www.keithley.com

S530 Parametric Test System

KTE Linear Parametric Test Library Manual

S530-900-01 Rev. B / January 2014

S530-900-01

A Greater Mesure of Confidence

S530 Parametric Test Systems

KTE Linear Parametric Test Library

Manual

© 2011-2014, Keithley Instruments, Inc.

Cleveland, Ohio, U.S.A.

All rights reserved.

Any unauthorized reproduction, photocopy, or use of the information herein, in whole or in part, without the prior written approval of Keithley Instruments, Inc. is strictly prohibited.

All Keithley Instruments product names are trademarks or registered trademarks of Keithley Instruments, Inc. Other brand names are trademarks or registered trademarks of their respective holders.

Document number: S530-900-01 Rev. B / January 2014

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 qualified 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 Instruments 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, as defined in the specifications and operating information, and as shown on the instrument or test fixture panels, or switching card.

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 danger. 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 caution, 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 dangers 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.

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 Instruments. Standard fuses with applicable national safety approvals may be used if the rating and type are the same. 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 Instruments to maintain accuracy and functionality of the product). If you are unsure about the applicability of a replacement component, call a Keithley Instruments office for information.

To clean an instrument, use a damp cloth 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 January 2013.

Table of Contents

Introduction **1-1**

Manual contents 1-1

Linear parametric test library **2-1**

Introduction 2-1

Instruments and instrument drivers 2-1

Instrument and terminal IDs 2-2

Measuring 2-3

Sourcing and compliance 2-4

Ranging 2-5

Matrix operations 2-6

Sweeping 2-6

Triggers 2-7

GPIB 2-8

Optimizing test sequences 2-8
- Fixed range vs. auto-range measurements 2-8
- Fix-range trigger instruments 2-8
- Make use of "combination" commands 2-8
- Synchronization 2-9

Error handling 2-10
- Calling execut and inshld as a function 2-10
- Calling getlpterr 2-10
- Error messages 2-10
- Special error values returned 2-10

Library structure 2-10
- Matrix commands 2-11
- Source commands 2-11
- Range commands 2-11
- Measure commands 2-11
- Combination commands 2-12
- Execution/synchronization commands 2-12
- General commands 2-12
- Timing commands 2-13
- GPIB commands 2-13

LPT Library Function Reference **3-1**

Introduction 3-1

LPT functions 3-4
- addcon - Add connection 3-4
- adelay - Array delay 3-4
- asweepX - Array sweep 3-5
- avgX - Average 3-7
- bmeasX - Block measure 3-9
- bsweepX - Breakdown sweep 3-11

clrcon - Clear connections 3-12
clrscn - Clear scan table 3-13
clrtrg - Clear trigger 3-14
conpin - Connect pin 3-16
conpth - Connect path 3-17
delay - Delay 3-18
delcon - Delete connection 3-19
devclr - Device clear 3-19
devint - Device initialize 3-20
disable - Disable timer 3-20
enable - Initialize and start timer 3-20
execut - Execute 3-21
forceX - Force a voltage or current 3-21
getlpterr - Get LPT error 3-22
getstatus - Get instrument status 3-22
imeast - Immediate measure 3-22
inshld - Instrument hold 3-23
intgX - Integrate 3-24
kibdefclr - Keithley GPIB define device clear 3-25
kibdefint - Keithley GPIB device initialize 3-26
kibrcv - Keithley GPIB receive 3-27
kibsnd - Keithley GPIB send 3-28
kibspl - Keithley GPIB serial poll 3-29
kibsplw - Keithley GPIB 3-30
limitX - Limit a voltage or current 3-31
lorangeX - Select bottom range 3-32
measX - Measure 3-33
mpulse - Measure pulse 3-35
pulseX - Voltage or current pulse 3-36
rangeX - Select range 3-38
rdelay - Real delay 3-39
rtfary - Return force array 3-39
savgX - Sweep average 3-40
searchX - Binary search measurement 3-42
setauto - Re-enable autoranging 3-45
setmode - Set component mode 3-46
setXmtr - Set meter mode 3-48
sintgX - Sweep integrate 3-49
smeasX - Sweep measure 3-50
ssmeasX - Steady state measurement 3-52
sweepX - Sweep 3-54
syncmode - Set synchronization mode 3-57
trigXg, trigXl - Trigger greater than, less than 3-58
tstel - Test station select 3-60

Error Definitions 4-1

Result values indicating an error 4-1

Error messages 4-2
3 LPT_NOCOMCHAN 4-2
5 SYS_MEM_ALLOC_ERR 4-2
20 LPT_PREVERR 4-2
21 LPT_FATAL 4-2
22 LPT_FATALINTEST 4-2
24 LPT_TOMANYARGS 4-2
100 MX_INVLDCNT 4-3
101 MX_NOPIN 4-3
102 MX_MULTICON 4-3
109 MX_ILLGLTSN 4-3
113 MX_NOSWITCH 4-3

114 MX_ILLGLCON 4-3
122 UT_INVLDPRM 4-3
126 UT_NOURAM 4-3
129 UT_TMRIVLD 4-4
137 UT_INVLDVAL 4-4
152 CB_BADFUNC 4-4
156 CB_NOFILE 4-4
157 CB_FORMAT 4-4
162 CB_INVLDERROR 4-4
163 CB_INVLDEVENT 4-4
166 CB_INSNOTREC 4-4
194 MX_INVLDTRM 4-5
233 FM_NOCON 4-5
455 ECP_PROTOVER 4-5
601 SYS_INTERNAL_ERR 4-5
610 SYS_SPAWN_ERR 4-5
611 SYS_NETWORK_ERR 4-5
612 SYS_PROTOCOL_ERR 4-5
650 TAPI_BADCHANNEL 4-5
651 TAPI_BADTESTER 4-6
652 TAPI_NOTFOUND 4-6
653 TAPI_REFUSED 4-6
656 TAPI_CHANLIMIT 4-6
657 TAPI_BUFOFLOW 4-6

Introduction

The Linear Parametric Test Library contains detailed information about the S530 LPTLib subroutines. It is intended as a reference guide for experienced users. This manual describes each subroutine in detail.

In this section:

Manual contents 1-1

Manual contents

Section 1 Introduction describes how the document is organized and what information is presented in each section.

Section 2 Linear parametric test library describes how LPTLib is used, and what functions it provides during test plan creation.

Section 3 LPT Library Function Reference provides full details for the user of the Linear Parametric Test Library, which contains the analysis subroutines built into the Parametric Test System software.

Section 4 Error Definitions describes error messages.

Section 2

Linear parametric test library

In this section:

Introduction 2-1
Instruments and instrument drivers 2-1
Instrument and terminal IDs 2-2
Measuring 2-3
Sourcing and compliance 2-4
Ranging 2-5
Matrix operations 2-6
Sweeping 2-6
Triggers 2-7
GPIB 2-8
Optimizing test sequences 2-8
Error handling 2-10
Library structure 2-10

Introduction

The Keithley line of parametric testers utilizes function libraries to control the instrumentation in the system. These libraries are called "Test Control Libraries." A Test Control Library is the lowest level software interface to a parametric tester. This manual documents one such library specifically known as the "Linear Parametric Test Library," or LPTLib.

Instruments and instrument drivers

LPTLib provides the lowest level of instrument control available, however, LPTLib does not control the hardware directly. Instead, LPTLib relies on hardware drivers to control instruments. Each of these drivers only supports LPTLib commands appropriate for their function. For example, the driver for a voltmeter would not support the function to measure frequency. The LPT LIbrary Function Reference (on page 3-1) section of this manual provides information about which LPTLib functions are supported by the driver for each instrument.

Instrument and terminal IDs

LPTLib uses instrument identification codes to refer to the various instruments. An instrument identification code is simply an integer value. This manual, never refers to the actual numbers used to identify the instruments; but, instead refers to the mnemonic codes that you can use in your programs to refer to the various instruments. An instrument ID generally consists of a mnemonic string that identifies the type of instrument and a number that specifically distinguishes which individual instrument of this type. For example, SMU2 is an instrument ID that refers to the second source measure unit in the system.

This manual often refers to a SMU instrument as SMUn (the "n" represents a number), however, it does not matter which SMU is being used. Anywhere an instrument ID is required by a library function, a particular instrument ID (mnemonic) can be used directly or as an integer variable, which was assigned the value of an instrument ID. For example, SMUn can indicate SMU1, SMU2, SMU3, and so on.

Most instruments have terminals which need to be connected to your circuit before the instrument can be used in a test sequence. For example, a simple voltmeter will have two terminals: a high terminal and a low terminal. An instrument's terminals have identification codes just like the instruments themselves do. In fact, in some places an instrument ID and terminal ID can be used interchangeably. Some example terminal IDs are SMU3H and SMU3L which are the high and low terminals of Source Measure Unit number three.

The LPT LIbrary Function Reference (on page 3-1) section contains information for each instrument in your system which can be controlled by LPTLib. This section specifies the instrument and terminal IDs which are associated with the instrument. There are also some special instrument IDs and terminal IDs that the system recognizes, but are not associated with any particular instrument.

Special instrument IDs:

- KI_SYSTEM - This refers to the system itself. Note KI_SYSTEM is a special pseudo instrument and the instrument ID does not refer to the collection of instruments in the system, but to the pseudo instrument itself.

Special terminal IDs:

- CHUCK - The chuck connection. The sense pin option cannot be used with CHUCK. Use CHUCKM instead.
- KI_EOC - This is a special terminal ID used to terminate a list of terminals in a connection subroutine. Its value is specifically 0 and one will usually use the value 0 instead of KI_EOC.

Measuring

The most important part of a parametric tester is its ability to make measurements. There are three types of measurements you can make with LPTLib: ordinary measurements, integrated measurements, and averaged measurements. The type of measurement you make depends on the type of noise you are trying to eliminate from your measurement.

Ordinary measurements are made with the **measX** function. This is the fastest single point measurement you can make. This type of measurement is used where speed is most important or where noise is not significant.

Averaged measurements are made with the **avgX** function. An averaged measurement is made by making several single point measurements and averaging them together. Averaged measurements reduce the effects of random noise.

Integrated measurements are made with the **intgX** function. Integrated measurements examine the signal over a longer period of time and reduce the effects of AC noise. Like averaged measurements, integrated measurements reduce the effects of random noise, but they also strongly reject noise with a period that is an integer multiple of the integration period or aperture.

Some instruments allow their integration aperture to be changed. Integration apertures are commonly defined in units of power line cycles or PLCs. A power line cycle is the time it takes for one complete AC cycle of the main power supplied to the system. The default integration aperture is one power line cycle (1 PLC). For 60Hz power, one power line cycle is 16.667ms; for 50Hz one power line cycle is 20ms. An integration aperture of 0.1 PLC on a 60Hz system is (0.1)(16.667ms) = 1.667ms.

A common mistake is to try and use large aperture integrated measurements to eliminate the effect of random noise. Although noise can be reduced this way, it is typically more productive to use an averaged measurement. The measurement will generally be as stable, but averaged measurements can usually be made faster than integrated measurements. This is especially true when performing autoranged measurements. When an instrument takes autoranged measurements, all of the measurements it takes when determining the best range will be performed at the same aperture. If the instrument throws out any measurements because they were on a sub-optimal range, much time would be wasted if they were made with a large aperture. With averaged measurements, the instrument typically spends less time finding the optimal range.

Some instruments allow combinations of integration and averaging to be performed together by allowing the behavior of one or more of the three types of measurements to be altered temporarily. In this case the instrument is capable of performing an averaged integrated measurement.

Sourcing and compliance

Sources are normally forcing 0.0 on their default sourcing function. The source value can be changed with the **forceX** function. All current and voltage sources restrict the complimentary function to the one they are sourcing. Current is the complimentary function of voltage, and voltage is the complimentary function of current. For example, when a voltage source is forcing voltage, it restricts how much current it will allow to flow (including when it is sourcing its default 0.0 value). This limit is known as the compliance limit. When this limit is reached, the source will back off on its force value. In our example, when the voltage source reaches its compliance limit, it will back off on the voltage being forced such that the current does not exceed its compliance limit. When this happens, the source is said to be in compliance. More specifically, the voltage source is in current compliance.

All sources have default compliance limits, but the compliance limit value the source uses can be changed with the **limitX** command. Note that a current limit is used for a voltage source and a voltage limit is used for a current source.

When there are active sources in a test sequence, all sources can be brought back to their default source level of 0.0 with the **devclr** function. This resets the source level only. If a compliance limit or any other instrument setting has been changed from its default, it is not reset by the **devclr** call.

To reset all instrument settings to their initial or default state, the **devint** function is used. The **devint** function will also clear all sources by internally calling **devclr** before resetting all instrument settings back to their default.

Ranging

The default mode of operation for all instruments is to automatically select the best range available for sourcing as well as for measuring. This is known as autoranging. For sources, the range is picked when the source value is changed. When measuring, the instrument may need to make several measurements, each on a different range, until it finds the best range for the measurement.

Autoranged measurements can take much longer to make than fixed range measurements because the meter may need to change ranges and take extra measurements before finding the correct range. There are two features that instruments may support that improve the performance of autoranged measurements. These are smart ranging and sticky ranging.

An instrument without smart ranging will successively uprange or downrange until the correct range is found. For instruments with many ranges, a large change in signal will cause the instrument to scan through many ranges before finding the correct one. With smart ranging, the instrument will use the measured value on the incorrect range to try and determine what the correct range will be. If the measurement is really small, the instrument will skip ranges and try to downrange directly to the correct range. When upranging, the instrument will uprange once. If it is still on the wrong range it will go straight to its highest allowable range (source limits affect the highest allowable range). If this range is too large, it will use the smart method of downranging to the correct range. Note that as instrument technologies evolve, new instruments may actually use variations of this technique.

The other special ranging feature an instrument may have is sticky ranging. Often an instrument will be required to make many measurements on the same range. If the instrument started on the default range each time, say the highest allowable range, and went through its autorange algorithm for each requested measurement, the instrument will need to take extra measurements to get to the correct range each time. Sticky ranging causes the instrument to stay on the last range it was on. If the next measurement it needs to make will be on this range, the instrument will only need to take one measurement. This feature is most useful during sweeps where most of the measurements are all on the same range. Sticky ranging also coordinates with fixed ranging. When an instrument is put on a fixed range and then switched back to autorange, the instrument will start its autoranging on that range.

Another issue associated with range changes is settling time. The lowest ranges of an instrument could have significantly larger settling times than the higher ranges. Both sticky ranging and smart ranging help with this problem as well. However, sometimes one wants autoranged measurements but does not care about resolution once the signal falls below a certain value. When this is the case, having an autoranged measurement go down to the lowest range an instrument has wastes valuable time. The **lorangeX** function allows you to specify the lowest range an instrument will use while autoranging. The instrument will then give you the resolution you need without wasting any time trying to make a measurement on a more accurate range.

Often, autoranging measurements are not required. If the range which a measurement will be made on is known before the measurement is made, fixed ranging can be used. An instrument can be put on a fixed range with the **rangeX** function. Fixed range measurements can always be made faster than autoranging measurements. Care must be used when making fixed range measurements, however. If the range is set too low, the signal may be too large to measure on this range. If the signal is larger than the full scale of the range, the instrument is said to be overranged. When this happens, the instrument will return a special value to indicate the error instead of the actual measurement.

When an instrument is on a range lower than its compliance limit, it will limit at full scale of range. For example, a voltage source that is fixed on the 10µA current measurement range will limit at 10µA even though the compliance may be programmed to a larger value. Because a fixed range measurement will not automatically uprange, it cannot resolve this artificial compliance. This is known as range limit. This can affect measurements made on another instrument because the source is not forcing the value it was programmed to.The system will automatically resolve problems like this, but only for instruments that are on autorange.

Matrix operations

Most instruments require their terminals to be connected to a device under test (DUT) before they can be used. This involves the use of matrix commands. A typical test sequence consists of making connections, sourcing, measuring, then finally calling **devint** to restore the entire system, including the matrix, to its default condition.

The function most used to make connections is **conpin**. Normally, several **conpin** calls are made together at the beginning of a test sequence. These **conpin** calls grouped together are collectively called a **conpin** sequence. If you need to clear all matrix connections in the middle of a test sequence, you can call **clrcon** to do this explicitly; but if you start a new **conpin** sequence in the middle of a test sequence, **conpin** will automatically perform a **clrcon** before making any new connections.

If you need to make new connections or remove connections in the middle of a test sequence, **addcon** and **delcon** can be used. These functions do not clear the matrix like **conpin** does.

To prevent damage to matrix relays, all switching is done with sources off. **addcon**, **delcon**, and **clrcon** will all internally call **devclr** before opening or closing any relays. **devint** doesn't directly call **devclr** as described in the sourcing section above. It actually calls **clrcon** which in turn calls **devclr** as part of its normal processing.

Sweeping

LPTLib can automatically perform multi-parameter sweeps. Sweeps are much more efficient than programmatically changing the source value on an instrument and performing a set of individual measurements. To set up a sweep, the **smeasX**, **sintgX**, and **savgX** functions are used to populate what is called the measurement scan table. Each call to one of these functions adds an entry to the table. The **sweepX** function can then be used to step a source through a range of source values. At each step, a measurement will be performed for each entry in the measurement scan table.

Prior to sweeping the source, the source range is automatically set to the range appropriate for the largest source value in the sweep. This will avoid a source range change part way through the sweep. Note that when ranging, the absolute value of the source value is used to determine range. For example, if a voltage source is swept from -10.0V to +0.5V, the range appropriate for +/-10V (the 10.0V range) will be used.

