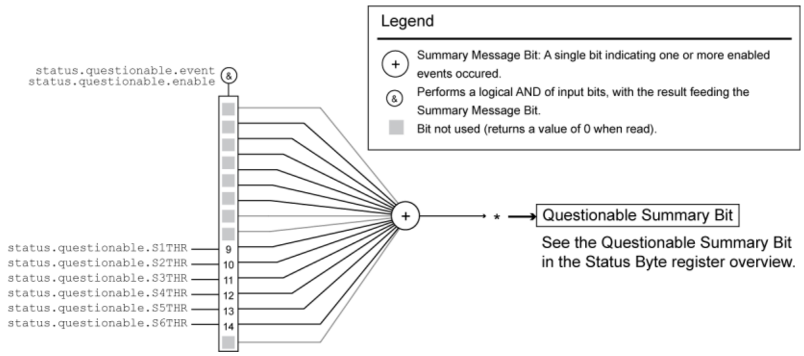
Messages in the error queue include a code number, message text, severity, and TSP-Link node number. For example, the following commands request the next complete error information from the error queue and display the code, message, severity, and node of the next error:

```
errorcode, message, severity, errornode = errorqueue.next()
print(errorcode, message, severity, errornode)
```

Questionable summary bit (Questionable event register)

The summary bit of the questionable event register provides enabled summary information to Bit B3 (QSB) of the status byte.

Figure 96: Questionable summary bit (Questionable Event Register)



As shown above, there is only one register set associated with the questionable status. Attributes are summarized in status.questionable.* (on page 7-186). Bit can also be set by using numeric parameter values. For details, see Program enable and transition registers (on page 12-14).

For example, any of the following statements set the thermal aspect enable bit of a card in slot 1:

```
status.questionable.enable = status.questionable.S1THR
status.questionable.enable = status.questionable.SLOT1_THERMAL
status.questionable.enable = 512
```

The following command requests the questionable enable register value in numeric form:

```
print(status.questionable.enable)
```

The bit used in this register set is `SxTHR`. A set bit indicates the thermal aspect of the card in slot x is in question, where x = 1 to 6.

Message available bit (Output queue)

The summary bit of the output queue provides enabled summary information to Bit B4, Message Available (MAV), of the status byte.

The MAV bit is set when the Output queue holds data that pertains to the normal operation of the instrument. The Output queue is one of the two Model 707B or 708B queues associated with the status model. The other queue sets the Error Available Bit (Error or Event queue) (on page 12-7). Both queues are first-in, first-out (FIFO) queues. The Status Byte Register overview (on page 12-4) shows how these queues are structured with regard to the other registers.

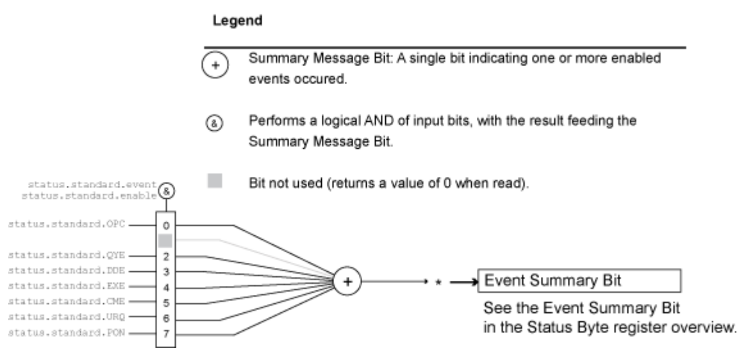
As an example, when a print command is sent, the response message is placed in the Output queue. When data is placed in the Output queue, the Message Available (MAV) bit in the Status Byte Register sets. A response message is cleared from the Output queue when it is read. The Output queue is considered cleared when it is empty. An empty Output queue clears the MAV bit in the Status Byte Register.

A message is read from the Output queue by addressing the Model 707B or 708B to talk.

Event summary bit (ESB register)

The summary bit of the Standard event register provides enabled summary information to Bit B5 (OSB) of the status byte.

Figure 97: Event summary bit (Standard Event Register)



As shown above, there is only one register set associated with the event status register. Attributes are summarized in status.standard.* (on page 7-192). Bits can also be set by using numeric parameter values. For details, see Program enable and transition registers (on page 12-14).

For example, any of the following statements set the operation complete enable bit:

```
standardRegister = status.standard.OPC
status.questionable.enable = status.standard.OPERATION_COMPLETE
status.questionable.enable = 1
```

The bits used in this register set are described as follows:

- **Bit B0, Operation Complete (OPC):** Set bit indicates that all pending selected device operations are completed and the instrument is ready to accept new commands. The bit is set in response to an `*OPC` command. The remote command `opc()` can be used in place of the `*OPC` command.
- **Bit B1:** Not used.
- **Bit B2, Query Error (QYE):** Set bit indicates that you attempted to read data from an empty Output queue.
- **Bit B3, Device-Dependent Error (DDE):** Set bit indicates that an instrument operation did not execute properly due to some internal condition.
- **Bit B4, Execution Error (EXE):** Set bit indicates that the instrument detected an error while trying to execute a command.
- **Bit B5, Command Error (CME):** Set bit indicates that a command error has occurred. Command errors include:
  - **IEEE-488.2 syntax error:** The instrument received a message that does not follow the defined syntax of the IEEE-488.2 standard.
  - **Semantic error:** instrument received a command that was misspelled or received an optional IEEE-488.2 command that is not implemented.
  - **GET error:** The instrument received a Group Execute Trigger (GET) inside a program message.
- **Bit B6, User Request (URQ):** Set bit indicates that the LOCAL key on the instrument front panel was pressed.
- **Bit B7, Power ON (PON):** Set bit indicates that the instrument has been turned off and turned back on since the last time this register has been read.

Master summary status bit (MSS bit register)

The master summary status bit provides summary information to Bit B6 (MSS) of the status byte. Although this bit is always enabled for the status byte, it has to be enabled (using `status.node_enable)` if needed in an expanded system (TSP-Link).

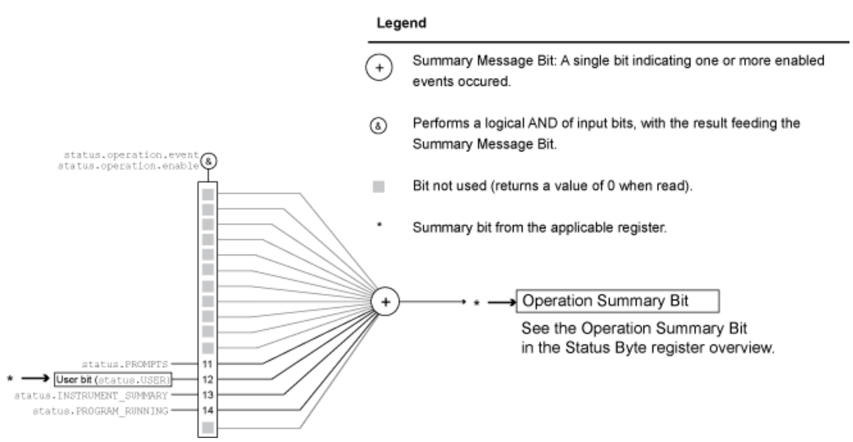
The Master Summary Status Bit (MSS) is set when an enabled summary bit of the Status Byte Register is set. This bit (B6) may also be interpreted as a Request Service (RQS) bit. Depending on how it is used, Bit B6 of the Status Byte Register is either the Request for Service (RQS) bit or the Master Summary Status (MSS) bit.

When using the GPIB serial poll sequence of the Switching System to obtain the status byte (serial poll byte), B6 is the RQS bit. See Serial polling and SRQ (on page 12-18) for details on using the serial poll sequence. For common and script commands (Status Byte Register), B6 is the MSS (Message Summary Status) bit. The serial poll, although automatically resetting the RQS bit, does not clear MSS. The MSS remains set until all Status Byte summary bits are reset.

Operation summary bit (Operation event register)

The summary bit of the operation event register provides enabled summary information to Bit B7 (OSB) of the status byte.

Figure 98: Operation summary bit (Operation Event Register)



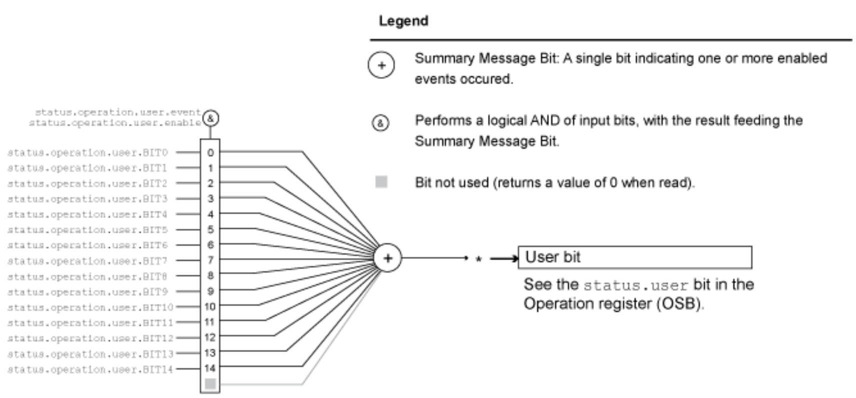
The bits used in this register set are described as follows:

- **Bits B1 to B10:** Not used.
- **Bit B11, Remote Summary (REM):** Set bit indicates that an enabled bit in the Operation Status Remote Summary Register is set.
- **Bit B12, User (USER):** Set bit indicates that an enabled bit in the operation status user register is set.
- **Bit B13, Instrument Summary (INST):** Set bit indicates that an enabled bit in the operation status instrument summary register is set.
- **Bit B14:** Not used.

Operation user bit (Operation User Register)

The summary bit of the Operation User Register provides the Bit B12 User bit to the Operation Status Register. In turn, the summary bit of the Operation Status Register provides the operation summary bit (OSB) (Bit B7) to the status byte.

Figure 99: Operation user summary bit (Operation User Register)



The bits used in this register set are described as follows:

- **Bits B0 to B14:** `status.operation.user.BIT0` through `status.operation.user.BIT14`
- **Bits B15:** Not used.

Status function summary

The following functions and attributes control and read the registers. Additional information is included in the command listings for the register sets.

Status function summary

Type	Function or attribute
System summary	status.condition (on page 7-175)
	status.node_event (on page 7-179)
	status.node_enable (on page 7-177)
	status.request_event (on page 7-190)
	status.request_enable (on page 7-188)
	status.reset() (on page 7-192)
Operation event	status.operation.* (on page 7-181)
	status.operation.user.* (on page 7-183)
Questionable event	status.questionable.* (on page 7-186)
Standard event	status.standard.* (on page 7-192)
System events	status.system.* (on page 7-195)
	status.system2.* (on page 7-197)
	status.system3.* (on page 7-199)
	status.system4.* (on page 7-201)
	status.system5.* (on page 7-203)

* The asterisk (`*`) at the end of a command represents `.ntr`, `.ptr`, `.enable`, `.event`, or `.condition`.

Reset and clear registers

You can use commands to reset the status registers.

`*CLS` resets the bits of the event and NTR registers to 0 and sets all PTR register bits on. This command also clears the output queue.

`status.reset()` resets bits of the event and NTR registers to 0 and sets all PTR register bits on. Refer to status.reset() (on page 7-192) for additional information.

In addition to these commands, you can reset the enable registers and the NTR to 0. To do this, send the individual command to program the register with a `0` as its parameter value. The PTR registers can be reset to their defaults by programming them with all bits on. The event registers are not programmable but you can clear them by reading them.

Startup state

When the Model 707B or 708B is turned on, register status elements are set as follows:

- The power on (PON) bit in the `status.operation.condition` register is set.
- Other bits are set appropriately based on the power-on configuration of the instrument.
- All enable registers (`.enable`) are set to 0.
- All negative transition registers (`.ntr`) are set to 0.
- All used positive transition registers (`.ptr`) bits are set to 1.
- The two queues are empty.

Program enable and transition registers

The only registers that you can program are the enable and transition registers. All other registers in the status structure are read-only. The following explains how to determine the parameter values for the commands that are used to program enable registers. The commands are summarized in Status function summary (on page 12-13).

A command to program an event enable or transition register is sent with a parameter value that determines the state (0 or 1) of each bit in the appropriate register. The bit positions of the register (see the following tables) indicate the binary parameter value and decimal equivalent. To program one of the registers, send the decimal value for the bits to be set. The registers are discussed further in Enable and transition registers (on page 12-20).

Bit	**B7**	**B6**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	128	64	32	16	8	4	2	1
Weights	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

Bit	**B15**	**B14**	**B13**	**B12**	**B11**	**B10**	**B9**	**B8**
Binary value	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Decimal	32,768	16,384	8,192	4,096	2,048	1,024	512	256
Weights	(2^{15})	(2^{14})	(2^{13})	(2^{12})	(2^{11})	(2^{10})	(2^9)	(2^8)

When using a numeric parameter, registers are programmed by including the appropriate *`mask`* value. For example:

```
*ese 1169
status.standard.enable = 1169
```

To convert from decimal to binary, use the information shown in the previous table. For example, to set bits B0, B4, B7, and B10, use a decimal value of 1169 for the mask parameter (1169 = 1 + 16 + 128 + 1024).

Read registers

You can read any register in the status structure either by sending the common command query (where applicable), or by including the script command for that register in the `print`() or `print(tostring())` command. The `print()` command outputs a numeric value; the `print(tostring())` command outputs the string equivalent. For example, any of the following commands requests the Service Request Enable Register value:

```
*SRE?
print(tostring(status.request_enable))
print(status.request_enable)
```

The response message is a decimal value that indicates which bits in the register are set. That value can be converted to its binary equivalent using the information in Program enable and transition registers (on page 12-14). For example, for a decimal value of 37 (binary value of 100101), bits B5, B2, and B0 are set.