After the sweep has completed, the measurement scan table is not cleared. If another sweep is performed, more measurements will again be made for each entry in the measurement scan table. The new data will be appended to the data from previous sweeps. Also note that calling **smeasX**, **sintgX**, or **savgX** also will NOT clear the measurement scan table. The new entries are simply added to the existing measurement scan table. The measurement scan table is automatically cleared when **devint** is called, but it can be explicitly cleared by calling **clrscn**.

Triggers

Several LPTLib functions make use of triggers. Triggers are simply boolean conditions, but to fully understand them you need to understand how LPTLib processes them. Triggers are registered with the system using the **trigXY** functions. This creates an entry in what is called the trigger table. During processing of various LPTLib commands the system evaluates the triggers. This is done by examining each entry in the trigger table to see if its condition is true or false. If any of the triggers are true, the trigger evaluation is true. Based on the evaluation of the triggers, the LPTLib function may alter how it continues processing.

The trigger table is cleared automatically when **devint** is called. Note that previous entries are NOT cleared by adding a new trigger. The new trigger is simply appended to the existing trigger table and it will be used along with all the rest the next time triggers are evaluated. This is a common mistake among even veteran LPTLib users. The trigger table can be cleared explicitly by calling **clrtrg**.

Most triggers are set up to monitor when a measurement on a particular instrument goes above or below a particular value. For example, **trigvg** (SMU1, 2.0) registers a trigger which is true when the voltage measured by SMU1 goes above 2.0 volts. When evaluating these types of triggers, the system does not monitor any measurements made by you, but instead requests a fresh reading from the triggering instrument which it compares to the threshold value. In other words, while evaluating triggers, the system will implicitly make a measurement for each entry in the trigger table for all entries of this type of trigger. The measurement is made using the **measX** function of the triggering instrument. The **setmode** function can be used to cause LPTLib to use the **intgX** or **avgX** function of the triggering instrument instead. In order to do this, all triggering instruments must support the **intgX** or **avgX** function, respectively.

These types of triggers use real numbers for the comparison. The **setmode** function can be used to make all triggers evaluate based on the absolute value of the measurement instead of the polarized measurement. This **setmode** option as well as the other triggering options mentioned above are cleared by the **clrtrg** function as well as the **devint** function.

GPIB

The **kibXXX** functions allow you to control GPIB instruments which do not have LPTLib drivers. Because the instruments controlled in this way do not have drivers which automatically interact with matrix control, care must be taken to ensure that all sources are cleared before any matrix operations are performed. Care must also be taken to ensure that the instruments are returned to their default state at the end of a test sequence. Two functions, **kibdefclr** and **kibdefint**, are provided to assist with this. These functions allow you to define strings which are sent to GPIB instruments any time the system is performing a **devclr** or **devint** operation respectively.

⚠ CAUTION

Failure to clear active GPIB based sources before a matrix operation is performed could result in damage to the matrix.

NOTE

Because of the slow speed of GPIB communication and the order instruments are cleared, it is not recommended to use GPIB instruments when performing bsweepX tests.

NOTE

Multiple GPIB interfaces are not currently supported. However, future systems may support multiple interfaces.

Optimizing test sequences

Fixed range vs. auto-range measurements

Use fixed-range measurements whenever possible. Depending upon the measurement, this can significantly increase the speed of the test sequence.

Fix-range trigger instruments

When you are using triggers, you should fix-range the triggering instrument with the value of the trigger. The trigger command forces a measurement to be taken after each sweep point and the measuring instrument will auto-range. Auto-ranging increases the total test execution time considerably.

Make use of "combination" commands

Do not use individual **forceX/measX** commands to sweep a set of points. Use the **sweepX** command which incurs significantly less overhead.

Synchronization

The S530 tester is designed to maximize throughput by decoupling the three tasks of sending LPT commands to the tester, executing them, and sending results back. The default mode of execution ensures that these three tasks are synchronized properly.

In this default mode of execution, a function which does not return results will return to the calling program before the command is executed on the tester. Those commands which return results will wait for the results before continuing. This is known as result synchronization. In this mode of execution, throughput is greatly increased while ensuring that return values are available at the completion of any function which returns a value.

Full synchronization will force LPT to wait for every command to completely finish executing before returning. This reduces throughput significantly because communication with the tester cannot be done in parallel with command execution.

Result synchronization, the default mode, allows communication with the tester to be done in parallel with test execution and allows LPT commands to be pre-queued in the tester to keep the executor busy without needing to wait for communications overhead. Although communications overhead is reduced, it is incurred for those commands which return values.

No synchronization allows commands which do return values to return before the results are actually received from the tester. If the result is not used immediately, this mode of synchronization will eliminate communication overhead associated with waiting for the results to be received. When the synchronization mode is set to NONE, it is your responsibility to resynchronize prior to using or returning any LPT results.

The **execut** and **inshld** commands both force synchronization with the tester and ensure that all results have been received and can be used. **execut** will also reset the synchronization mode back to the default mode SYNC_RESULT.

NOTE

The syncmode command itself does not resynchronize with the tester, it only sets the synchronization mode to be used by the next LPT command. **execut** or **inshld** must be used to resynchronize when required.

The KTE tools use result synchronization. If the synchronization mode is changed during execution of a userlib, it must be set back to SYNC_RESULT prior to exiting the userlib function. This can be done explicitly by calling **syncmode** or implicitly by calling **execut**. If the userlib function returns results, **execut** must be used to ensure all results have been received prior to exiting the function. Refer to the **syncmode** command for additional information.

```
double i1, i2;
syncmode(SYNC_NONE); /* don't wait */
/* for results */
. . . .
forcev(SMU1, 5.0);
forcev(SMU2, 1.0e-3);
measi(SMU1, &i1);
syncmode(SYNC_RESULT);
/* will cause measi to wait for results */
/* to be returned before continuing */
measi(SMU2, &i2);
```

Error handling

Error handling is done at two levels. The first is LPT function call return values. The other is error message processing performed by the KTE tools. For more information on how the KTE tools process error messages, see the documentation for the tool you are using.

Calling execut and inshld as a function

This is the simplest level of error handling available beyond the default. Calling **execut** or **inshld** as a function rather than as a subroutine allows you to detect errors during program execution, and take appropriate corrective actions. If an error is encountered during the execution of a test sequence, the return value of these functions is equal to the negative error code of the first error logged since the last **execut** command. A list of error codes is provided in the Error Definitions (on page 4-1) section of this manual.

Calling getlpterr

The function **getlpterr** can also be used to fetch error values. Using **getlpterr** is similar to **execut** or **inshld** but it returns the first error encountered since the last **devint**.

Error messages

There are two parts to an error message generated by the S530. These are the error header and the error text. An example error message looks like:

```
1997/03/01 12:00 - E0101
Argument #2 is not a pin in the current configuration.
```

In this example "1997/03/01 12:00 - E0101" is the error header and "Argument #2 is not a pin in the current configuration." is the text. The error header provides information about the error and the error text explains the cause of the error.

The first part of the header is the date and time the error occurred. Next is the letter "E" followed by the error number. In the example, the error number is 101. The return value of the **execut** call which followed the call that generated the error, would have returned the value -101.

Special error values returned

The LPT Library commands may return error values in place of actual measurement values. Table 4-1 summarizes these values. For example, the following KITT macro will generate a matrix error before trying to take a measurement.

```
conpin(SMU1, 999, 0);
measv(SMU1,V);
```

Rather than a true voltage reading, the actual value returned in the measv variable is 1E+23. This indicates that no measurement was taken due to an error; in this case, the matrix connection error.

Library structure

The LPT Library contains nine major sections as outlined below.

Matrix commands

`addcon`	Add connection
`clrcon`	Clear all connections
`conpin`	Connect a pin or instrument terminal
`conpth`	Connect path
`delcon`	Delete connection

Source commands

`forceX {i,v}`	Force a voltage or current
`limitX {i,v}`	Limit an instrument to a set voltage or current other than instruments default

Range commands

`lorangeX {i,v}`	Define lowest range an instrument should use during auto-range
`rangeX {c,i,v}`	Put a measuring instrument on a specific range
`setauto`	Re-enable auto-range mode

Measure commands

`avgX {c,cg,g,i,v}`	Averages measurements of voltage, current, conductance or capacitance
`intgX {c,cg,g,i,v}`	Integrate a measurement of voltage, current, conductance or capacitance
`measX {c,cg,g,i,v}`	Measure a voltage, current, conductance or capacitance
`ssmeasX {i,v}`	Steady state measurement

Combination commands

`asweepX {i,v}`	Sweep with a user-defined force array
`bmeasX {i,v}`	Block measurement — take a series of readings as quickly as possible
`bsweepX {i,v}`	Sweep and shutdown source if device meets trigger condition
`clrscn`	Clear any sweep measurements
`clrtrg`	Clear any set triggers
`rtfary`	Return forced array after sweep
`savgX {i,v}`	Average each point of an associated sweep
`searchX {i,v}`	Search for a specific voltage or current
`sintgX {i,v}`	Integrate each point of an associated sweep
`smeasX {i,t,v}`	Measure each point of an associated sweep
`sweepX {i,v}`	Sweep a specified range of current or voltage
`trigXg {i,t,v}`	Trigger if a measurement is greater than a specific value
`trigXl {i,t,v}`	Trigger if a measurement is less than a specific value

Execution/synchronization commands

`execut`	Enforce synchronization, verify that all previous commands are executed, and reset the instrumentation
`inshld`	Enforce synchronization and verify that all previous commands are executed
`syncmode`	Change the synchronization mode

General commands

`devint`	Initialize all devices
`devclr`	Clear all active devices
`getlpterr`	Get LPT error
`getstatus`	Return operating status of instrument
`setmode`	Set device-specific operating modes

Timing commands

`adelay`	Specify an array of delay points for use with asweep
`delay`	Delay during execution for a specified number of milliseconds
`disable`	Disable timer
`enable`	Enable timer
`imeast`	Return current time from timer
`rdelay`	Delay during execution for a specified number of seconds

GPIB commands

`kibdefclr`	Add device-clear commands to standard **devclr** command
`kibdefint`	Added device-initialization commands to standard **devint** command
`kibrcv`	Read a device-dependent GPIB string
`kibsnd`	Send a device-dependent GPIB string
`kibspl`	Serial poll an instrument
`kibsplw`	Synchronously serial poll an instrument

Section 3

LPT Library Function Reference

In this section:

Introduction 3-1
LPT functions 3-4

Introduction

The Keithley Linear Parametric Test Library (LPTLib) is a high-speed data acquisition and instrument control software library. It is the programmer's lowest level of command interface to the system's instrumentation. Its subroutines let the user configure the relay matrix and instrumentation to perform parametric tests.

This manual lists the subroutines included in the Keithley LPTLib and describes how to use them. The descriptions contained here follow the general pattern:

- A purpose and format of each argument.
- Remarks in which detailed information about the subroutine, along with the subroutine's placement and relationship to other subroutines in a test sequence are described.
- Examples showing a typical use of the subroutine in a test sequence.

A syncmode description for each of the commands in the LPT Library has been provided. This description will explain how each command responds to the different syncmode calls. As an example, look at how `conpin` reacts to the syncmode command:

If the syncmode parameter SYNC_NONE is called, LPTLib will respond to `conpin` by operating at normal parameters, since syncmode is not waiting for a result from `conpin` before continuing. The subroutine `conpin` does not have a result for the system before it can continue.

If the syncmode parameter SYNC_RESULT is called, LPTLib will respond to `conpin` as it did to the SYNC_NONE parameter, even though syncmode is telling the system to wait for a result before continuing. The subroutine `conpin` does not have a result to return to the system.

If the syncmode parameter SYNC_FULL is called, LPTLib will respond to `conpin` by waiting for the command to completely finish before moving to the next command. This is because syncmode is forcing LPTLib to wait for every command whether it returns a result or not.

Under the syncmode heading for the `conpin` command you would find this:

Syncmode: None = None, Result = None, Full = Full

Throughout this manual, the following conventions are used when explaining the subroutines:

- All LPTLib commands are case sensitive and must be entered as lower case when writing programs.
- A capital letter X shown in a command name indicates that the user must select from a list of replacement suffixes. Using the example `forceX`, the X can be replaced with either a v for voltage or i for current. The following is a table of possible suffixes, the parameter (function) each represents, and the units used throughout LPTLib for that parameter.

Suffix	Parameter	Unit
c	Capacitance	Farads
f	Frequency	Hertz
g	Conductance	Siemens
i	Current	Amperes
q	Charge	Coulombs
r	Resistance	Ohms
rh	Relative Humidity	Percent
temp	Temperature	Degree Celsius
t	Time	Seconds
v	Voltage	Volts

- **Brackets []** are used to enclose optional arguments of a subroutine.
- **Period strings (...)** indicate additional arguments or subroutines that can be added.
- **Periods (.)** indicate data not shown in an example because it is not necessary to help explain the specific subroutine.

The subroutine calls are presented in alphabetical order for ease of reference and are grouped by function in the next table.

Consolidated LPTLib subroutine listing	
Group	**Subroutine call**
Matrix	addcon (Add connection) clrcon (Clear connection) conpin (Connect pin) conpth (Connect path) delcon (Delete connection) tstsel (Test station select)
Ranging	lorangeX (Define lowest range, i,v) rangeX (Range c,i,v) setauto (Re-enable autorange)
Sourcing	forceX (Force i,v) limitX (Limit i,v)
Measuring	avgX (Average c,cg,g,i,v) bmeasX (Block measure i,v) intgX (Integrate c,cg,g,i,v) measX (Measure c,cg,g,i,v) setXmtr (i,v) ssmeasX (Steady state measure i,v)

Combination	asweepX (Array sweep i,v) bmeasX (Block measure i,v) bsweepX (Linear breakdown sweep i,v) clrscn (Clear scan) clrtrg (Clear trigger) rtfary (Return FORCE array) savgX (Sweep average i,v) searchX (Binary search i,v) sintgX (Sweep integrate i,v) smeasX (Measure i,v) sweepX (Linear sweep i,v) trigXg (Trigger if i,t,v is ³) trigXl (Trigger if i,t,v is <)
Timing	adelay (Array delay) delay (Delay in milliseconds) disable (TIMER) (Time measurement function) enable (TIMER) (Time measurement function) imeast (Immediate measure time) rdelay (Delay in seconds)
Execution/synchronizatio n	execut (Execute) inshld (Instrument hold) syncmode (Synchronization mode)
GPIB	kibdefclr (Clear instrument on devclr) kibdefint (Clear instrument on delay) kibrcv (Read device dependent string) kibsnd (Send device dependent command) kibspl (Serial poll an instrument) kibsplw (Synchronous serial poll)
General	devclr (Device clear) devint (Device initialize) getlpterr (Get LPT error) getstatus (Return operating status of instrument) setmode (Set operating mode)

LPT functions

addcon - Add connection

Purpose: Make new connections without clearing existing connections.

Format:

```
int addcon(int exist_connect, int connect2, [connectn, [...]] 0);
```

`exist_connect` A pin number or an instrument terminal id. This instrument or terminal may have been, but does not need to have been, previously connected with `addcon`, `conpin`, or `conpth`.

`connect2` A pin number or an instrument terminal id.

`connectn` A pin number or an instrument terminal id.

Syncmode: None = None, Result = None, Full = Full

Remarks: The S530 has an I/V matrix. `addcon` can be used to make additional connections on this matrix. I/V connections are direct connections.

When used with the I/V matrix, `addcon` will simply connect every item in the argument list together and there is no real distinction between `exist_connect` and the rest of the connection list. When using `addcon` with the I/V matrix, `addcon` behaves like `conpin` except previous connections are never cleared.

Prior to making the new connections, `addcon` will clear all active sources by calling `devclr`.

The value -1 will be ignored by `addcon` and is considered a valid entry in the connection list as well as for `exist_connect`. There must be at least two parameters with a value other than -1.

Matrix errors will be generated if the following condition is detected:

A dangerous connection is detected such as connecting an SMU high terminal directly to ground.

See also: clrcon (see "clrcon - Clear connections" on page 3-12), conpin (see "conpin - Connect pin" on page 3-16), conpth (see "conpth - Connect path" on page 3-17), delcon (see "delcon - Delete connection" on page 3-19)

adelay - Array delay

Purpose: Specifies an array of delay points to use with `asweepX` calls.

Format:

```
int adelay(unsigned int delaypoints, double *delayarray);
```

`delaypoints` The number of separate delay points defined in the array.

`delayarray` The name of the array defining the delay points. This is a single dimension floating point array that is "`delaypoints`" long and contains the individual delay times. Units of the delays are seconds.

Syncmode: None = None, Result = None, Full = Full

Remarks: Each delay in the array is added to the delay specified in `asweepX`. For example, if the array contained four delays (0.04 s, 0.05 s, 0.06 s, and 0.07 s) and the delay specified in `asweepX` is 0.1 s, then the resulting delays are (0.14 s, 0.15 s, 0.16 s, and 0.17 s).

asweepX - Array sweep

Purpose: Generates a waveform based on a user-defined forcing array (logarithmic sweep or other custom forcing functions).

Format:

```
int asweepi(int inst_id, unsigned int num_points, double delay_time, double
    *force_array);
int asweepv(int inst_id, unsigned int num_points, double delay_time, double
    *force_array);
```

`inst_id` The sourcing instrument's identification code.