Register programming example

The following command sequence programs the instrument to generate a service request (SRQ) and set the system summary bit in all TSP-Link nodes when the current limit on channel A is exceeded.

```
-- Clear all registers.
status.reset()

-- Enable SLOT1_THERMAL bit in questionable register.
status.questionable.enable = status.questionable.SLOT1_THERMAL

-- Set the system summary node QSB enable bit.
status.node_enable = status.QSB

-- Set the QSB bit of the service request enable register.
status.request_enable = status.QSB
```

Status byte and service request (SRQ)

Service requests (SRQs) allow an instrument to indicate that it needs attention or that some event has occurred. When the controller receives an SRQ, it allows the controller to interrupt tasks to perform other tasks in order to address the request for service.

For example, you might program your instrument to send an SRQ when:

- All instrument operations are complete
- An instrument error occurs
- A specific operation has occurred

Two 8-bit registers control service requests: The Status Byte Register and the Service Request Enable Register. See Status Byte Register (on page 12-16) for a description of the structure of these registers.

Service requests affect GPIB, USB, and VXI-11 connections. On a GPIB connection, the SRQ line is asserted. On a VXI-11 or USB connection, an SRQ event is generated.

Service Request Enable Register

The Service Request Enable Register controls the generation of a service request. This register is programmed by the user and is used to enable or disable the setting of bit B6 (RQS/MSS) by the Status Summary Message bits (B0, B1, B2, B3, B4, B5, and B7) of the Status Byte Register. As shown in Status Byte Register (on page 12-16), a logical AND operation is performed on the summary bits (&) with the corresponding enable bits of the Service Request Enable Register. When a logical AND operation is performed with a set summary bit (1) and with an enabled bit (1) of the enable register, the logic “1” output is applied to the input of the logical OR gate and, therefore, sets the MSS/RQS bit in the Status Byte Register.

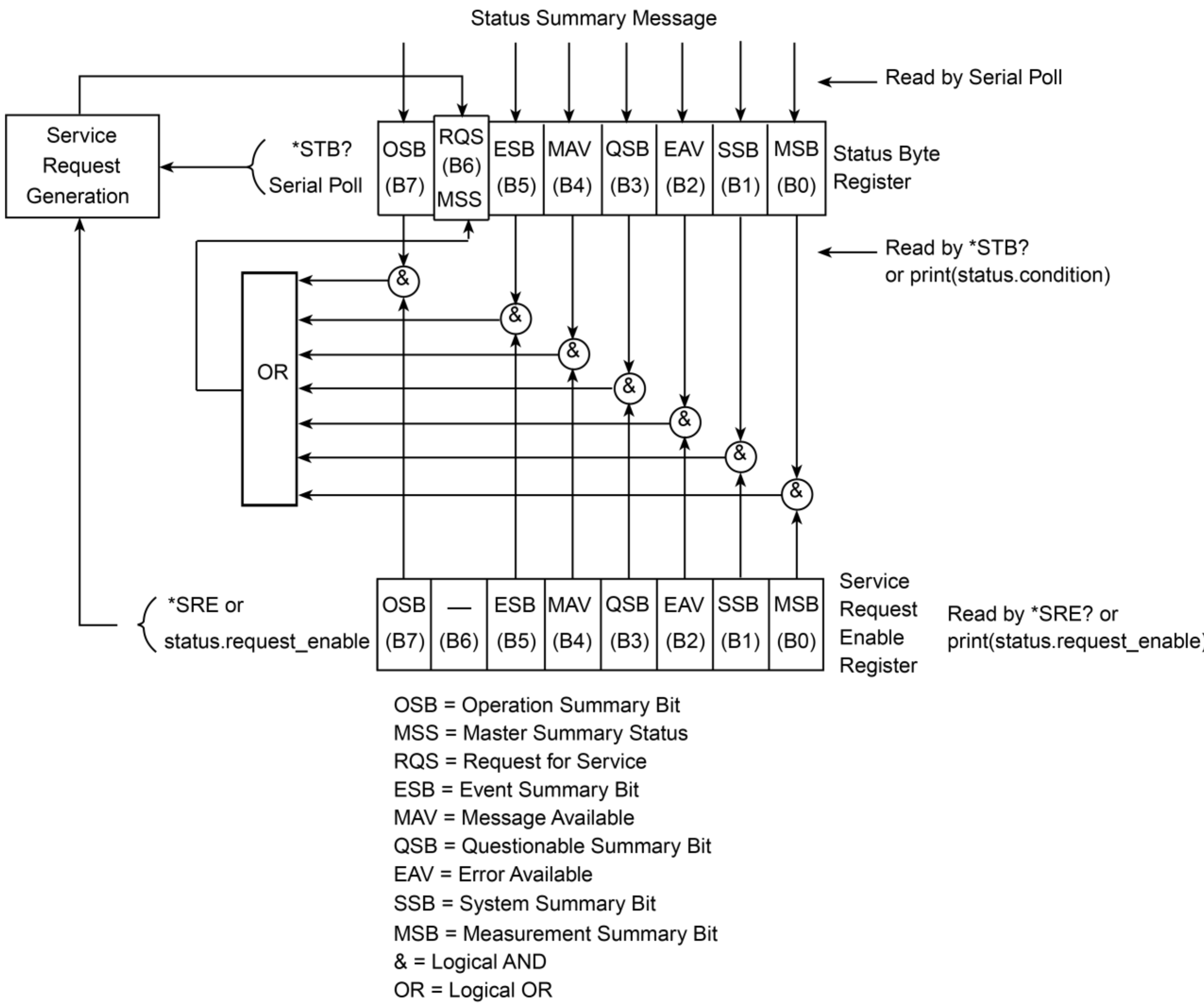
You can set or clear the individual bits of the Service Request Enable Register by using the `*SRE` common command or `status.request_enable`. To read the Service Request Enable Register, use the `*SRE?` query or `print(status.request_enable)`. The Service Request Enable Register clears when power is cycled or a parameter value of 0 is sent with a status request enable command (for example, a `*SRE 0` or `status.request_enable = 0` is sent). The commands to program and read the SRQ Enable Register are listed in Status byte and service request commands (on page 12-19).

Status Byte Register

The Status Byte Register receives summary bits from the other status register sets and queues. The summary messages from the status registers and queues are used to set or clear the appropriate bits (B0, B1, B2, B3, B4, B5, and B7) of the Status Byte Register. These summary bits do not latch, and their states (0 or 1) are dependent upon the summary messages (0 or 1). For example, if the Standard Event Register is read, its register is cleared. As a result, its summary message resets to 0, which then resets the ESB bit in the Status Byte Register.

The Status Byte Register also receives summary bits from itself, which sets the Master Summary Status, or MSS, bit.

Figure 100: Status byte and service request (SRQ)



The bits of the Status Byte Register are described as follows:

- **Bit B0, Measurement Summary Bit (MSB):** Set summary bit indicates that an enabled measurement event has occurred.
- **Bit B1, System Summary Bit (SSB):** Set summary bit indicates that an enabled system event has occurred.
- **Bit B2, Error Available (EAV):** Set bit indicates that an error or status message is present in the error queue.
- **Bit B3, Questionable Summary Bit (QSB):** Set summary bit indicates that an enabled questionable event has occurred.
- **Bit B4, Message Available (MAV):** Set bit indicates that a response message is present in the output queue.

- **Bit B5, Event Summary Bit (ESB):** Set summary bit indicates that an enabled standard event has occurred.
- **Bit B6, Request Service (RQS)/Master Summary Status (MSS):** Set bit indicates that an enabled summary bit of the Status Byte Register is set. Depending on how it is used, bit B6 of the Status Byte Register is either the Request for Service (RQS) bit or the Master Summary Status (MSS) bit:
  - When using the GPIB, USB, or VXI-11 serial poll sequence of the 707B and 708B to obtain the status byte (serial poll byte), B6 is the RQS bit. See Serial polling and SRQ (on page 12-18) for details on using the serial poll sequence.
  - When using the `*STB?` common command or status.condition Status byte and service request commands (on page 12-19) to read the status byte, B6 is the MSS bit.
- **Bit B7, Operation Summary (OSB):** Set summary bit indicates that an enabled operation event has occurred.

Serial polling and SRQ

Any enabled event summary bit that goes from 0 to 1 sets bit B6 and generates a service request (SRQ).

In your test program, you can periodically read the Status Byte to check if an SRQ occurred and what caused it. If an SRQ occurred, the program can, for example, branch to an appropriate subroutine that services the request.

SRQs can be managed by the serial poll sequence of the instrument. If an SRQ does not occur, bit B6 (RQS) of the Status Byte Register remains cleared, and the program proceeds normally after the serial poll is performed. If an SRQ does occur, bit B6 of the Status Byte Register is set, and the program can branch to a service subroutine when the SRQ is detected by the serial poll.

The serial poll automatically resets RQS of the Status Byte Register. This allows subsequent serial polls to monitor bit B6 for an SRQ occurrence that is generated by other event types.

The serial poll does not clear the low-level registers that caused the SRQ to occur. You must clear the low-level registers explicitly. Refer to Reset and clear registers (on page 12-13).

For common commands and TSP commands, B6 is the MSS (Message Summary Status) bit. The serial poll does not clear the MSS bit. The MSS bit remains set until all enabled Status Byte Register summary bits are reset.

Serial polling and SPE and SPD

For the GPIB interface only, the SPE and SPD general bus commands are used to serial poll the Switching System. Serial polling obtains the serial poll byte (status byte). Typically, the controller uses serial polling to determine which of several instruments has requested service with the SRQ line.

Service requests

Service requests (SRQs) affect both the GPIB and the VXI-11 connections. On a GPIB connection, the SRQ line is asserted. On a VXI-11 connection, an SRQ event is generated.

Status byte and service request commands

The commands to program and read the Status Byte Register and Service Request Enable Register are listed in the following table. The table includes both common commands and their script command equivalents. For details on programming and reading registers, see Program enable and transition registers (on page 12-14) and Read registers (on page 12-15).

To reset the bits of the Service Request Enable Register to 0, use 0 as the parameter value for the command (for example, `*SRE 0` or `status.request_enable = 0`).

Status Byte and Service Request Enable Register commands

Command	Description

`*STB?`
or
`print(status.condition)`
	Read the Status Byte Register.

`*SRE` *mask* or `status.request_enable =` *mask*	Program the Service Request Enable Register where *mask* = 0 to 255.

`*SRE?`
or
`print(status.request_enable)`
	Read the Service Request Enable Register.

Enable and transition registers

In general, there are three types of user-writable registers that are used to configure which bits feed the register summary bit and when it occurs. The registers are identified in each applicable command (as listed in TSP commands (on page 7-7)) as follows:

- **Enable register** (identified as `.enable` in the command listing of each attribute): Allows associated events to be included in the summary bit for the register.
- **Negative-transition register** (identified as `.ntr` in the command listing of each attribute): A particular bit in the event register is set when the corresponding bit in the NTR is set, and the corresponding bit in the condition register transitions from 1 to 0.
- **Positive-transition register** (identified as `.ptr` in the command listing of each attribute): A particular bit in the event register is set when the corresponding bit in the PTR is set, and the corresponding bit in the condition register transitions from 0 to 1.

Control node and SRQ enable registers

Attributes to control system node and service request (SRQ) enable bits and read associated registers are summarized in the Status byte register overview (on page 12-4). For example, either of the following commands set the system node QSB enable bit:

```
status.node_enable = status.QSB
status.node_enable = 8
```

TSP-Link system status

The TSP-Link™ expansion interface allows instruments to communicate with each other. The test system can be expanded to include up to 32 TSP-enabled instruments. In a TSP-Link system, one node (instrument) is the master and the other nodes are the subordinates. The master can control the other nodes (subordinates) in the system. See TSP-Link system expansion interface for details about the TSP-Link system.

The system summary registers, shown in System summary bit (System register), are shared by all nodes in the TSP-Link system. A status event that occurs at a subordinate node can generate an SRQ (service request) in the master node. After detecting the service request, your program can then branch to an appropriate subroutine that services the request. See Status byte and service request (SRQ) (on page 12-15) for details.

Status model configuration example

The following example illustrates the status model configuration for a TSP-Link system. In this example, a Node 15 thermal aspect event will set the RQS bit of the Status Byte of the master Node.

When the thermal aspect event occurs on Node 15, the following sequence of events will occur:

1. On Node 15, with Bit B9 of the Questionable event register enabled, when the thermal aspect event occurs, Bit B9 bit sets (`status.questionable.condition`) which causes Bit B9 to be set in `status.questionable.event`. This in turn causes the Questionable event summary bit (QSB) to set.
2. With QSB set, and Bit B3 of the System node enabled (`status.node_enable`), Bit B3 of the Status Byte register (Node 15) sets. This in turn causes the System node summary bit to set.
3. With the System node summary bit set, and Bit B1 of the System2 summary event register enabled (which is Node 15), Bit B1 of the System2 register sets. This in turn causes the System2 event summary bit (EXT) to set.
4. With EXT set, and Bit B0 of the System summary event register enabled, Bit B0 of the System register sets. This in turn causes the System event summary bit (SSB) to set.
5. With SSB set, and Bit B1 of the Service request enable register enabled, Bit B6 of the Status Byte register sets. This in turn initiates a request for service (SRQ).
6. When your program performs the next serial poll of the Master Node, it will detect the interlock event and can branch to a routine to service the request.

NOTE

The System Summary Registers are shared by all nodes in the TSP-Link system. When a bit in a system register of Node 15 sets, the same bit in the master node system register also sets.