`num_points` The number of separate current and voltage force points defined in the array.

`delay_time` The delay, in seconds, between each step and the measurements defined by the active measure list.

`force_array` The name of the user-defined force array. This is a single dimension array that contains all force points.

Syncmode: None = None, Result = Full, Full = Full

Remarks: `asweepX` is used with `smeasX`, `sintgX`, or `savgX` subroutines.

`trigXl` or `trigXg` can also be used with `asweepX`. However, once a trigger point is reached, the sourcing device stops moving through the array. The output is held at the last forced point for the duration of the `asweepX`. Data resulting from each step is stored in an array, as noted above, with `smeasX`. After the trigger point is reached, measurements are made at each subsequent point. Results are approximately equal since the source is held at a constant output.

`asweepv` and `asweepi` are sourcing-type subroutines. When called, an automatic limit is imposed on the sourcing device. Refer to the limit command for additional information.

The maximum number of times data is measured (using `smeasX`, `sintgX`, or `savgX`) is determined by the num_points argument in `asweepX`. A one-dimensional result array with the same number of data elements as the selected value of num_points must be defined in the test program.

When multiple calls to `asweepX` are executed in the same test sequence, the `smeasX`, `sintgX`, or `savgX` arrays are loaded sequentially. This appends the measurements from the second `asweepX` to the previous results. If the arrays are not dimensioned correctly, access violations will occur. The measurement table remains intact until `devint`, `clrscn`, or `execut` are executed.

Defining new test sequences using `smeasX`, `sintgX`, or `savgX` appends the command to the active measure list. Previous measures are still defined and will be used. `clrscn` is used to eliminate previous buffers for the second sweep. Using `smeasX`, `sintgX`, and `savgX` after a `clrscn` call will cause the appropriate new measures to be defined and used.

The following example gathers data to construct a graph showing the gain of a bipolar device over a wide range of base currents. A fixed collector-emitter bias is generated by SMU2. A logarithmic base current from 1.0E-10 to 1.0E-1A is generated by SMU1 using `asweepi`. The collector current applied by SMU2 is measured ten times by `smeasi`. The data gathered is then stored in the ICMEAS array.

Figure 1: S530 Bipolar device - wide range of base currents

```
double icmeas[10], ifrc[10];
.
.
ifrc[0]=1.0e-10;
for (i=1; i<10; i++)    /* Create decade array from */
   /* 1.0E-10 to 1.0E-1. */
    ifrc[i]=10.0*ifrc[i-1];
.
.
conpin(SMU1, 1, 0);     /* Base connection. */
conpin(SMU2, 2, 0);     /* Collector connection. */
conpin(GND, 3, 0);
limiti(SMU2, 200.0E-3);/* Reset I limit to maximum. */
smeasi(SMU2, icmeas);   /* Define collector current */
   /* array. */
forcev(SMU2, 5.0);      /* Force vce bias. */
asweepi(SMU1, 10, 10.0E-3, ifrc);    /* SweepIB, 10 points, 10ms */
   /* apart. */
execut();
```

avgX - Average

Purpose: Performs a series of measurements and averages the results.

Format:

```
int avgc(int inst_id, double *result, unsigned int stepno, double steptime);
int avgcg(int inst_id, double *capacitance, double *conductance, unsigned int
    stepno, double steptime);
int avgg(int inst_id, double *result, unsigned int stepno, double steptime);
int avgi(int inst_id, double *result, unsigned int stepno, double steptime);
int avgv(int inst_id, double *result, unsigned int stepno, double steptime);
```

`inst_id` The instrument identification code of the measuring instrument.

`result` The variable assigned the measurement's result.

`capacitance` The variable assigned the capacitance measurement.

`conductance` The variable assigned the conductance measurement.

`stepno` The number of steps averaged in the measurement. This number ranges from 1 through 32,767.

`steptime` The interval in seconds between each measurement.

Syncmode: None = None, Result = Result, Full = Full

Remarks: `avgX` is used primarily to obtain measurements where:

- The DUT being tested acts in an unstable manner.
- Electrical interference is higher than can be tolerated if `measX` were to be used.

The programmer specifies the number of samplings and the duration between each sampling.

After this subroutine executes, all closed relay matrix connections remain closed and the sources continue to generate voltage or current. This allows additional sequential measurements.

In general, measurement functions which return multiple results are more efficient than performing multiple measurement functions. For example, performing a single `avgcg` call is faster than performing `avgc` followed by `avgg`.

`rangeX` directly affects the operation of `avgX`. The use of `rangeX` prevents the instrument addressed from automatically changing ranges. This can result in an "overrange condition" such that would occur when measuring 10.0V on a 2.0V range. An overrange condition returns the value as the result of the measurement.

If `rangeX` is not located in the test sequence before the `avgX` call, the measurements performed automatically select the optimum range.

This example shows a test sequence used to measure the capacitance between a MOSFET gate and substrate. The capacitance returned is the result of ten measurements each separated by 2ms.

Figure 2: S530 Capacitance between MOSFET gate and substrate


```
double ciss;
.
.
conpin(CMTR1L, 3, 0);
conpin(CMTR1H, 2, 0);
rangec(CMTR1, 2.0E-12);/* Select range for 2.0pF. */
avgc(CMTR1, &ciss, 10, 2.0E-3);      /* Measure capacitance ten */
   /* times with 2ms between each;*/
   /* return average of results to*/
   /* ciss. */
execut();
```

bmeasX - Block measure

Purpose: Takes a series of readings as fast as possible. This measurement mode allows for waveform capture and analysis (within the resolution of the measurement instrument).

Format:

```
int bmeasi(int inst_id, double *results, unsigned int numrdg,  double delay, int
    timerid, double *timerdata);
int bmeasv(int inst_id, double *results, unsigned int numrdg, double delay, int
    timerid, double *timerdata);
```

`inst_id` The measuring instrument identification code.

`results` The result name of array to receive readings. The array must be large enough to hold readings.

`numrdg` The number of readings to return in the array.

`delay` The delay between points to wait (in seconds).

`timerid` The device name of the timer to use (0 = No timer data desired).

`timerdata` The array used to receive the time points at which the readings were taken. If `timerid` = 0, the timer is not read and this array is not updated. If used, the array must be large enough to hold readings.

Syncmode: None = None, Result = Result, Full = Full

Remarks: This subroutine collects data using the range presently selected. The measurement range is typically the same as the force range. If a different range is desired, you are required to place the instrument on the desired range prior to calling `bmeasX`.

When used with the Time Module, the measurements and the time when each measurement is performed are stored. The specific timer is defined in the subroutine, and the time array is returned with the result array.

Example: The example below shows how `bmeasX` is used with a timer. Each measurement is associated with a time stamp. This time stamp marks the interval when each reading is made. This information is useful when determining how much time was required to obtain a specific reading.

```
double irange, volts, rdng[5], timer[5];
.
.
enable(TIMER2);  /* Enable timer module. */
.
.
conpin(GND, 11, 0);     /* Make connections. */
conpin(SMU3, 14, 0);
.
.
forcev(SMU3, volts);    /* Perform test. */
measi(SMU3, &irange);   /* Set I range of SMU based */
rangei(SMU3, irange);   /* on initial measurement. */
.
forcev(SMU3, volts);
bmeasi(SMU3, rdng, 5, .0001,  /* Block I measurement of 5 */
 TIMER2, timer); /* readings using SMU3 with */
    /* 100ms delay between */
    /* readings, using TIMER2 with */
    /* time data labeled timer. */
```

```
execut();
```

Example: Using no timer.

This example shows how the **bmeasX** subroutine is used without a timer. When used without a timer, the returned measurement is not associated with a time stamp.

```
double volts, rdng [5];
.
conpin(GND, 11, 0);     /* Make connections. */
conpin(SMU3, 14, 0);
.
forcev(SMU3, volts);    /* Perform test. */
.
bmeasi(SMU3, rdng, 5, 0, 0, 0);      /* Block current measurement */
   /* of 5 readings using SMU3. */
execut();
```

bsweepX - Breakdown sweep

Purpose: Supplies a series of ascending or descending voltages or currents and shuts down the source when a trigger condition is encountered.

Format:

```
int bsweepi(int inst_id, double startval, double endval, unsigned int num_points,
   double delay_time, double *result);
int bsweepv(int inst_id, double startval, double endval, unsigned int num_points,
   double delay_time, double *result);
```

`inst_id` The sourcing instrument's identification code.

`startval` The initial voltage or current level applied as the first step in the sweep. This value can be positive or negative.

`endval` The final voltage or current level applied as the last step in the sweep. This value can be positive or negative.

`num_points` The number of separate current and voltage force points between `startval` and `endval`. The range may be from 1 through 8,000.

`delay_time` The delay in seconds between each step and the measurements defined by the active measure list.

`result` Assigned to the result of the trigger. This value represents the source value applied at the time of the trigger or breakdown.

Syncmode: None = None, Result = Full, Full = Full

Remarks: `bsweepX` is used with `trigXg` or `trigXl`. These trigger functions are used to provide the termination point for the sweep. At the time of trigger or breakdown, all sources are shut down to prevent damage to the device under test. Typically, this termination point is the test current required for a given breakdown voltage.

Once triggered, `bsweepX` terminates the sweep and clears all sources by executing a `devclr` command internally. The standard `sweepX` command will continue to force the last value. This is useful for device characterization curves but can cause problems when used in device breakdown conditions.

`bsweepX` can be used with `smeasX`, `sintgX`, `savgX`, or `rtfary` subroutines. Measurements are stored in a one-dimensional array in the consecutive order in which they were taken.

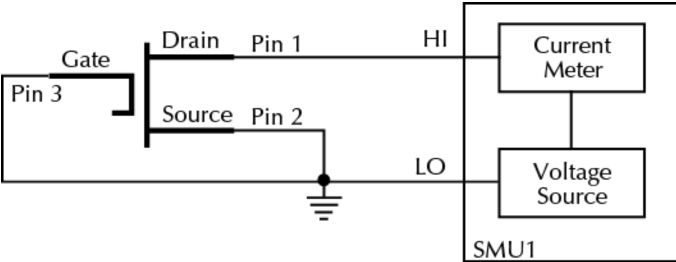
The system maintains a measurement scan table consisting of devices to test. This table is maintained using the `smeasX`, `sintgX`, `savgX`, or `clrscn` calls. As multiple calls to sweep functions are made, these commands are appended to the measurement scan table. Measurements are taken after the programmed delay_time is performed at the beginning of each `bsweepX` step.

When multiple calls to `bsweepX` are executed in the same test sequence, the arrays defined by `smeasX`, `sintgX`, or `savgX` calls are all loaded sequentially. This results from the second `bsweepX` call are appended to the results of the previous `bsweepX`. This can cause access violation errors if the arrays were not dimensioned for the absolute total. The measurement scan table remains intact until a `devint`, `clrscn`, or `execut` command completes.

Defining new test sequences using `smeasX`, `sintgX`, or `savgX` adds the command to the active measure list. The previous measures are still defined and used; however, previous measures for the second sweep can be eliminated by calling `clrscn`. New measures are defined and used by calling `smeasX`, `sintgX`, or `savgX` after `clrscn`.

Example: The following example measures the drain to source breakdown voltage of a FET. A linear voltage sweep is generated from 10.0 to 50.0V by SMU1 using the `bsweepv` subroutine. The breakdown current is set to 10mA by using `trigil`. The voltage at which this current is exceeded is stored in the variable BVDSS and is returned to the application processor by `execut`.

Figure 3: S530 Drain to source breakdown voltage




```
double bvdss;
.
.
conpin(SMU1, 1, 0);
conpin(GND, 2, 3, 0);
limiti(SMU1, 100.E-6);  /* Define I limit for device. */
rangei(SMU1, 100.E-6);  /* Select fixed range */
   /* measurement. */
trigil(SMU1, -10.E-6);  /* Set trigger point to 10mA. */
bsweepv(SMU1, 10.0, 50.0, 40, /* Sweep from 10 to 50V in 40 */
 10.0E-3, &bvdss);      /* steps with 10ms settling */
   /* time per step. */
execut();
```

clrcon - Clear connections

Purpose: Opens or de-energizes all DUT pin and instrument matrix relays, disconnecting all crosspoint connections. If any sources are actively generating current or voltage, `devclr` is automatically called before the relay matrix is de-energized.

Format:

```
int clrcon(void);
```

Syncmode: None = None, Result = None, Full = Full

Remarks: `clrcon` is called automatically by `devint` and `execut`. The first in a series of one or more connection type subroutines automatically calls a `clrcon`. Because this subroutine is automatically called, it is not normally used by a programmer.

clrscn - Clear scan table

Purpose: Clears the measurement scan tables associated with a sweep.

Format:

```
int clrscn(void);
```

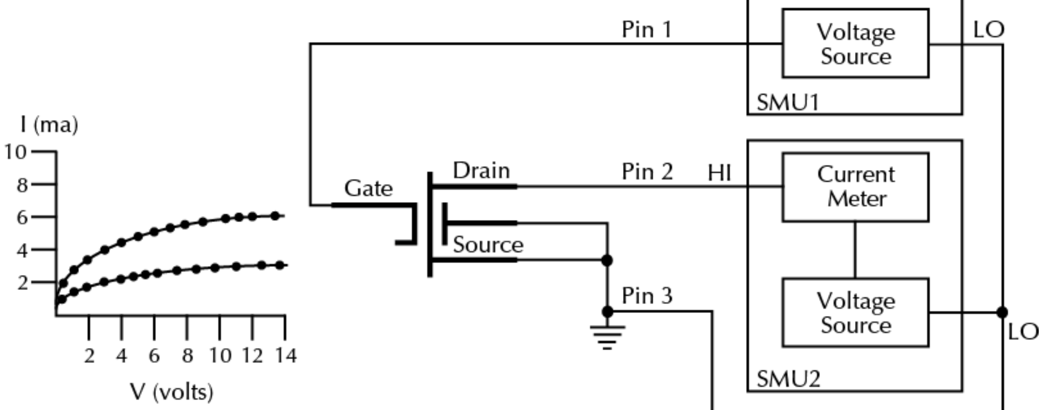
Syncmode: None = None, Result = None, Full = Full

Remarks: When a single `sweepX` is used in a test sequence, there is no need to program a `clrscn` since `execut` clears the table. `clrscn` is only required when multiple sweeps and multiple sweep measurements are used within a single test sequence.

Example: In the following example, `sweepX` configures SMU2 to source a voltage that sweeps from 0 through +14V in 14 steps. The results of the first `sweepv` are stored in an array called res1. Because of `clrscn`, the data and pointers associated with the first sweepv are cleared. Then 5V are forced to the gate, and the measurement process is repeated. Results from these second measurements are stored in an array called res2.

This example obtains the measurement data needed to construct a graph showing a FET's gate voltage-to-drain current characteristics. The program samples the current generated by SMU2 14 times. This is done in two phases: first with 4V applied to the gate, and second with 5V applied. The gate voltages are generated by SMU1.

Figure 4: S530 Gate voltage-to-drain current charecteristics



```
double res1 [14], res2 [14];
.
.
conpin(SMU1, 1, 0);
conpin(SMU2, 2, 0);
conpin(GND, 3, 0);
forcev(SMU1, 4.0);      /* Apply 4V to gate. */
smeasi(SMU2, res1);     /* Measure drain current in */
   /* each step; store results */
   /* in res1 array. */
sweepv(SMU2, 0.0, 14.0, 13,
 2.0E-2); /* Perform 14 measurements */
   /* over a range 0 through 14V. */
clrscn(); /* Clear smeasi. */
forcev(SMU1, 5.0);      /* Apply 5V to gate. */
smeasi(SMU2, res2);     /* Measure drain current in */
   /* each step; store results in */
   /* res2 array. */
sweepv(SMU2, 0.0, 14.0, 13,
 2.0E-2); /* Perform 14 measurements */
   /* over a range 0 through 14V. */
execut();
```

clrtrg - Clear trigger

Purpose: Clears the user-selected voltage or current level used to set trigger points. This permits the use of `trigXl` or `trigXg` more than once with different levels within a single test sequence.

Format:

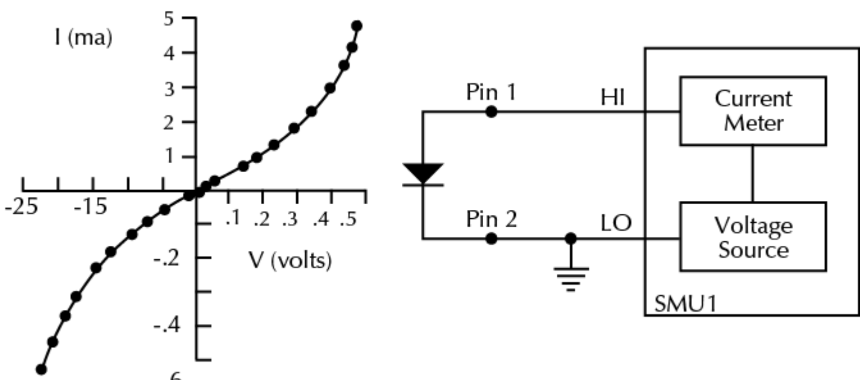
```
int clrtrg(void);
```

Syncmode: None = None, Result = None, Full = Full

Remarks: `searchX`, `sweepX`, `asweepX`, or `bsweepX`, each with different voltage or current levels, may be used repeatedly within a subroutine provided each is separated by a `clrtrg`.

Example: The following example collects data and constructs a graph showing the forward and reverse conduction characteristics of a diode. `clrtrg` allows `trigig` to be programmed twice in the same test sequence. Each result is returned to a separate array.

Figure 5: S530 Diode conduction characteristics



```
double forcur [11], revcur [11];     /* Defines arrays. */
.
.
conpin(SMU1, 1, 0);
conpin(GND, 2, 0);
trigig(SMU1, 5.0e-3);  /* Increase ramp to I = 5mA. */
smeasi(SMU1, forcur);  /* Measure forward */
   /* characteristics; */
   /* return results to forcur */
   /* array. */
sweepv(SMU1, 0.0, 0.5, 10,    /* Output 0 to 0.5V in 11 */
 5.0e-3); /* steps, each 5ms duration. */
clrtrg(); /* Clear 5mA trigger point. */
clrscn(); /* Clear sweepv. */
trigil(SMU1, -0.5e-3); /* Decrease ramp to I = -0.5mA. */
smeasi(SMU1, revcur);  /* Measure reverse */
   /* characteristics; */
   /* return results to revcur */
   /* array. */
sweepv(SMU1, 0.0, -30.0, 10,        /* Output 0 to -30V in 11 steps */
 5.00e-3);       /* each 5ms in duration. */
execut();
```

conpin - Connect pin

Purpose: Connect pins and instruments together.

Format:

```
int conpin(int connect1, int connect2, [connectn, [...]] 0);
```

`connect1` A pin number or an instrument terminal id.

`connect2` A pin number or an instrument terminal id.

`connectn` A pin number or an instrument terminal id.

Syncmode: None=None, Result=None, Full=Full

Remarks: The S530 has an I/V matrix. `conpin` is used to perform connections. I/V connections generally are direct connections.

When used with an I/V matrix, `conpin` will simply connect every item in the argument list together. Because I/V connections do not require pathways to be allocated, a pin or terminal may be used in more than one `conpin` call. As long as there are no connection rules violated, the pin or terminal will be connected to the additional items along with everything to which it is already connected.

The first `conpin` or `conpth` after any other LPT call will clear all sources by calling `devclr` and then clear all matrix connections by calling `clrcon` before making the new connections.

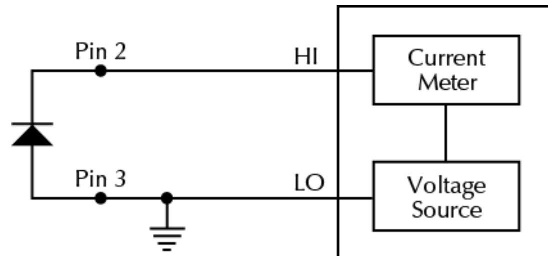
The value -1 will be ignored by `conpin` and is considered a valid entry in the connection list, however, there must be at least two entries in the list with a value other than -1.

Matrix errors will be generated when the following condition is detected:

- A dangerous connection is detected such as connecting an SMU high terminal directly to ground.

See also: addcon (see "addcon - Add connection" on page 3-4), clrcon (see "clrcon - Clear connections" on page 3-12), conpth (see "conpth - Connect path" on page 3-17), delcon (see "delcon - Delete connection" on page 3-19)

Figure 6: Conpin performs connections




```
conpin(3, GND, 0);      /* Connect pin 3 to GND */
   /* and ground. */
conpin(2, SMU1, 0);     /* Connect pin 2 to SMU1. */
.
.
execut();
```

conpth - Connect path

Purpose: Connect pins and instruments together using a specific pathway.

Format:

```
int conpth(int path, int connect1, int connect2, [connectn, [...]] 0);
```

`path` Pathway number to use for the connections.

`connect1` A pin number or an instrument terminal id.

`connect2` A pin number or an instrument terminal id.

`connectn` A pin number or an instrument terminal id.

Syncmode: None = None, Result = None, Full = Full

The S530 has an I/V matrix where I/V connections generally are direct connections.

The first `conpin` or `conpth` after any other LPT call will clear all sources by calling `devclr` and then clear all matrix connections by calling `clrcon` before making the new connections.

The value -1 for any item in the connection list will be ignored by `conpth` and is considered a valid entry in the connection list.

Matrix errors will be generated if the following condition is detected:

I/V connections are included in the connection list except as noted above.

See also: addcon (see "addcon - Add connection" on page 3-4), clrcon (see "clrcon - Clear connections" on page 3-12), conpin (see "conpin - Connect pin" on page 3-16), delcon (see "delcon - Delete connection" on page 3-19)

delay - Delay

Purpose: Provides a user-programmable dwell within a test sequence.

Format:

```
int delay(unsigned int n);
```

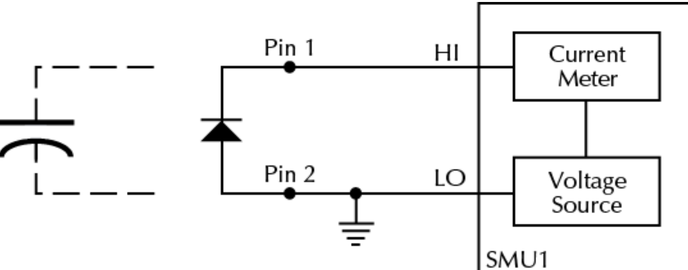
`n` The desired duration of the delay in milliseconds.

Snycmode: None = None, Result = None, Full = Full

Remarks: `delay` can be called anywhere within the test sequence.

Example: This example measures a variable capacitance diode's leakage current. SMU1 applies 60V across the diode. This device is always configured in the reverse bias mode, so the high side of SMU1 is connected to the cathode. This type of diode has very high capacitance and low leakage current; therefore, a 20ms delay is added. After the delay, current through SMU1 is measured and stored in the variable IR4.

Figure 7: S530 delay diode leakage current




```
double ir4;
.
.
conpin(SMU1, 1, 0);
conpin(GND, 2, 0);
forcev(SMU1, 60.0);     /* Generate 60V from SMU1. */
delay(20);      /* Pause for 20ms. */
measi(SMU1, &ir4);      /* Measure current; return */
   /* result to ir4. */
execut();
```

delcon - Delete connection

Purpose: Remove specific matrix connections.

Format:

```
int delcon(int exist_connect, [int exist_connectn, [...]] 0);
```

`exist_connect` A pin number or an instrument terminal id.

`exist_connectn` A pin number or an instrument terminal id.

Syncmode: None = None, Result = None, Full = Full

Remarks: All connections to each terminal or pin listed will be disconnected. Prior to making the disconnections, `delcon` will clear all active sources by calling devclr.

If GND is included in the list, all ground connections will be removed. SMU lows are hard-wired to ground.

A programmer can perform a series of tests within a single test sequence using `addcon` and `delcon` together without breaking existing connections. Only the required terminal and pin changes are made before performing the next sourcing and measuring operations.

See also: addcon (see "addcon - Add connection" on page 3-4), clrcon (see "clrcon - Clear connections" on page 3-12), conpin (see "conpin - Connect pin" on page 3-16), conpth (see "conpth - Connect path" on page 3-17)

Example:

```
double i1, i2;
conpin(3, 4, GND, 0);
conpin(1, SMU1, 0);
conpin(2, SMU2, 0);
forcev(SMU1, 1.0);
forcei(SMU2, .001);
measi(SMU1, &i1);
delcon(GND, 4, 0);      /* remove SMU2 from the circuit */
forcev(SMU1, 1.0);      /* because delcon cleared sources */
measi(SMU1, &i2);
```

devclr - Device clear

Purpose: Sets all sources to a zero state.

Format:

```
int devclr(void);
```

Syncmode: None = None, Result = None, Full = Full

Remarks: This function will clear all sources sequentially in the reverse order from which they were originally forced. Prior to clearing all Keithley supported instruments, GPIB based instruments will be cleared by sending all strings defined with `kibdefclr`.

`devclr` is implicitly called by `clrcon`, `devint`, and `execut`.

devint - Device initialize

Purpose: Resets the instruments and clears the system by opening all relays and disconnecting the pathways. Meters and sources are reset to the default states. Refer to the specific hardware manuals for listings of available ranges together with the default conditions and ranges for the instrumentation.

Format:

```
int devint(void);
```

Syncmode: None = None, Result = None, Full = Full

Remarks: This function will reset all instruments in the system to their default states.

This function will preform the following actions prior to resetting the instruments:

1. Clear all sources by calling `devclr`.
2. Clear the matrix cross-points by calling `clrcon`.
3. Clear the trigger tables by calling `clrtrg`.
4. Clear the sweep tables by calling `clrscn`.
5. Reset GPIB instruments by sending the string defined with kibdefint.

`devint` is implicitly called by `execut`.

disable - Disable timer

Purpose: Stops the timer and sets the time value to zero. Timer reading is also stopped.

Format:

```
int disable(int inst_id);
```

`inst_id` The instrument ID of timer module (`TIMERn`).

Syncmode: None = None, Result = None, Full = Full

Remarks: `disable`(`TIMERn`) stops the timer and resets the time value to zero.

enable - Initialize and start timer

Purpose: Provides correlation of real time to measurements of voltage, current, conductance, and capacitance.

Format:

```
int enable(int instr_id);
```

`instr_id` The instrument ID of timer module (`TIMERn`).

Syncmode: None = None, Result = None, Full = Full

Remarks: `enable`(`TIMERn`) initializes and starts the timer and allows other measurements to read the timer. The time starts at zero at the time of the enable call.

execut - Execute

Purpose: Causes system to wait for the preceding test sequence to be executed.

Format:

```
int execut(void);
```

Syncmode: None = Full, Result = Full, Full = Full

This function will wait for all previous LPT commands to complete and then will issue a `devint`. The synchronization mode will be set back to SYNC_RESULT.

After an `execut` call, ALL previous results are guaranteed to be valid regardless of synchronization mode.

forceX - Force a voltage or current

Purpose: Programs a sourcing instrument to generate a voltage or current at a specific level.

Format:

```
int forcei(int inst_id, double value);
int forcev(int inst_id, double value);
```

`inst_id` The instrument identification code. Refer to the instrument documents for the code.

`value` The level of the bipolar voltage or current forced in volts or amperes.

Syncmode: None = None, Result = None, Full = Full

Remarks: `forcev` and `forcei` generate either a positive or negative voltage as directed by the sign of the value argument. With both `forcev` and `forcei`:

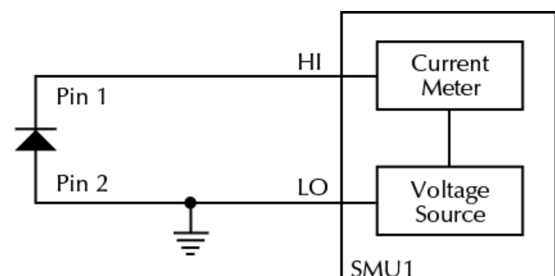
- Positive values generate positive voltage or current from the high terminal of the source relative to the low terminal.
- Negative values generate negative voltage or current from the high terminal of the source relative to the low terminal.

When using `limitX`, `rangeX`, and `forceX` on the same source at the same time in a test sequence, call `limitX` and `rangeX` before `forceX`.

The ranges of currents and voltages available from a voltage or current source vary with the specific instrument type. For more detailed information, refer to the specific hardware manual for each instrument.

Example: The reverse bias leakage of a diode is measured after applying 40.0V to the junction

Figure 8: diode reverse bias leakage



```
double ir12;
.
.
conpin(2, GND, 0);
conpin(SMU1, 1, 0);
limiti(SMU1, 2.0E-4);   /* Limit 1 to 200mA. */
forcev(SMU1, 40.0);     /* Apply 40.0V. */
measi(SMU1, &ir12);     /* Measure leakage; */
   /* return results to ir12. */
execut();
```

getlpterr - Get LPT error

Purpose: Get the first LPT error since the last `devint`.

Format:

```
int getlpterr(void);
```

Syncmode: None = Full, Result = Full, Full = Full

Remarks: This function will fetch the error code of the first error encountered since the last `devint` call.

Return value: Returns the first error code encountered since the last `devint`.

getstatus - Get instrument status

Purpose: Returns the operating state of the desired instrument.

Format:

```
int getstatus(int inst_id, unsigned int parameter, double *result);
```

`inst_id` The instrument identifier.

`parameter` The parameter of query.

`result` The data returned from the instrument. `getstatus` returns one item.

Syncmode: None = None, Result = Result, Full = Full

Remarks: `getstatus` returns instrument specific operating states.

imeast - Immediate measure

Purpose: Force a read of the timer and return the result.

Format:

```
int imeast(int inst_id, double *result);
```

`inst_id` The device ID.

`result` The variable assigned to the measurement.

Syncmode: None = None, Result = Result, Full = Full

Remarks: This command applies to all timers.

inshld - Instrument hold

Purpose: Wait for test execution to complete, but do not reset the state of instruments.

Format:

```
int inshld(void);
```

Syncmode: None = Full, Result = Full, Full = Full

Remarks: This function will wait for all previous LPT commands to complete. Unlike the `execut` command, instrumentation will not reset, and the synchronization mode will not change.

After an `inshld` call, ALL previous results are guaranteed to be valid regardless of synchronization mode.

intgX - Integrate

Purpose: Performs voltage or current measurements averaged over a user-defined period (usually, one AC line cycle). This averaging is done in hardware by integration of the analog measurement signal over a period of specified time. The integration is automatically corrected for 50 or 60Hz power mains.

Syncmode: None = None, Result = Result, Full = Full

Format:

```
int intgc(int inst_id, double *result);
int intgcg(int inst_id, double *capacitance, double *conductance);
int intgg(int inst_id, double *result);
int intgi(int inst_id, double *result);
int intgv(int inst_id, double *result);
```

`inst_id` The measuring identification code.

`result` The variable assigned the result of the measurement.

`capacitance` The variable assigned the capacitance measurement.

`conductance` The variable assigned the conductance measurement.

Remarks: The default integration time is one AC line cycle. This default time can be overriden with the `KI_INTGPLC` option of `setmode`. The `devint` command returns the integration time to the one AC line cycle default value.

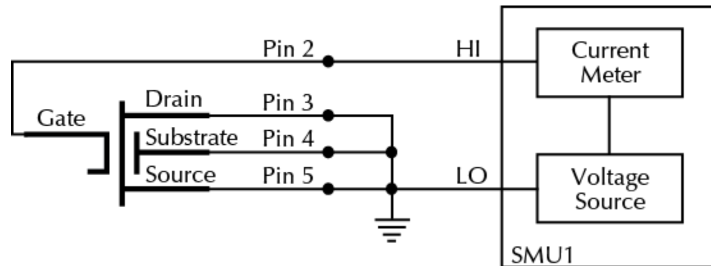
`rangeX` directly affects the operation of `intgX`. The use of `rangeX` prevents the instrument addressed from automatically changing ranges. This can result in an overrange condition that would occur when measuring 10.0V on a 2.0V range. An overrange condition returns the value as the measurement result.

If used, `rangeX` must be located in the test sequence before the associated `intgX` subroutine.

In general, measurement functions which return multiple results are more efficient than performing multiple measurement functions. For example, performing a single `intgcg` call is faster than performing `intgc` followed by `intgg`.

This example measures the relatively low leakage current of a MOSFET.

Figure 9: MOSFET current low leakage



```
double idss;
.
.
conpin(GND, 5, 4, 3, 0);
conpin(SMU1, 2, 0);
limiti(SMU1, 2.0E-8);   /* Limits to 20.0nA. */
rangei(SMU1, 2.0E-8);   /* Select range for 20.0nA */
forcev(SMU1, 25.0);     /* Apply 25V to the gate. */
intgi(SMU1, &idss);     /* Measure gate leakage; */
   /* return results to idss. */
execut();
```

kibdefclr - Keithley GPIB define device clear

Purpose: Defines a device dependent command sent to an instrument connected to the GPIB1 interface. This string is sent during any normal tester based `devclr`. It ensures that if the tester is calling `devclr` internally, any external GPIB device will be cleared with the given string.

Format:

```
int kibdefclr(int pri_addr, int sec_addr, unsigned int timeout, double delay,
   unsigned int snd_size, char *sndbuffer);
```

`pri_addr` The primary address of the instrument; numbers 0 through 30 are valid.

`sec_addr` The secondary address of the instrument; numbers 0 through 31 are valid. If the instrument device does not support secondary addressing, this parameter must be -1.

`timeout` The GPIB timeout for the transfer. This timeout is in 100ms units (i.e., timeout = 40 = 4.0s).

`delay` The time to wait after the device dependent string is sent to the device in seconds.

`snd_size` The number of bytes to send over the GPIB interface.

`sndbuffer` The physical byte buffer containing the data to send over the bus. This is the physical CLEAR string. A maximum of 1024 bytes are allowed.

Syncmode: None = None, Result = None, Full = Full

Remarks: Each call to `kibdefclr` copies parameters into a data structure within the tester memory. These data structures are allocated dynamically. These tables are cleared at `devint`. Any strings previously defined MUST be redefined.

The tester system allows you to define a maximum of 20 clear and 20 initialization strings. Each string may contain up to a maximum of 1024 bytes. Once defined, these strings remain in effect until the `execut` statement is processed.

Strings are sent over the GPIB interface in a first-in-first-out queue. This means that the first call to `kibdefclr` or `kibdefint` will be the first string sent over the GPIB. `devclr` (`kibdefclr`) strings are ALWAYS sent prior to initialization.