The following commands (sent from the master node) enable the appropriate register bits for the above example:

Node 15 status registers: The following commands enable the events for Node 15:

```
node[15].status.questionable.enable = status.questionable.S1THR
node[15].status.node_enable = status.QSB
```

The affected status registers for the above commands are indicated by labels (1) and (2) in the following figure.

System registers: The following commands enable the required system summary bits for Node 15:

```
status.system2.enable = status.system2.NODE15
status.system.enable = status.system.EXT
```

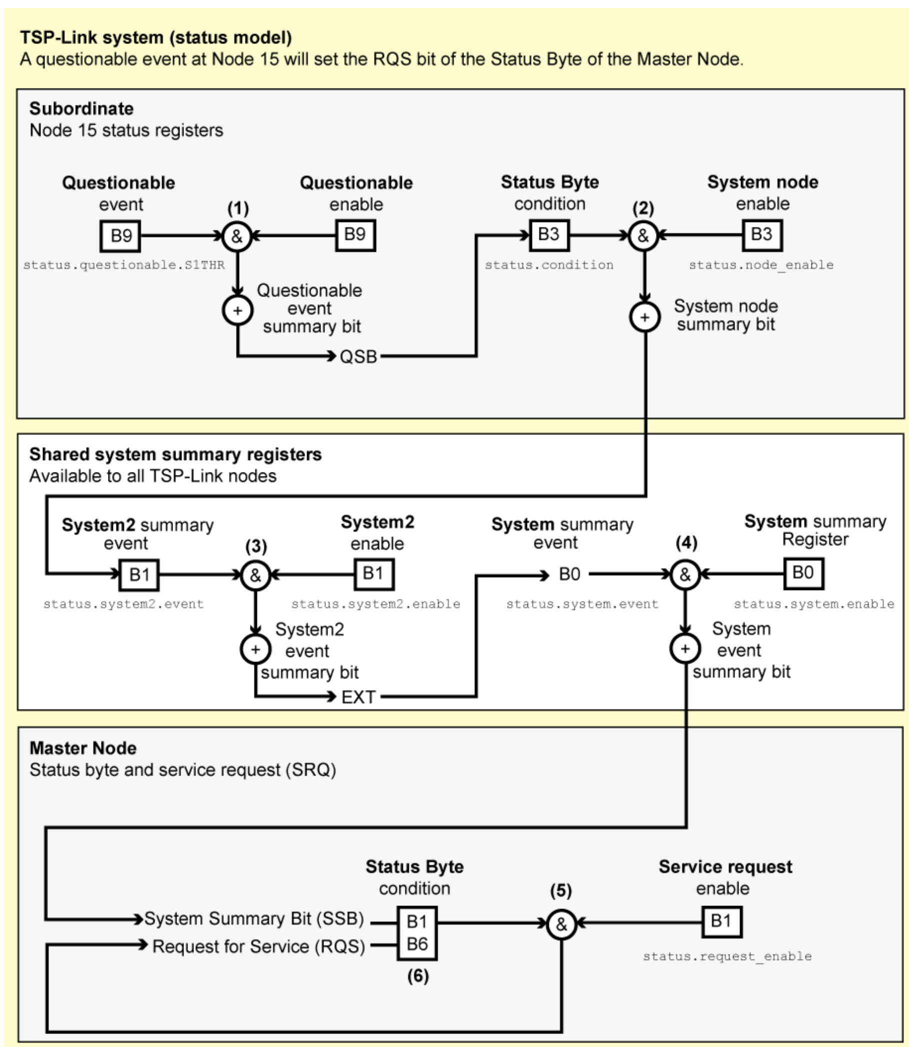
The affected system registers for the above commands are indicated by labels (3) and (4) in the following figure.

Master Node service request: The following command enables the service request for the measurement event:

```
status.request_enable = status.SSB
```

The affected status register for the above command is indicated by labels (5) and (6) in the following figure.

Figure 101: TSP-Link status model configuration example



Migrate from Models 707A and 708A

In this section:

Migrate Model 707A or 708A programs to Model 707B or 708B

This section provides information about migrating existing programs from the Model 707A or 708A to use the TSP programming syntax on the Model 707B or 708B.

The Model 707A and 708A instruments used device-dependent command (DDC) programming. The Model 707B or 708B use the TSP scripting model. This section provides:

- A brief explanation of some Model 707A and 708A model-specific terminology and its Model 707B or 708B equivalent
- Command syntax differences

NOTE

You can also run programs using Model 707A or 708A DDC commands in compatibility mode. Refer to Use Models 707A and 708A compatibility mode (on page 14-1).

Platform differences

When writing a script for the Model 707B or 708B instrument, consider the following platform differences.

	Model 707A or 708A	Model 707B or 708B
Execution host	A computer sends commands over GPIB from a user-generated program such as Visual Basic, C# or C/C++	A computer sends either commands or user-generated scripts over GPIB, USB, or Ethernet. Scripts can also be run from the front panel of the Model 707B or 708B.
Command structure	Single ASCII capital letter commands, followed by argument if necessary.	Descriptive, word-based command and argument structure using modern dot notation format.
Store program flow control	Not possible; instrument can only step sequentially through a stored relay setup in response to an external trigger.	TSP scripting environment allows access to all Lua program control structure operations, such as for-next, if-then-else, while-do, and repeat-until.

Execution host

One of the most significant differences between the Model 707A or 708A and Model 707B or 708B is the ability to store user scripts on the instrument. Once loaded onto the instrument, these scripts can be run without connection to a computer. This is different from the Model 707A or 708A, which could only iterate through an existing stored relay setup if there was no connection to a computer. The ability of the Model 707B or 708B to run scripts that are stored locally results in fast execution time and no communications bottlenecks between the instrument and the computer.

Command structure

The DDC structure of the Model 707A or 708A instrument is based on a single ASCII capital letter with possible alphanumeric arguments and terminated with the "X" execute command.

The Model 707B or 708B instrument uses a modern dot notation format that logically organizes commands by family, subset 1, subset 2 – optional, and argument. For example:

```
channel.open("allslots")
```

Stored program flow control

DDC programming depended on an additional programming environment to send commands to the 707A or 708A instrument. This provided options for program flow control, but at the expense of added complexity.

The TSP scripting environment uses the Lua programming language, providing a robust and cohesive flow control framework.

DDC to TSP command equivalencies

In many cases, DDC commands have a single corresponding TSP equivalent command that performs the same or a similar function.

In some cases, more than one TSP command is needed to provide the same functionality of its equivalent DDC counterpart because of the expanded capabilities and additional flexibility of the TSP scripting environment.

The following sections describe the equivalent commands or sets of commands.

Commands with one-to-one equivalents

The commands in the following table are equivalent.

Commands with one-to-one equivalents

DDC	TSP equivalent command	Description
Crc	`channel.close(channelList)`	Closes a channel
Nrc	`channel.open(channelList)`	Opens a channel
Hn	`display.sendkey()`	Emulates a keypress
P0	`channel.open("allslots")`	Opens all channels on all slots
R0	`reset()`	Restore factory defaults
U2, 0	`channel.getclose("allslots")`	Get closed channels on all slots
X	`No corresponding command`	Execute
J0	`No corresponding command`	Self-test
U7	`digio.readport()`	Read value from digital I/O port

Relay setup commands

The following table shows DDC commands that refer to Model 707A and 708A relay setups. The Model 707B or 708B equivalent of relay setups are called scans and patterns.

Scans allow a predetermined set of channels to be closed in a sequential order based on a particular event or combination of events. For more information, see Scan and trigger (on page 2-21).

There are some distinct differences between relay setups and scans:

- Scans can only be accessed sequentially. Unlike relay setups, you cannot specify an alternate starting location.
- You can only retrieve scan lists in string format.
- You can only append steps to scans. You cannot insert scan steps before steps that are already in the scan list.
- You cannot remove scan steps from the scan list.

The correlation to scanning holds true for DDC commands `In`, `En`, `E0`, `Lbbbb`, `Pn`, and `Qn`, while the DDC commands `Z0,n`, `Zn,0`, and `Zm,n` resemble patterns on the Model 707B or 708B instrument. Patterns allow several channels to be associated by a name, and any operation that occurs to the pattern (such as close or open) happens to each channel in the pattern. For more information, see Channel patterns (on page 2-14).

Scanning trigger

DDC	TSP equivalent command	Description
`En`	No corresponding command	Point to present relay setup
`E0`	No corresponding command	Point to stored relay setup
`Lbbbb`	`print(scan.list())`	Download setups
`Pn`	No corresponding command	Clear relay setup
`Qn`	`scan.create("")`	Delete setup
`Z0,n`	`channel.pattern.snapshot("`*`patternName`*`")`	Copy present relays to *n*
`Zn,0`	`channel.close("`*`patternName`*`")`	Copy setup from *n* to relays
`Zm,n`	`channel.pattern.setimage(`*`pattern1, pattern2`*`)`	Copy setup from *m* to *n*
`In`	No corresponding command	Insert blank setup to mem

Disable or enable the scan event trigger

When enabling external triggering on the Model 707A or 708A instruments, there are two possible trigger sources, the external trigger line or a GPIB `Get` command. The Model 707B or 708B has multiple trigger events that you can use to iterate through a scan list. The following table shows that specifying a scan event trigger automatically enables it. For more information, see Scan and trigger (on page 2-21).

Scanning trigger

DDC	TSP equivalent command	Description
`F0`	`scan.trigger.channel.stimulus = 0`	Disable scanning trigger
`F1`	`scan.trigger.channel.stimulus = eventID`	Enable scanning trigger

Trigger polarity

The Model 707A or 708A instrument has one external trigger input that can accept TTL level trigger inputs to advance the stored relay setup step.

The Model 707B or 708B instrument allows 14 TTL level inputs through the digital I/O connector to act as events to iterate through a scan list. The handling of this command is similar to the `A0` and `A1` DDC commands shown in the following table, except that you must specify [*N*] as the digital I/O line polarity you need to set. For more information, see Scan and trigger (on page 2-21).

Scan trigger polarity

DDC	TSP equivalent command	Description
`A0`	`digio.trigger[N].mode = digio.TRIG_FALLING` `scan.trigger.channel.stimulus = digio.trigger[N].EVENT_ID`	Select falling edge for scan trigger
`A1`	`digio.trigger[N].mode = digio.TRIG_RISINGA` `scan.trigger.channel.stimulus = digio.trigger[N].EVENT_ID`	Select falling edge for scan trigger

Matrix ready

The Model 707A or 708A instrument provides a dedicated TTL output for when a channel close or open operation completes. This output includes the settling time of the specific channel (fixed in hardware) added to a programmed settling time that could be specified by the user. These commands are shown in the following table.

Matrix ready polarity

DDC	TSP equivalent command	Description
`B0`	No corresponding command	Rising edge on matrix ready
`B1`	No corresponding command	Falling edge on matrix ready

The Model 707B or 708B instrument does not have a dedicated matrix ready line when the DDC command compatibility is disabled. However, you can create a TSP function that can emulate the dedicated hardware matrix ready function onto a digital I/O line. The following examples show you how to create TSP code functions for channel close and channel open, respectively, that will duplicate matrix ready functionality for digital I/O Bit B10. After including these functions in your script, a channel close operation is made by calling the function with the channel argument:

```
chanClose("1A01")
```

For more information, see Digital I/O port (on page 3-9).

Example: Channel close with matrix-ready functionality

```
function chanClose(chan)
digio.writebit(10,0)
channel.close(chan)
digio.writebit(10,1)
end
```

Function accepts valid channels as arguments.
Clear the 10th digital I/O bit before the close operation.
Close the channels in the argument.
Set the 10th digital I/O bit after the close operation.
End the function.

Example: Channel open with matrix-ready functionality

```
function chanOpen(chan)
digio.writebit(10,0)
channel.open(chan)
digio.writebit(10,1)
end
```

Function accepts valid channels as arguments.
Clear the 10th digital I/O bit before the open operation.
Open the channels in the argument.
Set the 10th digital I/O bit after the open operation.
End the function.

Set front-panel display

The following commands display a value on the front-panel display. To return the display to the power-on default, send the command:

```
display.screen = display.main
```

The display can also be cleared with the `display.clear()` command.

Display

DDC	TSP equivalent command	Description
`Dccccccc`	`display.screen = display.USER`	Set front-panel display text
	`display.settext()`	

Set relay setup data format

The `Gn` DDC command controls the data format of relay setups that are returned when queried with the `U2,s` command. The Model 707B or 708B uses a string format for returning scan list, so data formatting is not used.

Relay setup data format

DDC	TSP equivalent command	Description
`Gn`	No corresponding command	Data format

GPIB EOI hold off control

The `Kn` DDC command controls how the GPIB EOI line behaves. The Model 707B or 708B commands are processed from an internal buffer as they are received, so no control is required for the GPIB EOI line.

GPIB EOI hold off

DDC	TSP equivalent command	Description
`Kn`	`No corresponding command`	GPIB EOI/hold off control

Service request configuration

These commands control the behavior of the SRQ line. The Model 707B or 708B instrument can configure some of the events that cause the SRQ to assert, but none of those events address Matrix Ready or when the instrument is ready for trigger. For more information, see Status model functions (on page 5-14).

Service request configuration

DDC	TSP equivalent command	Description
`M0`	`status.request_enable = 0`	Disable SRQ
`M8`	No corresponding command	SRQ on Matrix Ready
`M16`	No corresponding command	SRQ on Matrix Ready
`M32`	`status.request_enable = status.ERROR_AVAILABLE`	SRQ on error

Digital I/O control

The Model 707B or 708B instrument provides increased flexibility, with 14 digital I/O lines. The Model 707A has eight input lines and eight output lines, and the Model 708A has sixteen input lines and sixteen output lines. On the Model 707B or 708B, it is possible to write to the digital I/O port simultaneously or on a per-bit basis. For more information, see Digital I/O (on page 5-7).