The KIBLIB `devclr` strings are sent PRIOR to `devclr` and `devint` execution. This may be a problem when communicating with any Keithley supported GPIB instruments. This may also have an impact on `bsweepX`, since `bsweepX` issues a `devclr` to clear active sources. It is not recommended to use GPIB instruments when performing `bsweepX` tests.

kibdefint - Keithley GPIB device initialize

Purpose: Defines a device dependent command sent to an instrument connected to the GPIB1 interface. This string is sent during any normal tester based `devint`. It ensures that if the tester is calling `devint` internally, any external GPIB device will now be initialized along with the rest of the known instruments.

Format:

```
int kibdefint(int pri_addr, int sec_addr, unsigned int timeout, double delay,
   unsigned int snd_size, char *snd_buff);
```

`pri_addr` The primary address of the instrument; 0 through 30 are valid.

`sec_addr` The secondary address of the instrument; 0 through 31 are valid. If the instrument device does not support secondary addressing, this parameter must be -1.

`timeout` The GPIB transfer timeout. This timeout is in l00ms units (i.e. timeout = 40 = 4.0s).

`delay` The time to wait after the device dependent string is sent to the device in seconds.

`snd_size` The number of bytes to send over the GPIB interface.

`snd_buff` The physical byte buffer containing the data to send over the bus. This is the INITIALIZE string. A maximum of 1024 bytes are allowed.

Syncmode: None = None, Result = Result, Full = Full

Remarks: Each call to `kibdefclr` and `kibdefint` copies parameters to a data structure within tester memory. These data structures are allocated dynamically. These tables are cleared at `devint`. Any strings previously defined MUST be redefined.

The tester system lets you define a maximum of 20 clear and 20 initialization strings. Each string may contain up to a maximum of 1024 bytes. Once defined, these strings remain in effect until the `execut` statement is executed.

Strings are sent over the GPIB in a first-in-first-out queue. This means that the first call to `kibdefclr` or `kibdefint` will be the first string sent over the GPIB interface. `devclr` (`kibdefclr`) strings are ALWAYS sent prior to initialization.

All `kiblib`, `devclr`, and `devint` strings are sent PRIOR to `devclr` and `devint` execution. This may be a problem when communicating with any Keithley supported GPIB instruments. This may also have an impact on `bsweepX`, since `bsweepX` issues a `devclr` to clear ALL active sources. It is not recommended to use GPIB instruments when performing `bsweepX` tests.

kibrcv - Keithley GPIB receive

Purpose: Used to read a device dependent string from an instrument connected to the GPIB interface.

Format:

```
int kibrcv(int pri_addr, int sec_addr, char term, unsigned int timeout, unsigned
   int rcv_size, int *rcv_len, char *rcv_buff);
```

`pri_addr` The primary address of the instrument; 0 through 30 are valid.

`sec_addr` The secondary address of the instrument; 0 through 31 are valid. If the instrument device does not support secondary addressing, this parameter must be -1.

`term` The ASCII terminator character of the returned string. This is the byte used for terminating data buffer read.

`timeout` The GPIB transfer timeout. This timeout is in 100ms units (i.e., timeout = 40 = 4.0s).

`rcv_size` The physical receive buffer size. This is the maximum number of bytes that can be read from the device.

`rcv_len` The number of bytes that are read from the device on the GPIB interface. This variable is returned by the tester after all bytes are read from the device.

`rcv_buff` The physical BYTE buffer destined to receive the data from the device connected to the GPIB interface.

Syncmode: None = None, Result = Full, Full = Full

Remarks: `kibrcv` receives a buffer from the GPIB interface by performing the following steps:

1. Assert attention (ATN).
2. Send my LISTEN address.
3. Send device TALK address.
4. Send secondary address (if not -1).
5. De-assert ATN.
6. Read byte array from device into `rcv_buff` until end-or-identify (EOI) or the terminator is received.
7. Assert ATN.
8. Send UNTalk (UNT).
9. Send UNListen (UNL).
10. De-assert ATN.

`rcv_size` defines the maximum number of bytes physically allowed in the buffer. If `rcv_size` is less than the byte string returned by the instrument, then the device is short-cycled and only the maximum number of bytes are returned.

kibsnd - Keithley GPIB send

Purpose: Sends a device dependent command to an instrument connected to the GPIB interface.

Format:

```
int kibsnd(int pri_addr, int sec_addr, unsigned int timeout, unsigned int send_len,
   char *send_buff);
```

`pri_addr` The primary address of the instrument; 0 through 30 are valid.

`sec_addr` The secondary address of the instrument; 0 through 31 are valid. If the instrument device does not support secondary addressing, this parameter must be -1.

`timeout` The GPIB transfer timeout. This timeout is in 100ms units (for example, timeout = 40 = 4.0s).

`send_len` The number of bytes to send over the GPIB interface.

`send_buff` The physical byte buffer containing the data to send over the bus.

Syncmode: None = None, Result = None, Full = Full

Remarks: `kibsnd` sends a buffer out the GPIB interface by performing the following steps:

1. Assert attention (ATN).
2. Send device LISTEN address.
3. Send secondary address (if not -1).
4. Send my TALK address.
5. De-assert ATN.
6. Send `send_buff` with end-or-identify (EOI) asserted with the last byte.
7. Assert ATN.
8. Send UNTalk (UNT).
9. Send UNListen (UNL).
10. De-assert ATN.

kibspl - Keithley GPIB serial poll

Purpose: Serial polls an instrument connected to the GPIB interface.

Format:

```
int kibspl(int pri_addr, int sec_addr, unsigned int timeout, int
   *serial_poll_byte);
```

`pri_addr` The primary address of the instrument; 0 through 30 are valid.

`sec_addr` The secondary address of the instrument; 0 through 31 are valid. If the instrument device does not support secondary addressing, this parameter must be -1.

`timeout` The GPIB polling timeout. This timeout is in l00ms units (i.e., timeout = 40 = 4.0s).

`serial_poll_byte` The variable name for the serial poll byte returned by the current.

Syncmode: None = None, Result = Result, Full = Full

Remarks: `kibspl` performs the following steps:

1. Assert attention (ATN).
2. Send serial poll enable (SPE).
3. Send LISTEN address.
4. Send device TALK address.
5. Send secondary address (if not -1).
6. De-assert ATN.
7. Read serial_poll_byte from the instrument.
8. Assert ATN.
9. Send serial poll disable (SPD).
10. Send UNTalk (UNT).
11. Send UNListen (UNL).
12. De-assert ATN.

The `serial_poll_byte` variable must be four bytes long.

kibsplw - Keithley GPIB

Purpose: Used to synchronously serial poll an instrument connected to the GPIB interface. This command waits for SRQ to be asserted on the GPIB by any device. After SRQ is asserted, a serial poll sequence is initiated for the device and the serial poll byte is returned.

Format:

```
int kibsplw(int pri_addr, int sec_addr, unsigned int timeout, int
    *serial_poll_byte);
```

`pri_addr` The primary address of the instrument; 0 through 30 are valid.

`sec_addr` The secondary address of the instrument; 0 through 31 are valid. If the instrument device does not support secondary addressing, this parameter must be -1.

`timeout` The GPIB polling timeout. The timeout is in l00ms units (i.e., timeout = 40 = 4.0s).

`serial_poll_byte` The serial poll byte variable name returned by the device presently being polled.

Syncmode: None = None, Result = Result, Full = Full

Remarks: `kibsplw` performs the following steps:

1. Wait with timeout for general SRQ assertion on the GPIB.
2. Call `kibspl`.

limitX - Limit a voltage or current

Purpose: Allows the programmer to specify a current or voltage limit other than the instrument's default limit.

Format:

```
int limiti(int instr_id, double limit_val);
int limitv(int instr_id, double limit_val);
```

`inst_id` The instrument identification code of the instrument on which to impose a source value limit.

`limit_val` The maximum level of the current or voltage. The value is bidirectional. For example, a `limitv` (SMU1, 10.0) limits the voltage of the current source SMU1 to ±10.0V. A `limiti` (SMU1, 1.5E-3) limits the current of the voltage source SMU1 to ±l.5mA.

Syncmode: None = None, Result = None, Full = Full

Remarks: Use `limiti` to limit the current of a voltage source. Use `limitv` to limit the voltage of a current source.

NOTE

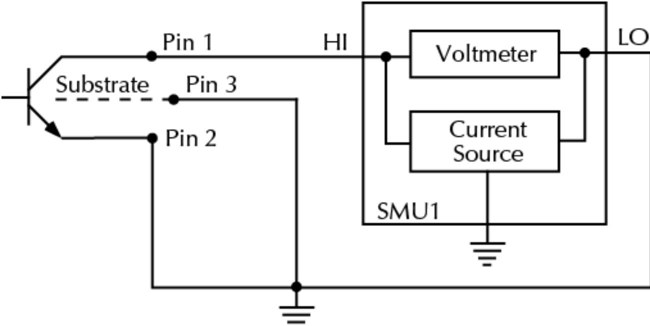
If the instrument is ranged below the programmed limit value, the instrument will temporarily limit to full scale of range.

This subroutine must be called in the test sequence before the associated `forceX`, `sweepX`, or `searchX` subroutine is used to generate the voltage or current. `limitX` also sets the top measurement range of an autoranged measurement.

The limits set within a particular test sequence are cleared when `devint` or `execut` are called.

This example measures the breakdown voltage of a device. The limit is set at 150.0V. This limit is necessary to override the default limit of the SMU, which would otherwise be in effect

Figure 10: SMU breakdown voltage




```
double ibceo, vbceo;
.
.
conpin(2, 3, GND, 0);
conpin(SMU1, 1, 0);
limitv(SMU1, 150.0);    /* Limit voltage at 150V. */
forcei(SMU1, ibceo);    /* Force current through DUT. */
measv(SMU1, &vbceo);    /* Measure breakdown voltage; */
.   /* return results to vbceo. */
.
execut();
```

lorangeX - Select bottom range

Purpose: Defines the bottom autorange limit.

Format:

```
int lorangei(int inst_id, double range);
int lorangev(int inst_id, double range);
```

`inst_id` The instrument identification code.

`range` The value of the desired instrument range in volts, amperes, farads, or siemens.

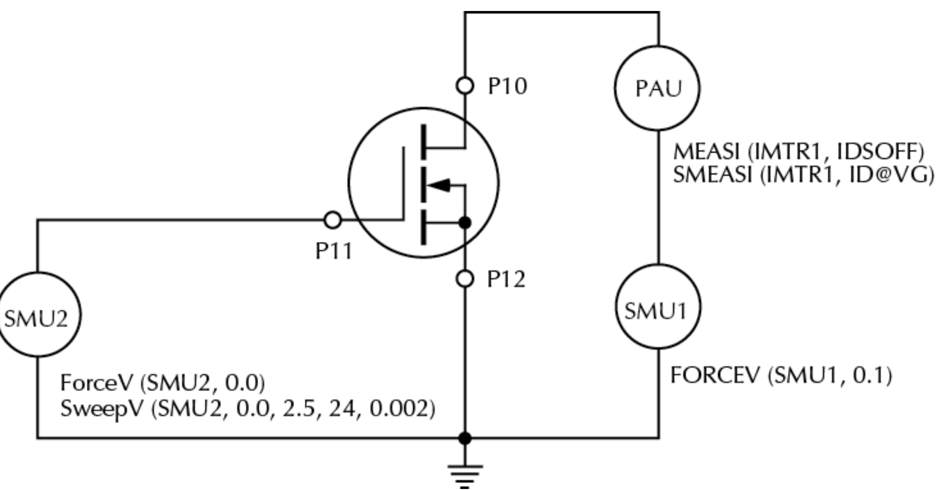
Syncmode: None = None, Result = None, Full = Full

Remarks: `lorangeX` is used with autoranging to limit the number of range changes and thus saves test time.

If the instrument was on a range lower than the one specified by `lorangeX`, the range is changed. S530 automatically provides any range change settling delay that may be necessary due to this potential range change.

Once defined, `lorangeX` is in effect until a `devclr`, `devint`, `execut`, or another `lorangeX` executes.

Figure 11: Range change settling delay




```
double idatvg [25];
:
conpin(SMU1, 10, 0);
conpin(SMU2, 11, 0):
conpin(12, GND, 0);
lorangei(SMU1, 2.0E-6);/* Select 2mA as minimum */
   /* range during auto-ranging. */
smeasi(SMU1, idatvg);  /* Setup sweep measurement */
   /* of IDS. */
sweepv(SMU2, 0.0, 2.5, 24,
    0.002);      /* Sweep gate from 0 to */
                 /* 2.5V. */
execut();
```

measX - Measure

Purpose: Allows the measurement of voltage, current, charge capacitance, or conductance.

Format:

```
int measc(int inst_id, double *result);
int meascg(int inst_id, double *capacitance, double *conductance);
int measg(int inst_id, double *result);
int measi(int inst_id, double *result);
int measv(int inst_id, double *result);
```

`inst_id` The instrument identification code.

`result` The variable assigned to the result of the measurement.

`capacitance` The variable assigned to the capacitance of the measurement.

`conductance` The variable assigned to the conductance of the measurement.

Syncmode: None = None, Result = Result, Full = Full

Remarks: After the subroutine is called, all relay matrix connections remain closed, and the sources continue to generate voltage or current. For this reason, two or more measurements can be made in sequence.

`rangeX` directly affects the operation of the `measX` subroutine. The use of `rangeX` prevents the instrument addressed from automatically changing ranges when `measX` is called. This can result in an overrange condition such that would occur when measuring 10.0V on a 2.0V range. An overrange condition returns the value as the result of the measurement.

If used, `rangeX` must be located in the test sequence before the associated `measX` subroutine.

All measurements except meast invoke a timer snapshot measurement to be taken by all enabled timers. This timer snapshot can then be read with the meast command.

In general, measurement functions which return multiple results are more efficient than performing multiple measurement functions. For example, performing a single `meascg` call is faster than performing `measc` followed by `measg`.

Figure 12: Diode voltage forward bias

```
double if46, vf47;
.
.
if46 = 50e-3;
.
.
conpin(3, GND, 0);
conpin(SMU1, 2, 0);
forcei(SMU1, if46);    /* Forward bias the diode; */
   /* set SMU current */
   /* limit to 50mA. */
measv(SMU1, &vf47);    /* Measure forward bias; */
   /* return result to vf47. */
execut();
```

mpulse - Measure pulse

Purpose: This subroutine forces a voltage pulse and measures both the voltage and current for exact device loading.

Format:

```
mpulse (long inst_id, doube pulse_amplitude, double pulse_duration, double* vmeas,
    double* i_meas);
```

`inst_id` The name of instrument under control

`pulse_amplitude` The pulse height in volts

`pulse_duration` The pulse width in seconds. The measurements will be taken at the end of the pulse before the `mpulse` is shutdown.

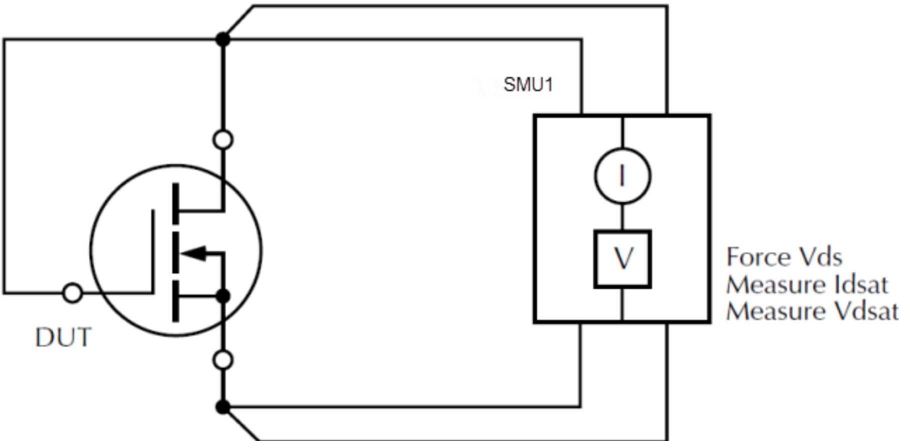
`v_meas` The variable used to receive the voltage on the output of the `SMU` at the time the pulse terminates. This reading is buffered internally until an `execut` or `inshld` is issued.

`i_meas` The variable used to receive the current drawn from the `SMU`. This measurement is taken simultaneously with the voltage, so the combined values are an exact representation of the device load at pulse termination.

Remarks: Voltage and current are measured just before the pulse terminates. Pulse mode is used because the device under test will be destroyed if the voltage is applied for a long period of time as with GaAs type devices or high-power bipolar.

Example: The following example measures the drain current of a MOSFET when Vds equals Vgs. A voltage pulse, Vds, is applied to the drain. The pulse duration is 1 millisecond. Voltage across the MOS transistor, Vdsat, and drain current, Idsat, are measured.

Figure 13: MOSFET drain current measurement




```
.
.
mpulse(SMU1, Vds, 1.0E-3, Vdsat, Idsat)
execut()
```

pulseX - Voltage or current pulse

Purpose: This subroutine directs a sourcing instrument to generate a voltage or current at a specific level for a predetermined length of time.

Format:

```
pulseX(long inst_id, double forceval, double time);
```

`inst_id` The instrument identification code. This may be SMUn with a `pulseV` or `pulseI`.

`forceval` The level of voltage or current in volts or amperes, respectively, to be forced. The value can be positive or negative. For example, a `pulseV` (SMU1, 10.0, time) generates +10.0 V, and a `pulseI` (SMU1, -1.5E-3, time) generates -1.5 mA.