Digital I/O control

DDC	TSP equivalent command	Description

`Onnn`
 	
`Db,s`
	`digio.writeport(data)`
	or
	`digio.writebit(bit,data)`
		Sets states of digital I/O

Additional channel settling time

The Model 707A or 708A had the ability to add relay settling time to the system after a channel closes or opens. The Model 707B or 708B improves on the settling time and allows you to apply additional settling delay on a per-channel basis. For more information, see channel functions and attributes (on page 5-6).

Additional channel settling time

DDC	TSP equivalent command	Description
`Sn`	`channel.setdelay(channelList,value)`	Additional settling time

Retrieve card settling time

Although base channel settle time cannot be returned, additional delay time that was specified with the `channel.setdelay(channelList,delay)` command can be returned using:

```
channel.getdelay("allslots")
```

Delays for each channel in csv format are returned. For more information, see channel functions and attributes (on page 5-6).

Card settling time

DDC	TSP equivalent command	Description
`U6`	No corresponding command	Send longest settling time

Trigger source control

The DDC commands in the following table control the trigger source for advancing the relay setups. The Model 707B or 708B instrument provides similar functionality, but omits triggering on talk and triggering on X. For more information, see Scan and trigger (on page 2-21).

Trigger source control

DDC	TSP equivalent command	Description
`T0 or T1`	No corresponding command	Trigger on Talk
`T2 or T3`	`scan.trigger.channel.stimulus = trigger.EVENT_ID`	Trigger on GPIB Get
`T4 or T5`	No corresponding command	Trigger on X
`T6 or T7`	`scan.trigger.channel.stimulus = digio.trigger[N].EVENT_ID`	Trigger on external trigger

Send machine status word

The DDC `U0` command provides information on several areas of the Model 707A or 708A instrument in one long csv form. There is no one command to provide the same functionality as the `U0` command, but the series of Model 707B or 708B commands shown in the following table can be used to provide similar information, without the need to parse a long unwieldy string.

Send machine status word

DDC	TSP equivalent commands	Description

`U0`	`print(localnode.model)`
	`print(scan.trigger.channel.stimulus)`
	`print(display.getlastkey())`
	`print(status.request_enable)`
	`print(digio.readport())`
		Returns the model number
		Returns trigger stimulus line
		Returns the last key pressed
		Gets SRQ Mask
		Reads Digital I/O port

Error status word

This series of commands provides a descriptive list of error codes and error messages until no more errors are present. For more information, see Error queue (on page 5-8).

Error status word

DDC	TSP equivalent command	Description

`U1`	`count = errorqueue.count`
	`for x=count,0,1 do`
	`errorcode, message = errorqueue.next()`
	`print(errorcode)`
	`print(message)`
	`end`
		Get the number of errors in the queue
		For loop
		Get next error codes
		Print `errorcode`
		Print error messages
		End

Relay pointer operations

The DDC commands in the following table use relay setups as the Model 707B or 708B instrument use patterns.

The `U2,n` DDC command returns the channels that are closed in the format set by the DDC `Gn` command. The corresponding command for this operation returns the channels contained in the pattern as a csv list. There is no need to send a pattern to the relays. Operations that are to be done to the channels of a pattern are done on the pattern correctly. For more information, see channel functions and attributes (on page 5-6).

Relay pointer operations

DDC	TSP equivalent command	Description
`U2,n`	`channel.pattern.getimage(pattern)`	Point to stored relay setup
`U3`	No corresponding command	Send relay step pointer

Number of subordinates

You can add the Model 707B or 708B to a TSP-Link system to communicate with other Keithley products that are enabled for TSP-Link. When the command listed in the following table is executed, it returns the number of nodes present in the TSP-Link system.

A feature of the Model 707B or 708B is that these remote nodes are not limited to Model 707B or 708B instruments. They can be remotely linked to any Keithley product that supports TSP-Link. For more information, see TSP-Link system (on page 6-47).

Number of subordinates

DDC	TSP equivalent command	Description
`U4`	`print(tsplink.reset())`	Send number of subordinate devices

Get card identification information

The following command returns the model of the card in the specified slot of the local instrument, along with a description of the card, the firmware version, and the serial number.

To get information from cards in TSP-Link connected instruments, send:

```
print(node[nodeNumber].slot[slotNumber].idn)
```

Where *nodeNumber* is the TSP-Link node of the remote system and *slotNumber* is the slot on the instrument to query. For more information, see Slot (on page 5-13) and TSP-Link (on page 5-15).

Card model number

DDC	TSP equivalent command	Description
`U5,n`	`print(slot[slotNumber].idn)`	Send identification of each card in the instrument

Relay test input

The Model 707B or 708B instrument does not use a relay test connector, so this command is not supported.

Relay test input

DDC	TSP equivalent command	Description
`U8`	No corresponding command	Send relay test input

Connect rules

On the Model 707B or 708B, this command sets the connect rules for the entire instrument. You cannot set this rule on a row-by-row basis. If this feature is required, the high performance of the Model 707B or 708B instrument provides ample time to change the connection rules as needed before channel close or open operations. For more information, see channel functions and attributes (on page 5-6).

Connect rules

DDC	TSP equivalent command	Description
`Vabcdefg`	`channel.connectrule = channel.MAKE_BEFORE_BREAK`	Make-before-break
`Wabcdefg`	`channel.connectrule = channel.BREAK_BEFORE_MAKE`	Break-before-make

Termination character sent by instrument

The Model 707B or 708B always sends a linefeed (ASCII 10) as a termination character after any data. This behavior cannot be changed.

Termination characters

DDC	TSP equivalent command	Description
`Yn`	No corresponding command	Termination character

Models 707A and 708A compatibility mode

In this section:

Use Models 707A and 708A compatibility mode

This document provides information about using the Models 707B and 708B as a drop-in replacement in an existing Keithley Model 707A or 708A application.

The Models 707B and 708B provides improved functionality compared to the Model 707A or 708A. However, established customer test systems can be sustained after replacement without significant reconfiguration and code changes by using compatibility mode.

When you use the Models 707B and 708B compatibility mode, you can emulate a Model 707A or 708A, but you will not have access to the full range of features available in its native mode.

This section describes:

- How to configure the 707B or 708B for compatibility mode.
- The differences between the Model 707A or 708A and the 707B or 708B.

Model A to Model B differences

You can use a Model 707B or 708B in an existing 707A or 708A application. The units are compatible with the following exceptions:

- Master/subordinate operation is not supported.
- The digital I/O is limited to twelve bits for all models.
- Relay test is not supported.
- Some commands operate differently (see Models 707A and 708A commands (on page 14-4)).

When using 707A or 708A compatibility mode, the only compatible remote interface is GPIB.