`time` The pulse duration in seconds. For example, a time of 0.5 initiates a time of 0.5 seconds, and a time of 2.0E-2 initiates a time of 20 milliseconds. The minimum practical time for a SMU source is dependent on the voltage or current level being sourced and the impedance of the DUT.

The ranges of current and voltage available vary with the specific instrument type. For more detailed information, refer to the specific hardware manuals.

Remarks: Use `x` = v to generate a voltage pulse or `i` to generate a current pulse.

`pulseV` and `pulseI` generate either a positive or negative voltage as directed by the sign of the value argument. With both the `pulseV` and `pulseI` subroutines:

- A positive value generates a positive voltage from the high terminal of the source.
- A negative value generates a negative voltage from the high terminal of the source.

After `pulseX` is run, other subroutines may be used to perform measurements. `measX` can measure:

- Residual voltage or current as it decays after removal of the initial application.
- Capacitance between DUT pins as the residual voltage or current decays.

All measurements performed using the `pulseX` and `measX` functions are performed after the pulse has completed.

Whenever `pulseX` is run, either a default or a programmed current or voltage limit is in effect. The type of limit depends on the type of `pulseX` subroutine, as follows:

- `pulseV`, has an automatic current limit default. For example, an SMU used as a voltage source defaults to a current limit of 10.0 mA. `limitI` called before `pulseV` can be used to override the default.
- `pulseI`, has a an automatic voltage limit default. For example, an SMU used as a current source defaults to a voltage limit of 20.0 V. `limitV` called before `pulseI` can be used to override the default.

When using `limitX` and `pulseX` on the same source at the same time in a test sequence, call `limitX`, then `pulseX`.

Example: Here the threshold voltage shift of a FET is measured by performing two `searchV` subroutines, as follows:

1. `searchV` measures the gate voltage required to initiate a drain current of 10 µA.
2. `searchV` measure the gate voltage required to initiate a drain current of 10 µA immediately after a 20 V pulse is applied to the gate.

Note that the second `searchV` was called without reprogramming `trigIl`. This is possible since `clrtrg` was not used.

Figure 14: FET threshold voltage


```
float res1, res2;
.
.
conpin(GND, 2, 3, 0);
conpin(SMU1, 1, 0)
conpin(SMU2, 4, 0)
forcev(SMU1, .5)
trigil(SMU1, -1.E-5) /* Set the trigger point for 10mA. */
searchv(SMU2, 0.0, 3.0, 7, /* Increase voltage until trigger */
2.0E-5, &res1) /* point occurs. Return results to
res1. */
pulsev(SMU2, 20.0, 5.E-4) /* Apply a 20V pulse to the */
/* gate for 500ms. */
searchv(SMU2, 0.0, 3.0, 7, /* Increase voltage until trigger */
2.0E-5, &res2) /* point occurs. Return results */
/* to res2. */
execut():
```

rangeX - Select range

Purpose: Selects a range and prevents the selected instrument from autoranging. By selecting a range, the time required for autoranging is eliminated.

Format:

```
int rangec(int inst_id, double range);
int rangei(int inst_id, double range);
int rangev(int inst_id, double range);
```

`inst_id` The instrument identification code.

`range` The value of the highest measurement to be taken. The most appropriate range for this measurement will be selected. If range is set to 0, the instrument will autorange.

Syncmode: None = None, Result = None, Full = Full

Remarks: `rangeX` is primarily intended to eliminate the time required by the automatic range selection performed by a measuring instrument. Because `rangeX` will prevent autoranging, an overrange condition can occur, for example, measuring 10.0V on a 2.0V range. The value is returned when this occurs.

`rangeX` can also reference a source since an SMU can simultaneously be:

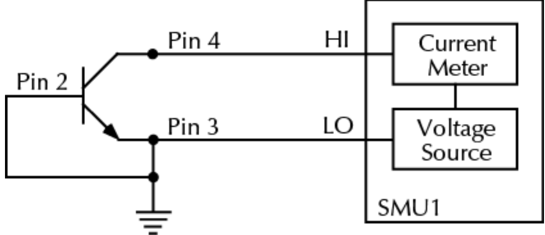
- A voltage source, voltmeter, and current meter, or
- A current source, current meter, and voltmeter.

The range of a SMU is the same for the sourcing function and the measuring function.

When selecting a range below the limit value, whether it is explicitly programmed or the default value, an instrument will temporarily use the full scale value of the range as the limit. This will not change the programmed limit value and, if the instrument range is restored to a value higher than the programmed limit value, the instrument will again use the programmed limit value.

When changing the instrument range, be careful not to overrange the instrument. For example, a test initially performed in the 10mA range with a 5mA limit is changed to test in the 1mA range with a 1mA limit. Notice that the limit is lowered from 5mA to 1mA to avoid overranging the 1mA setting.

Figure 15: SMU measure and source range function.png




```
double icer2;
.
.
conpin(3, 2, GND, 0);
conpin(SMU1, 4, 0);
limiti(SMU1, 1.0E-3);   /* Limit current to 1.0mA. */
rangei(SMU1, 2.0E-3);   /* Select range for 2mA. */
forcev(SMU1, 35.0);     /* Force 35V. */
measi(SMU1, &icer2);    /* Measure leakage; return */
    /* results to icer2. */
execut();
```

rdelay - Real delay

Purpose: A user-programmable delay in seconds.

Format:

```
int rdelay(double n);
```

`n` The desired delay duration in seconds.

Syncmode: None = None, Result = None, Full = Full

Example: The following example measures a variable capacitance diode's leakage current. SMU1 presets 60V across the diode. The device is configured in reverse bias mode with the high side of SMU1 connected to the cathode. This type of diode has high capacitance and low leakage current. Therefore, a 20ms delay is added. After the delay, current through SMU1 is measured and stored in the variable IR4.

```
double ir4;
.
.
conpin(SMU1, 1, 0);
conpin(GND, 2, 0);
forcev(SMU1, 60.0);     /* Generate 60V from SMU1. */
rdelay(0.02);    /* Pause for 20ms. */
measi(SMU1, &ir4);      /* Measure current; return */
   /* result to ir4. */
execut();
```

rtfary - Return force array

Purpose: Returns the force array determined by the instrument action. This eliminates the need to calculate the forced array in the application.

Format:

```
int rtfary(double *result);
```

`result` The floating point array where the force values will be stored.

Syncmode: None = None, Result = None, Full = Full

Remarks: This function, when used in conjunction with one of the sweep routines, lets the user determine the exact forced value for each point in the sweep.

When the test sequence is executed, the sweep subroutine initiates the first step of the voltage or current sweep. The sweep then logs the force point that the buffer specified by `rtfary`.

Locate `rtfary` before the sweep. The number of data points returned by `rtfary` is determined by the number of force points generated by the sweep.

savgX - Sweep average

Purpose: Performs an averaging measurement for every point in a sweep.

Format:

```
int savgi(int instr_id, double *results, unsigned int count, double delay);
int savgv(int instr_id, double *results, unsigned int count, double delay);
```

`instr_id` The measuring instrument's identification code.

`results` The floating point array where the results are stored.

`count` The number of measurements made at each point before the average is computed.

`delay` The time delay in seconds between each measurement within a given ramp step.

Syncmode: None = None, Result = None, Full = Full

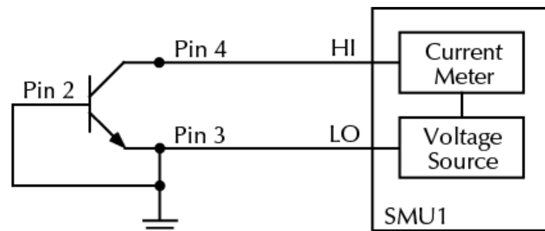
Remarks: This function is used to create an entry in the measurement scan table. During any of the sweeping functions, a measurement scan is performed for every force point in the sweep. During each scan, a measurement will be made for every entry in the scan table. The measurements are made in the same order which the entries were made in the scan table.

`savgX` sets up the new scan table entry to perform an averaging measurement. The measurement results will be stored in the array specified by results. Each time a measurement scan is made, a new measurement result will be stored at the next location in the results array. If the scan table is not cleared, performing multiple sweeps will simply keep adding new measurement results to the end of the array. Care must be taken to be sure the results array is large enough to hold all measurements which will be taken before the scan table is cleared. The scan table is cleared by an explicit call to `clrscn` or implicitly when devint or `execut` is called.

When making each averaged measurement, *`count`* actual measurements will be made on the instrument *`delay`* seconds apart and the average calculated. This average is the value which will be stored in the results array.

Example: This example obtains the measurement data needed to construct a graph showing the capacitance-versus-voltage characteristics of a variable capacitance diode. This diode is operated in reverse biased mode. SMU1 outputs a voltage that sweeps from 0 through -50V. Capacitance is measured 26 times during the sweep. The results are stored in an array called `res1`.

Figure 16: Variable capcitance diode measurement



```
double res1 [26];
.
.
conpin(3, 2, GND, 0);
conpin(SMU1, 4, 0);
savgi(SMU1, res1, 8, 1.0E-3); /* Measure average */
   /* current 8 times per */
   /* sample; return results to */
   /* res1 array. */
sweepv(SMU1, 0.0, -50.0, 25,
    2.0E-2);    /* Generate a voltage from 0 */
   /* to -50V over 26 steps.*/
execut();
```

searchX - Binary search measurement

Purpose: Used to determine the voltage or current required to obtain a desired current, voltage, capacitance, or conductance. It is useful in finding initial threshold points such as junction breakdown or transistor turn on.

Format:

```
int searchi(int inst_id, double min_val, double max_val, unsigned int iterate_no,
   double iterate_time, double *result);
int searchv(int inst_id, double min_val, double max_val, unsigned int iterate_no,
   double iterate_time, double *result);
```

`inst_id` The sourcing instrument's identification code.

`min_val` The lower limit of the source range.

`max_val` The upper limit of the source range.

`iterat_no` The number of separate current or voltage levels to generate. The range of iterations is from 1 through 16.

`iterate_time` The duration, in seconds, of each iteration.

`result` The floating point variable assigned to the search operation result. It represents the voltage, with searchv, or current, with searchi, applied during the last search operation.

Syncmode: None = None, Result = Full, Full = Full

Remarks: `trigXg` or `trigXl` must be used with *search*. Triggers and *search* together initiate a search operation consisting of a series of steps referred to as iterations. During each iteration, the following events occur:

A voltage or current is applied to a circuit node of the DUT.

All triggers are evaluated.

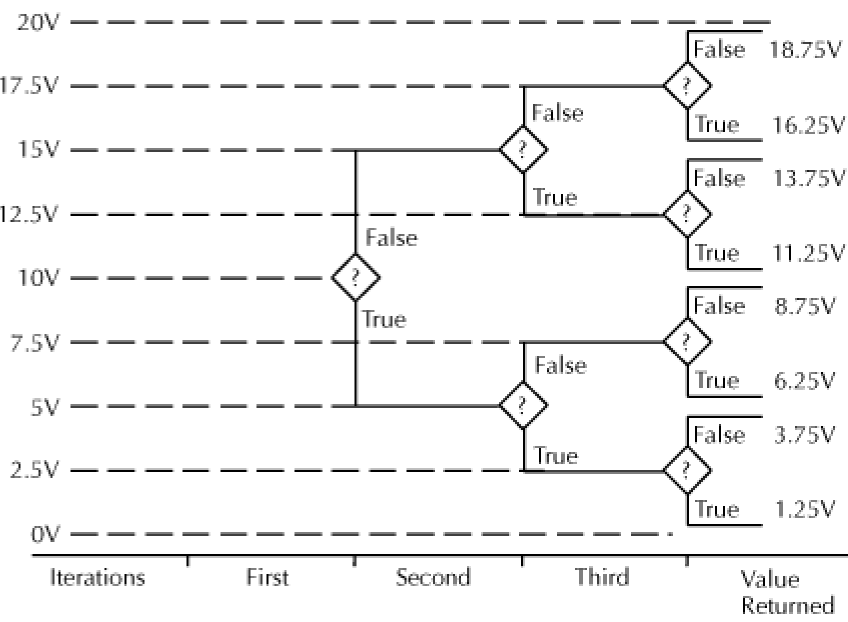
If the triggers evaluate true, the source value is moved toward the `min_val`. Otherwise, the source value is moved toward `max_val`. The source range is then divided in half for the next iteration.

Up to 16 iterations can be programmed. When all iterations are completed, a value of voltage or current is returned as the result of the search operation. This value is the voltage or current level required to match the trigger point.

The following example shows all binary search possibilities where the minimum and maximum source values are 0 and 20V, respectively. Study the example and note the following:

- Three iterations, numbered one through three, are shown. Within a given iteration, the values of possible sourcing voltages are indicated.
- During the first iteration of the binary search process, 10V are applied. This represents the midpoint of the minimum and maximum values.
- At the end of each iteration, the program determines whether to increase or decrease the source voltage. The determination is dependent on the evaluation of the trigger point

Figure 17: Minimum and maximum source values



The question mark (?) is the true or false determination.

As shown in the previous figure, the true or false decision determines the voltage generated in the next step of the binary progression.

Since the subroutine initiates a current or voltage from a source, its placement in a test sequence is critical. Therefore:

- Call `limitX` and/or `rangeX` before `searchX` when all three refer to the same instrument.
- Call `trigXg` or `trigXl` before `searchX`.

The search operation determines the source voltage or current required at one circuit node to generate a desired trigger point value at a second node. The resolution of the result depends on the number of iterations or steps and the actual current or voltage range being used by the instrument

Figure 18: Voltage or current range formula

$$\frac{\text{voltage or current range}}{2^{(\text{iteration} + 1)}}$$

For example, assume a source's minimum value and maximum value range is from 0 to 20V, and the number of iterations is 16. The 20V level automatically initiates a SMU 20V sourcing range. Therefore, the resolution of the final source voltage returned is:

Figure 19: Voltage or current range formula result

$$\frac{20}{2^{(16+1)}} = 1.2\text{mV}$$

Example: The following example searches for the gate voltage required to generate a drain current of 1 µA. Eight separate gate voltages within the range of 0.6 through 1.7V are specified by `searchv`. After the eight iterations complete, the drain current is close to 1 µA, and the `searchv` operation is terminated. The gate voltage generated at this time by SMU1 is returned as the result VGS1.

Figure 20: Gate voltage requirement

```
double ssbiasv, vgs1, vds1;
.
conpin(SMU1, 1, 0);
conpin(SMU2, 2, 0);
conpin(SMU3, 3, 0);
conpin(GND, 4, 0);
trigig(SMU2, +1.0E-6);  /* Set trigger point for 1mA. */
forcev(SMU3, ssbiasv);  /* Apply a substrate bias */
   /* voltage ssbiasv. */
forcev(SMU2, vds1);     /* Apply a drain voltage of */
   /* vds1. */
searchv(SMU1, 0.6, 1.7, 8,    /* Set for 8 steps from 0.6 to */
 1.0E-3, &vgs1); /* 1.7V at 1ms.*/
   /* per iteration; return the */
   /* result to vgs1. */
execut();
```

setauto - Re-enable autoranging

Purpose: Re-enables autoranging and cancels any previous `rangex` command for the specified instrument.

Format:

```
setauto(long inst_id);
```

`inst_id` The instrument identification code.

Syncmode: None = Full, Result = Full, Full = Full

Remarks: Due to the dual mode operation of the SMU, `setauto` places both voltage and current ranges in autorange mode.

The specific range is not changed until a measurement is taken. When a measurement is taken, the autorange software evaluates the data and changes the range if necessary.

Example:

```
.
rangei(SMU1, 2.0E-9);   /* Select manual range. */
delay(200);      /* Delay after range change. */
measi(SMU1, &icer1);   /* Measure leakage; return. */
.
.
setauto(SMU1);   /* Enable autorange mode. */
lorangei(SMU1, 2.0E-6);/* Select 2mA as minimum range */
    /* during autoranging. */
delay(200);      /* Delay after range change. */
smeasi(SMU1, idatvg);   /* Setup sweep measurement of IDS. */
sweepv(SMU2, 0.0, 2.5, 24, 0.002);   /* Sweep gate from 0 to 2.5V. */
execut();
```

setmode - Set component mode

Purpose: Set instrument specific operating mode parameters.

Format:

```
int setmode(int instr_id, unsigned int modifier, double value);
```

`instr_id` Instrument ID of the instrument being operated on.

`modifier` Instrument specific operating characteristic to change.

`value` Value to set the operating parameter to.

Syncmode: None = None, Result = None, Full = Full

Remarks: `setmode` allows control over certain instrument specific operating characteristics.

A special instrument ID called KI_SYSTEM is used to set operating characteristics of the system. All modifiers listed in the next tables are used with the KI_SYSTEM pseudo-instrument.

Modifiers that affect SMU behavior		
Modifier	**Value**	**Comment**
KI_INTGPLC	CMTR: 0.006 to 10.002 24xx: 0.01 to 10 26xx: 0.001 to 25	Defines a period in terms of AC line cycles over which measurements are averaged.
KI_LIM_INDCTR	any	Controls what measure value is returned if the SMU is at its programmed limit. The `devint` default is SOURCE_LIMIT (7.0e22). **NOTE**: The SMU always returns INST_OVERRANGE (7.0e22) if it is on a fixed range that is too low for the signal being measured.
KI_LIM_MODE	KI_INDICATOR KI_VALUE	Controls whether SMU will return an indicator value when in limit or overrange, or the actual value measured. The default mode after a devint is to return a value.

Modifiers that affect triggering		
Modifier	**Value**	**Comment**
KI_TRIGMODE	KI_MEASX KI_INTEGRATE KI_AVERAGE	Redefines all existing triggers to use a new method of measurement.
KI_TRIGMODE	KI_ABSOLUTE KI_NORMAL	Use absolute value or polarized readings for trigger condition.
KI_AVGNUMBE R	<value>	Number of readings to take when KI_TRIGMODE is set to KI_AVERAGE.
KI_AVGTIME	<value>	Time between readings when KI_TRIGMODE is set to KI_AVERAGE.

Modifiers that affect CVU measurements		
Modifier	**Value**	**Comment**
KI_CVU_LENGTH	0 - 3	Sets the cable length (in meters) for the CVU card.
KI_CVU_MODEL	0 - 5	0 = Z, theta 1 = R, jx 2 = Cp, Gp (default) 3 = Cs, Rs 4 = Cp, D 5 = Cs, D
KI_CVU_MODE	0 or 1	0 = User Mode 1 = System Mode (default)
KI_CVU_SPEED	0 - 2	0 = Fast (default) 1 = Normal 2 = Quiet Does not affect `intgc`, `intgg`, or `intgcg` LPT functions. To adjust the speed of these lpt functions, use the KI_INTGPLC setmode.
KI_CVU_ACV	<value>	10 mV to 100 mV (45 mV is default)
KI_CVU_ACZ_RANGE	<value>	0 = auto range (default) 1e-6 = 1 µA range 30e-6 = 30 µA range 1e-3 = 1 mA range
KI_CVU_FREQ	<value>	1 kHz to 2 MHz (100 kHz is default)
KI_CVU_CORRECT_OPEN	0 or 1	0 = Off (default) 1 = On
KI_CVU_CORRECT_SHORT	0 or 1	0 = Off (default) 1 = On
KI_CVU_CORRECT LOAD	0 or 1	0 = Off (default) 1 = On

Example:

```
double ic[10];
double vb[10];

conpin(SMU1, 1, 0);
conpin(SMU2, 2, 0);
conpin(GND, 3, 0);

setmode(SMU1, KI_INTGPLC, 0.025);

forcev(SMU2, 5.0);
sintgv(SMU1, vb);
smeasi(SMU2, ic);
sweepi(SMU1, 0.0, 1.0e-6, 9, 0.0);
```

setXmtr - Set meter mode

Purpose: Allows a source to operate as a voltmeter or current meter. The source function is disabled after calling setXmtr.

Format:

```
int setimtr(int instr_id);
int setvmtr(int instr_id);
```

`instr_id` Instrument ID of the instrument to control.

Syncmode: None = Full, Result = Full, Full = Full

Remarks: Use x = v for volts and i for current.

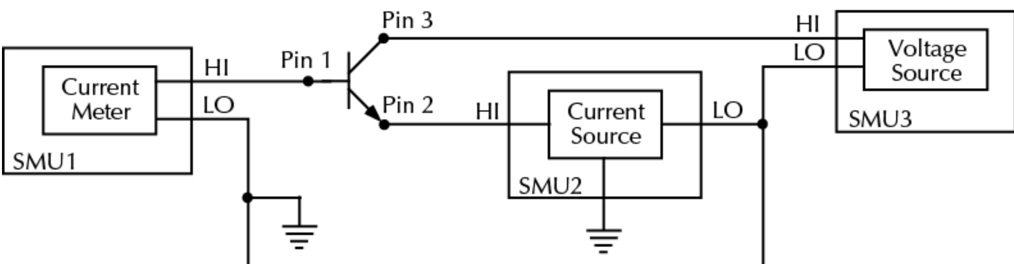
`setXmtr` does not affect any existing connections.

Note that the `setvmtr` operates as a current source with 0.00 A output; it also activates the instrument's voltmeter function. Additionally, the `setimtr` operates as a voltage source with 0.0 V output; it also activates the instrument's current meter function.

The effects of `setXmtr` are also cleared when devint, device initialize, or execut, located at the end of the test sequence, is called.

Example: This figure shows a transistor beta measurement at a specified emitter current and collector-base voltage.

Figure 21: Transistor measurement




```
float vcc12, icc8, ib47;
.
.
conpin SMU1H, 1, 0);
conpin(SMU2H, 2, 0);
conpin(SMU3H, 3, 0);
setimtr(SMU1);/* Set SMU1 as a current meter only. */
forcev(SMU3, vcc12);/* Apply vcc12V to collector. */
forcei(SMU2, icc8);/* Enable icc8 current through emitter. */
measi(SMU1, &ib47);/* Measure base current return result */
   /* to ib47. */
execut();
```

sintgX - Sweep integrate

Purpose: `sintgX` performs an integrated measurement for every point in a sweep.

Format:

```
int sintgi(int instr_id, double *results);
int sintgv(int instr_id, double *results);
```

`inst_id` The measuring instrument's identification code.

`results` The floating point array where the results are stored.

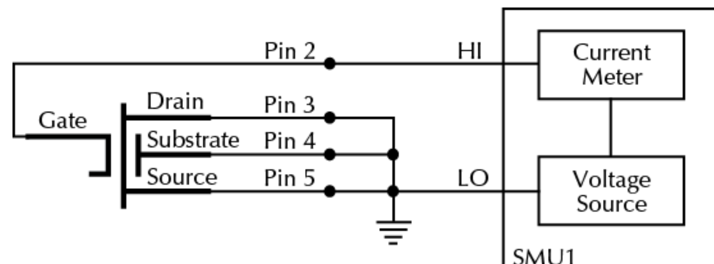
Syncmode: None = None, Result = None, Full = Full

Remarks: This function is used to create an entry in the measurement scan table. During any of the sweeping functions, a measurement scan is performed for every force point in the sweep. During each scan, a measurement will be made for every entry in the scan table. The measurements are made in the same order which the entries were made in the scan table.

`sintgX` sets up the new scan table entry to perform an integrated measurement. The measurement results will be stored in the array specified by results. Each time a measurement scan is made, a new measurement result will be stored at the next location in the results array. If the scan table is not cleared, performing multiple sweeps will simply keep adding new measurement results to the end of the array. Care must be taken to be sure the results array is large enough to hold all measurements which will be taken before the scan table is cleared. The scan table is cleared by an explicit call to `clrscn` or implicitly when `devint` or `execut` is called.

Example: The following example collects information on the low-level gate leakage current of a MOSFET. Sixteen integrated measurements are made as the voltage is increased from 0 to 25.0V

Figure 22: MOSFET low-level gate leakage




```
float idss [16];
.
.
conpin(SMU1, 2, 0);
conpin(GND, 5, 4, 3, 0);
limiti(SMU1, 1.5E-8);
rangei(SMU1, 2.0E-8);  /* Select range for 20nA. */ sintgi(SMU1, idss);     /*
   Measure current with */
   /* SMU1; */
.  /* return results to idss. */
.
sweepv(SMU1, 0.0, 25.0, 15,   /* Perform 16 measurements */
 1.0E-3); /* (steps) from 0 through */
   /* 25V; each step 1ms in */
.  /* duration. */
.
execut();
```

smeasX - Sweep measure

Purpose: `smeasX` allows a number of measurements to be made by a specified instrument during a `sweepX` function. The results of the measurements are stored in the defined array.

Format:

```
int smeasi(int instr_id, double *results);
int smeast(int instr_id, double *results);
int smeasv(int instr_id, double *results);
```

`instr_id` The measuring instrument's identification code.

`results` The floating point array that stores the results.

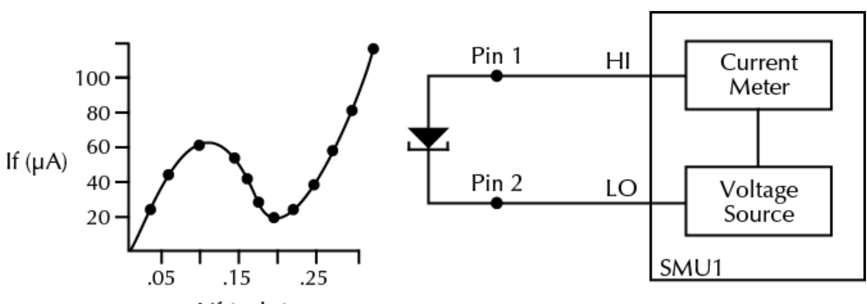
Syncmode: None = None, Result = None, Full = Full

Remarks: This function is used to create an entry in the measurement scan table. During any of the sweeping functions, a measurement scan is performed for every force point in the sweep. During each scan, a measurement will be made for every entry in the scan table. The measurements are made in the same order which the entries were made in the scan table.

`smeasX` sets up the new scan table entry to perform an ordinary measurement. The measurement results will be stored in the array specified by results. Each time a measurement scan is made, a new measurement result will be stored at the next location in the results array. If the scan table is not cleared, performing multiple sweeps will simply keep adding new measurement results to the end of the array. Care must be taken to be sure the results array is large enough to hold all measurements which will be taken before the scan table is cleared. The scan table is cleared by an explicit call to `clrscn` or implicitly when `devint` or `execut` is called.

Example: This example determines the measurement data needed to construct a graph showing the negative resistance characteristics of a tunnel diode. SMU1 generates a voltage ramp ranging from 0 through 0.3V. The current through the diode is sampled 13 times with a duration of 25ms at each step. The results are stored in an array called res1

Figure 23: SMU measurement graph data



```
double resi[13]; /* Defines array. */
.
.
conpin(SMU1, 1, 0);
conpin(GND, 2, 0);
smeasi(SMU1, resi);     /* Make a series of */
   /* measurements; */
.  /* return the results to the */
.  /* resi array. */
sweepv(SMU1, 0.0, 0.3, 12,
 25.0E-3);       /* Make 13 measurements as the */
   /* voltage ranges from 0 to */
   /* 0.3V. */
execut();
```

ssmeasX - Steady state measurement

Purpose: Takes a series of readings until the change (delta) between readings is within a specified percentage. This subroutine is used when the device stability is uncertain. It will continually read the instrument until the resulting measurement is stable and provides the fastest measurement possible.

Format:

```
int ssmeasi(int inst_id, double *result, double delta, unsigned int max_read,
    double delay);
int ssmeasv(int inst_id, double *result, double delta, unsigned int max_read,
    double delay);
```

`inst_id` The measuring instrument's identification code.

`result` The floating point variable assigned to the result of the measurement.

`delta` The termination definition. This is the percentage of the first reading that defines the steady-state condition.

`max_read` The maximum number of readings taken to determine whether or not the reading is steady. If the reading never stabilizes, due to oscillations, charge/ discharge, etc., this reading count will expire and a reading of 'MEAS_NOT_PERFORMED (1.00E23)' is returned.

`delay` The delay between points to wait, in seconds.

Syncmode: None = None, Result = None, Full = Full

Remarks: Any instrument that uses `measX` can use `ssmeasX`. This subroutine calls `measX` for each reading. Any `rangeX` rule applies to this subroutine.

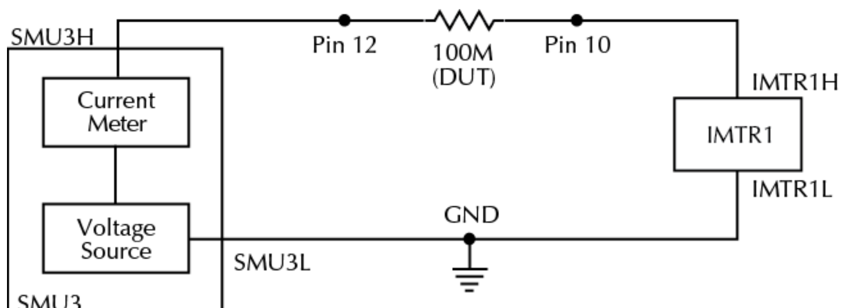
`ssmeasX` is used when taking single point readings. It is NOT used for any of the combination measurements, (for example, `XsweepY`, `trigXY`, etc.).

Example: The following example performs a series of measurements and tests to verify if the present measurement and the previous measurement are within 0.1%. If the measurements are within 0.1%, the result of the last measurement is stored and the program continues. If the measurements are not within 0.1 %, it waits 15ms before taking another measurement. It then compares this measurement with previous measurement. If the measurements are within 0.1%, the result of the last measurement is stored and the program continues. If the measurements are not within 0.1%, it repeats the comparison until the change is within 0.1%. If after 300 attempts the change is not within the specified limit, the following error is reported:

```
"MEAS_NOT_PERFORMED (l.000E23)"
```

There are test conditions where `ssmeasX` is not ideal. An oscillation where two contiguous measurements happen to be within the given percentage will return a stable reading when in reality, the device can not be measured.

Figure 24: Two contiguous measurements



```
double meascur;
.
.
conpin(SMU3, 12, 0);    /* Make connections. */
conpin(SMU2, 10, 0);
setimtr(SMU2);
.
.
forcev(SMU3, .10);      /* Perform test. */
ssmeasi(SMU2, &meascur, 0.1,
    300, .015); /* Steady state measurement /*
   /* with delta of 0.1%, with */
   /* maximum of 300 readings */
.  /* before error, wait 15ms */
.  /* between readings. */
execut();
```

sweepX - Sweep

Purpose: Generates a ramp consisting of ascending or descending voltages or currents. The sweep consists of a sequence of steps each with a user-specified duration.

Format:

```
int sweepi(int inst_id, double startval, double endval, unsigned int stepno, double
    step_delay);
int sweepv(int inst_id, double startval, double endval, unsigned int stepno, double
    step_delay);
```

`inst_id` The sourcing instrument's identification code.

`startval` The initial voltage or current level output from the sourcing instrument and applied for the first sweep measurement. This value can be positive or negative.

`endval` The final voltage or current level applied in the last step of the sweep. This value can be positive or negative.

`stepno` The number of current or voltage changes in the sweep. The actual number of forced data points is one greater than the number of steps specified.

`step_delay` The delay in seconds between each step and the measurements defined by the active measure list.

Syncmode: None = None, Result = Full, Full = Full

Remarks: `sweepX` is always used in conjunction with `smeasX`, `sintgX`, `savgX`, or `rtfary`.

`sweepX` causes a sourcing instrument to generate a series of ascending or descending voltages or current changes called steps. During this source time, a measurement scan is performed at each step. The actual number of forced data points is ONE MORE than the number of steps. Thus, the number of measurements performed is the number of steps plus one. This is important when dimensioning the size of the results array. Failure to make sure the array is big enough will produce operating system access violation errors.

Measurements are stored in a one-dimensional array in the order they were taken.

`trigXg` or `trigXl` can be used with `sweepX` even though they are being used in conjunction with `smeasX`, `sintgX`, or `savgX`. In this case, data resulting from each of the steps is stored in an array, as noted above. However, once a trigger point (for example, a level of current or voltage) is reached, the sourcing device stops incrementing or decrementing and is held at a steady output level for the remainder of the sweep.

The system maintains a measurement scan table consisting of devices to measure. This table is maintained by calls to `smeasX`, `sintgX`, or `savgX`, or `clrscn`. As multiple calls to these functions are made, the commands are appended to this table.

When multiple calls to `sweepX` are executed in the same test sequence, the `smeasX`, `sintgX`, or `savgX` arrays are loaded sequentially. This appends the measurements from the second `sweepX` call to the previous results. If the arrays are not dimensioned correctly, access violations will occur. The measurement table remains intact until clrscn, devint, or execut are executed.

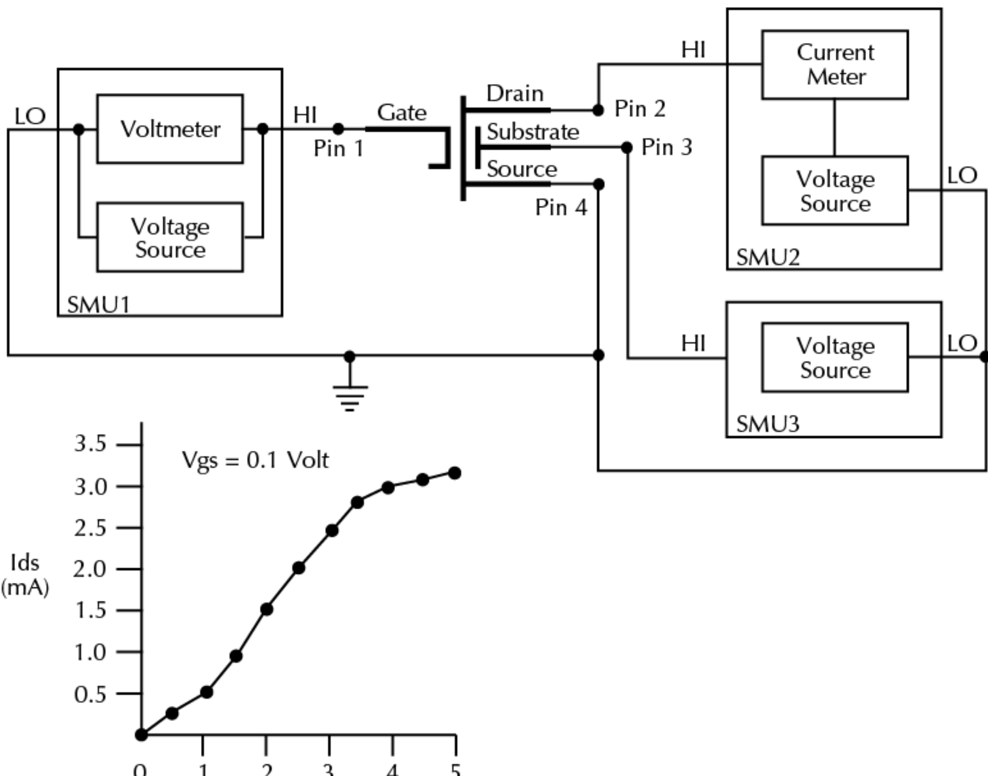
Defining new test sequences using `smeasX`, `sintgX`, or `savgX` adds commands to the active measure list. The previous measures are still defined and used. `clrscn` is used to eliminate the previous measures for the second sweep. Using `smeasX`, `sintgX`, or `savgX` after clrscn causes the appropriate new measures to be defined and used.