You can select one of two options when enabling Model 707A or 708A DDC compatibility mode:

- **70xA-VERSION**: This option most closely matches Model 707A or 708A operation. Use this version if you are transferring applications directly from a Model 707A or 708A with few changes.
- **70xB-VERSION**: This options provides enhanced operation, including error checking and more robust settling time operation. Use this version if you are updating existing applications.

To enable Model 707A or 708A DDC compatibility mode:

1. From the front panel, select **MENU**.
2. Select **DDC**.
3. Select **ENABLE**.
4. Select the version.
5. Press **ENTER**.
6. Cycle instrument power.

Front-panel relay closure indicators

When you are using Model 707A or 708A compatibility mode, the channel information is displayed on the updated display.

NOTE

Models 707B and 708B channel notation is different than on the Models 707A and 708A. For Models 707A and 708A, the slot number is built into the column number. For Models 707B and 708B, the slot number is the first number of the channel notation. For example, in Models 707A and 708A, crosspoint "A56" refers to slot 5, row A, column 8. This same crosspoint appears as "5A08" on the Models 707B and 708B front-panel display. However, with DDC emulation enabled, the Models 707B and 708B will accept "A56" and close the correct crosspoint.

Timing issues

The 707B and 708B run much faster than their predecessors. When using code from an older switch model, be aware that timing problems could be introduced into the system.

Digital interface

When you are using Model 707A or 708A compatibility mode, the digital I/O is fixed as follows:

- Digital input: Digital I/O lines 1 to 6
- Digital output: Digital I/O lines 7 to 12
- External trigger: Digital I/O line 13
- Matrix ready: Digital I/O line 14

Refer to Digital I/O port (on page 3-9) for the pinout diagram for the digital I/O connector.

Memory setups

Memory setups are handled as a single channel pattern when you use Model 707A or 708A compatibility mode.

The Model 707B or 708B supports 100 memory setups, in addition to the channel patterns normally available in the Model 707B or 708B.

The memory patterns are named MEMSETUP*xxx*, where *xxx* is between 001 and 100.

Memory patterns are created as they are used.

You can work with memory patterns through TSP as you do with channel patterns. See Channel patterns (on page 2-14).

Models 707A and 708A commands

Models 707A and 708A commands

Command	Description	Differences in 707B and 708B
`An`	Edge for which an externally generated pulse executes a trigger	None.
`Bn`	Select logic sense of matrix ready	Actual matrix ready signal may have different timing characteristics. If a relay does not change state, the matrix ready signal does not include the relay settle time.
`Crc(,rc)..(,rc)`	Close crosspoints in a setup	Actual ready and matrix ready signals may have different timing characteristics. The Ready signal includes the relay settle time. Subordinate units are not supported. In the case of multiple commands, all crosspoints are closed simultaneously.
`Dnnnnnn`	Set text on the display	Only available on Model 707B. The display character limit has been increased to 20 and the text is displayed on the first line of the VFD display.
`Db,s`	Set digital output	Only available on Model 708B. The actual digital output settings will not function for bits that are outside of the physical interface specifications (see Digital interface (on page 14-3)).
`En`	Specify setup number	None.
`Fn`	Enable or disable triggers	None.
`Gn`	Output format	None.
`Hn`	Front-panel key	Only Model 707A. This command has no effect and does not issue an error message when used.
`In`	Insert blank setup	None.
`Jn`	Self-test	There is no self-test for the 707B and 708B.
`Kn`	EOI and hold off	Actual ready and matrix ready signals may have different timing characteristics. The Ready signal on the Model 707B or 708B includes the relay settle time. The Models 707A and 708A could hold off the GPIB bus on the specific 'X' character. To increase performance, the 707B and 708B use a message-based system. Therefore, the hold off has been changed to 'hold off on end of message'. When the 'X' is at the end of the message, the functionality is identical.
`Lbbbb…X`	Download setup	None.
`Mn`	SRQ and Serial Poll Byte	Performing a self-test or pressing a key does not clear the SRQ Ready bit.
`Nrc(,rc)...(,rc)`	Open crosspoint	Actual ready and matrix ready signals may have different timing characteristics. The ready signal on the Model 707B or 708B includes the relay settle time. Subordinate units are not supported. All crosspoints are opened. For example, for the command `NA1NA2NA3`, all three crosspoints are opened. All crosspoints are opened simultaneously.
`Ovvv`	Digital output	The actual digital output settings will not function for bits that are outside the physical interface specifications (see Digital interface (on page 14-3)).
`Pn`	Clear crosspoints	None.

Models 707A and 708A commands

Command	Description	Differences in 707B and 708B
Qn	Delete setup	None.
Rn	Restore defaults	None.
Sn	Programmable settle time	Works as intended in 70xB-VERSION. For 70xA-VERSION, settle time is always zero (0).
Tn	Trigger source	The trigger source 0/1 (GPIB Talk) is not available for the 707B and 708B. The factory default address for the 707B and 708B is 16. The trigger source 8/9 (front-panel key) is not implemented.
Un	Instrument Config/Status	For U0, the last pressed key always reads 05. For U1, the Self-Test, PowerUp, and Master/Subordinate Loop bits always read zero. For U4, the number is always zero. For U5, the subordinates always read zero. For U7, though compatible bits are returned, the actual digital input settings read zero for those bits outside of the physical interface specifications (see digital input/output difference). For U8, the test relay is not supported and always reads 15.
Vabcdefgh	Make before break	None.
Wabcdefgh	Break before make	None.
<cmd>X	Execute	None.
Yn	Change line terminate	None.
Zm,n	Copy setup	None.
<space>	Model and version	Sending a <space> only in a message causes the unit to return the model and version number.
*idn?	Unit identification	For compatibility, the version number is A03 (version number of DDC compatibility, not the firmware version). For example, on the Model 707A, the return string is 707A03.

There is a limit of 64 commands per execution.

For more detail on the Model 707A and 708A commands, see the appropriate instruction manual:

- For the Model 707A: 707A-901-01 (A - Sep 1998)(Instruction).pdf
- For the Model 708A: 708A-901-01 (A - Sep 1998)(Instruction).pdf

These instruction manuals are available on tek.com/support.

Section 15

Next steps

In this section:

Additional 707B and 708B information

For additional information about the Model 707B or 708B, refer to tek.com/keithley, which contains the most up-to-date information. From the website, you can access:

- *The Low Level Measurements Handbook: Precision DC Current, Voltage, and Resistance Measurements*
- *Switching Handbook: A Guide to Signal Switching in Automated Test Systems*
- Application notes
- Updated drivers
- Information about related products, including:
  - Switch cards, including the Models 7072, 7072-HV, 7173-50, and 7174A
  - Series 2600B System SourceMeter™ Instruments
  - The Model 4200A-SCS Parameter Analyzer

In addition, your local Field Applications Engineer can help you with product selection, configuration, and usage. Check the website for contact information.

Keithley Instruments • 28775 Aurora Road • Cleveland, Ohio 44139 • 1-800-833-9200 • tek.com/keithley

KEITHLEY
A Tektronix Company

04/2022