In cases where the first sweep point is non-zero, it may be necessary to precharge the circuit so that sweep will return a stable value for the first measured point without penalizing remaining points in the sweep. For example:

```
double ires[6];
conpin(SMU1, 10);
conpin(2, GND, 0);
forcev(SMU1, 5.0);      /* Force 5V to charge. */
delay(10);       /* Wait for precharge. */
smeasi(SMU1, &ires);    /* Set up measurement. */
sweepv(SMU1, 5.0, 10.0, 5,     /* Do the real measurement. */
 2.5E-3);
```

Example: The following example gathers data to construct a graph showing the common drain-source characteristics of a FET. A fixed gate-to-source voltage is generated by SMU1. A voltage ramp from 0 through 5V is generated by SMU2. Drain current applied by SMU2 is measured 11 times by `smeasi`. Data is stored in the array `resi`.

Figure 25: FET common drain-source characteristics



```
double resi[11], ssbiasv;
.
.
conpin(SMU1, 1, 0);
conpin(SMU2, 2, 0);
conpin(SMU3, 3, 0);
conpin(GND, 4, 0);
forcev(SMU3, ssbiasv); /* Apply substrate bias vol- */
   /* tage SSBIASV. */
forcev(SMU1, -.1);     /* Apply a gate-to-source */
   /* voltage of -0.1V. */
smeasi(SMU2, resi);    /* Perform a series of current */
   /* measurements; return */
   /* the results to the array */
   /* resi. */
sweepv(SMU2, 0.0, 5.0, 10,    /* Generate 11 steps and 11 */
 2.5E-3); /* points each 2.5ms duration, */
   /* ranging from 0 to 5V. */
execut();
```

syncmode - Set synchronization mode

Purpose: To set the synchronization mode of the tester.

Format:

```
int syncmode(int mode);
```

`mode` The synchronization mode to use.

Remarks: The S530 tester is designed to maximize throughput by decoupling the three tasks of sending LPT commands to the tester, executing them, and sending results back. The default mode of execution ensures that these three tasks are synchronized properly.

In this default mode of execution, a function which does not return results will return before the command is actually executed on the tester. Those commands which return results will wait for the results before continuing. This is known as result synchronization. In this mode of execution, throughput is greatly increased while ensuring that return values are available at the completion of any LPT function which returns a value.

For the advanced user, throughput can be maximized by adjusting the synchronization mode. The `syncmode` command supports three synchronization modes:

SYNC_FULL — Full synchronization will force LPT to wait for every command to completely finish executing before returning. This reduces throughput significantly because communication with the tester cannot be done in parallel with command execution.

SYNC_RESULT — Result synchronization, the default mode, allows communication with the tester to be done in parallel with test execution and allows LPT commands to be pre-queued in the tester to keep the instruments busy without needing to wait for communications overhead. Although communications overhead is reduced, it is still incurred for those commands which return values.

SYNC_NONE — No synchronization allows commands which do return values to return before the results are actually received from the tester. If the result is not used immediately, this mode of synchronization will eliminate communication overhead associated with waiting for the results to be received. When the synchronization mode is set to SYNC_NONE, it is the users responsibility to resynchronize prior to using or returning any LPT results.

The `execut` and `inshld` commands both force synchronization with the tester and ensure that all results have been received and can be used. `execut` also resets the synchronization mode back to the default mode SYNC_RESULT.

NOTE

The syncmode command itself does not resynchronize with the tester, it only sets the synchronization mode to be used by the next LPT command. `execut` or `inshld` must be used to resynchronize when required.

The KTE tools use result synchronization. If the synchronization mode is changed during execution of a `userlib`, it must be set back to SYNC_RESULT prior to exiting the `userlib` function. This can be done explicitly by calling syncmode or implicitly by calling `execut`. If the `userlib` function returns results, `execut` can be used to ensure all results have been received prior to exiting the function.

```
double i1, i2;
syncmode (SYNC_NONE);  /* don't wait for results */
. . . .
forcev (SMU1, 5.0);
forcev (SMU2, 1.0e-3);
measi (SMU1, &i1);
```

```
syncmode (SYNC_RESULT);
/* will cause measi to wait for results
to be returned before continuing */
measi (SMU2, &i2);
```

trigXg, trigXl - Trigger greater than, less than

Purpose: Monitors for a predetermined level of voltage, current, conductance, capacitance, or time.

Format:

```
int trigig(int inst_id, double value);
int trigil(int inst_id, double value);
int trigtg(int inst_id, double value);
int trigtl(int inst_id, double value);
int trigvg(int inst_id, double value);
int trigvl(int inst_id, double value);
```

`inst_id` The monitoring instrument's identification code.

`value` The voltage, current, capacitance, conductance, or time specified as the trigger point. This trigger point value is considered to be reached when:
The measured value is equal to or greater than the value argument of `trigXg`, or
The measured value is less than the value argument of `trigXl`.

Syncmode: None = None, Result = None, Full = Full

Remarks: `trigXl` and `trigXg` are used with `searchX` or with one of the sweep measurement routines: `smeasX`, `sintgX`, or `savgX`.

`trigXg` or `trigXl` provides `sweepX` the digital feedback to allow for the increase or decrease in sourcing values.

`trigXl` and `trigXg` must be located before any associated `searchX`.

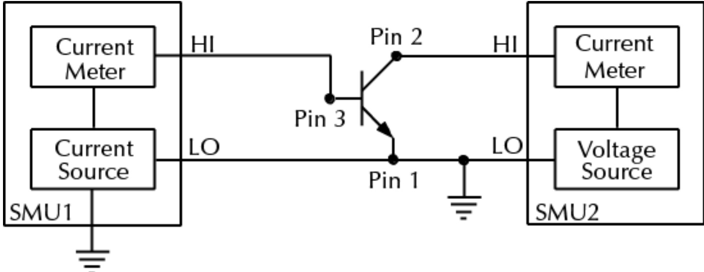
Triggers are not automatically reset by `searchX` or `sweepX`. A single `trigXl` or `trigXg` can be followed by two or more `searchX` or `sweepX` calls.

The specified trigger point is automatically cleared when a `clrtrg`, `devint`, or `execut` is run.

Example: `trigig` programming example

The following example uses `trigig` and `searchi` together to generate and search for a specific current level. A search is initiated to find the base current needed to produce 5mA of collector current. The collector-to-emitter voltage supplied by SMU2 is defined by the variable VCC8. `searchi` generates the base current from SMU1. This current ranges between 50 mA and 200 mA in 15 iterations. `trigig` continuously monitors the current through SMU1. The base current supplied by SMU1 is stored as the result `res22`.

Figure 26: Generate and search current level



```
double res22, vcc8;
.
.
conpin(SMU1, 3, 0);
conpin(SMU2, 2, 0);
conpin(GND, 1, 0);
forcev(SMU2, vcc8);    /* Apply collector voltage to
          /* vcc8.*/
trigig(SMU2, +5.0E-3); /* Search for a collector cur- */
   /* rent of 5mA. */
searchi(SMU1, 5.0E-5, 2.0E-4,
    15, 1.0E-3, &res22);      /* Generate a current ranging */
   /* from 50 to 200mA in */
   /* 15 iterations. Return the */
   /* current resulting from the */
   /* last iteration as res22. */
execut();
```

Example: `trigil` programming example

The following example uses `trigil` together with `smeasi` and `sweepv` to generate a voltage staircase-type waveform and then measure the resulting current. The waveform holds at its last value when the `trigil` value is reached. The data needed to generate a graph showing the forward voltage-to-current characteristics of a diode are stored in an array. SMU1 is configured as a voltage source and current meter. The `sweepv` sources voltage from SMU1. This voltage staircase ranges from 0.6 to 0.0V in 18 steps with a 1ms duration at each step. `trigil` stops the ramping initiated by `sweepv` when the current through the diode is greater than -5mA. `smeasi` measures the current applied by SMU1 at each voltage step and stores the results in array res1.

Figure 27: Generate and measure current

```
double res1[19];
.
.
conpin(SMU1, 1, 0);
conpin(GND, 2, 0);
trigil(SMU1, +5.0E-3); /* If greater than -5mA, */
   /* stop ramping. */

smeasi(SMU1, res1);    /* Measure current at each of */
   /* the 19 levels; return re- */
   /* sults to the res1 array. */
sweepv(SMU1, 0.6, 0.0, 18,
    1.00E-3);   /* Generate 0.6 to 0.0V */
   /* in 19 steps. */
execut();
```

tstel - Test station select

Purpose: Used to enable or disable a test station. Only one test station can be active at any given time.

Format:

```
tstsel (long x);
```

`x` The test station number, 1 is the only value allowed.

Remarks: `tstsel` is normally called at the beginning of a test program. Only one `tstsel` per program is recommended.

Calling a new test station within a test program cancels any prior `tstsel` calls.

To relinquish control of an individual test station so another station can access the system, call `tstsel` with a value of zero. A new test station must now be selected before any subsequent test control subroutines are run.

The error message, "No command buffer allocated. Call `tstsel` first" will appear if a test sequence is executed before executing `tstsel`.

Attempting to run a test program on an already active test station will cause the message "Error: `tstsel` failed with status = -653. Exiting" to be displayed.

Section 4

Error Definitions

In this section:

Result values indicating an error 4-1
Error messages 4-2

Result values indicating an error

The following table contains errors that are returned as MEASURED results and NOT as LPTLib return status codes. Each value has a specific meaning. If one of these values is returned from any measurement subroutine, the test was not performed and action is required by the Keithley Programmer/Manager.

Table: Errors

Value	Description
1.0E22	Instrument Overrange. Measurement instrument performed the measurement, but the resulting value was inappropriate for the range specified. This is either from selecting the wrong range for manual ranging or from autoranged measurement on a dynamically changing signal (slowly charging device).
2.0E22	Current Overload. Measurement was not performed due to a shutdown condition. The source went into a current overload condition and shutdown.
3.0E22	Oscillation Detection. Instrument was forced into a shutdown state because oscillations were detected at the instrument.
4.0E22	Thermal Shutdown. Instrument is dissipating too much heat. The instrument is shutdown to keep from self-destructing.
5.0E22	SOA Exceeded. The instrument is designed to operate within the Safe Operating Area. If the programming conditions exceed this area, this error value is returned.
6.0E22	Pulse Width Too Short. The specified pulse width is less than the minimum specified for the instrument.
7.0E22	Source Limit. Measurement was not performed because source was in limit.
1.0E23	Measurement Not Performed. If an error causes a measurement not to be performed, this result is returned. Previously, if a measurement was not taken, nothing was returned, so the result available to the user (from the execution of the test sequence) was the previous reading. This could be confusing. Now, whenever possible, it will return this error value as a default value.

Error messages

This section lists all of the error messages. The error number is indicated for each error, the error message describes each error, if it occurs, and any remarks about the error message. Note that LPTLib functions return negative values while the messages will indicate them as positive values.

3 LPT_NOCOMCHAN

Message: **This is a host side only error. No message will be generated by the tester.

Remarks: LPTLib calls may only be issued after a successful call to tstsel. Make sure tstsel is called before any other LPTLib command.

5 SYS_MEM_ALLOC_ERR

Message: Memory allocation failure.

Remarks: The tester does not have enough memory for the Keithley system software to run correctly.

20 LPT_PREVERR

Message: Command not executed because a previous error was encountered.

Remarks: There was an error encountered during a test sequence. All LPTLib commands after that point will generate this error. Correct the problem in the test sequence causing the first error.

21 LPT_FATAL

Message: Tester is in a fatal error state.

Remarks: The tester is in a state requiring user attention. Correct the problem requiring attention.

22 LPT_FATALINTEST

Message: Fatal condition detected while in testing state.

Remarks: The tester entered a fatal state while a test sequence was in progress. Results generated during this test sequence may not be valid.

24 LPT_TOMANYARGS

Message: Too many arguments.

Remarks: An LPTLib function has passed more arguments than it can handle. Reduce the number of arguments by trying to split the call into two or more smaller calls.

100 MX_INVLDCNT

Message: Invalid connection count, number of connections passed was NNN.

Remarks: The matrix driver could not determine what to connect because there were not enough terminal IDs or pins specified. This is usually caused by passing -1 for all but one argument to a matrix function.

101 MX_NOPIN

Message: Argument #NNN is not a pin in the current configuration.

Remarks: A request was made to make a connection to a pin that does not exist.

102 MX_MULTICON

Message: Multiple connections on XXX.

Remarks: Certain terminals can only be connected to one pin at a time. An attempt was made to connect this terminal to more than one pin at a time.

109 MX_ILLGLTSN

Message: Illegal test station: NNN.

Remarks: An internal system software error has occurred.

113 MX_NOSWITCH

Message: There are no switching instruments in the system configuration.

Remarks: The system did not detect any matrix hardware during system configuration.

114 MX_ILLGLCON

Message: Illegal connection.

Remarks: An attempt was made to make a connection that physically cannot be made, or is disallowed.

122 UT_INVLDPRM

Message: Illegal value for parameter #NNN.

Remarks: An invalid value was passed as the NNNth argument to an LPTLib function.

126 UT_NOURAM

Message: Insufficient user RAM for dynamic allocation.

Remarks: The system could not allocate memory for user data. This could be caused by sweeps with an unusually large number of steps or by large sweeps that measure too many parameters.

129 UT_TMRIVLD

Message: Timer not ENABLED.

Remarks: Time measurements can only be made on a timer when the timer is enabled.

137 UT_INVLDVAL

Message: Invalid value for modifier.

Remarks: An invalid option was used for setmode or getstatus.

152 CB_BADFUNC

Message: Function not supported by XXX (NNN).

Remarks: The driver for this instrument does not support the LPTLib function used.

156 CB_NOFILE

Message: Configuration file does not exist: XXX.

Remarks: A required configuration file is missing.

157 CB_FORMAT

Message: Configuration file format error. File: XXX, Section: XXX, Key: XXX.

Remarks: A system configuration file has a missing entry, or one that was formatted in an unexpected way.

162 CB_INVLDERROR

Message: Invalid error number: NNN.

Remarks: The logging .ini file has a format error.

163 CB_INVLDEVENT

Message: Invalid event number: NNN.

Remarks: The logging.ini file has a format error.

166 CB_INSNOTREC

Message: Instrument with model code XXX is not recognized.

Remarks: An instrument is not recognized by the system. The instruments may be misread over the 170 CB_INITFAIL.

194 MX_INVLDTRM

Message: Invalid terminal: XXX.

Remarks: The terminal specified is invalid for the command.

233 FM_NOCON

Message: Cannot force when not connected.

Remarks: The instrument must be connected to a DUT before it can be used.

455 ECP_PROTOVER

Message: Protocol version mismatch.

Remarks: The application controller software version and the tester software version do not match. Either the system software was not installed correctly, or the application controller is being used to control an older/newer tester not intended to be used with this application controller.

601 SYS_INTERNAL_ERR

Message: System software internal error.

Remarks: An internal system software error has occurred.

610 SYS_SPAWN_ERR

Message: Could not start XXX.

Remarks: An internal system software error has occurred.

611 SYS_NETWORK_ERR

Message: Network error.

Remarks: The system is having problems communicating over the network. Make sure that all network connections are secure, all network cables are in tact, and any network routers are functioning properly.

612 SYS_PROTOCOL_ERR

Message: Protocol error.

Remarks: An internal system software error has occurred.

650 TAPI_BADCHANNEL

Message: Request to open unknown channel type XXX.

Remarks: An internal system software error has occurred.

651 TAPI_BADTESTER

Message: **This is a host side only error. No message will be generated by the tester.

Remarks: There is no network node corresponding to the tester address. Make sure the network configuration is correct.

652 TAPI_NOTFOUND

Message: **This is a host side only error. No message will be generated by the tester.

Remarks: The tester cannot be located on the network. Make sure the tester is powered on and has started correctly. This problem could also be caused by network problems.

653 TAPI_REFUSED

Message: **This is a host side only error. No message will be generated by the tester.

Remarks: The tester is in use. This can be caused by another process running diagnostics or a test plan.

656 TAPI_CHANLIMIT

Message: Channel limit exceeded.

Remarks: There are too many active network connections to the tester. Close some of the tools that communicate with the tester and try again.

657 TAPI_BUFOFLOW

Message: **This is a host side only error. No message will be generated by the tester.

Remarks: An internal system software error has occurred.

Specifications are subject to change without notice.
All Keithley trademarks and trade names are the property of Keithley Instruments, Inc.
All other trademarks and trade names are the property of their respective companies.

Keithley Instruments, Inc.
Corporate Headquarters • 28775 Aurora Road • Cleveland, Ohio 44139 • 440-248-0400 • Fax: 440-248-6168 • 1-888-KEITHLEY • www.keithley.com

A Tektronix Company

A Greater Measure of Confidence

7/13