Copy or move UTM GUI definitions

The UTM GUI definition consists of one file for each user library. Use the kultcopy command to move a user library to another 4200. This command copies the user modules (c code), UTM GUI definition (XML files), and any bitmaps for the UTM GUI (bitmapped files with the .bmp extension). For more information, see kultcopy in "Copying user libraries using kultcopy" on page 8-47.

LPT Library Function Reference

The Keithley Instruments 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 functions let the user configure the relay matrix and instrumentation to perform parametric tests.

This section lists the functions included in the Keithley Instruments 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 function, along with the function's placement and relationship to other functions in a test sequence are described.
- Examples showing a typical use of the function in a test sequence.

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

- **All LPTLib functions are case-sensitive and must be entered as lower case when writing program codes.**
- **A capital letter X shown in a function 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.

Table 8-10
Possible suffixes

Suffix	Parameter	Unit
`i`	Current	Amperes
`t`	Time	Seconds
`v`	Voltage	Volts

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

NOTE *In this section, **`10 point Courier italic`** distinguishes the parameters in the Linear Parametric Test Library (LPTLib) function prototype from other elements.*

*In this section, **`10 point Courier bold`** highlights references to the name of the LPTLib function being described, as well as references to names of associated LPTLib functions.*

Using source compliance limits

When sourcing voltage (`forcev`), a current compliance limit can be set. When sourcing current (**`forcei`**), a voltage compliance limit can be set. The SMU will not exceed the compliance limits.

The **`limitX`** functions are used to set the compliance limits (**`limiti`** sets current compliance and **`limitv`** sets voltage compliance). The actual compliance limit that is in effect depends on the present measurement range. If the SMU is on a range that can accommodate the compliance limit value, then that compliance limit will be in effect. If the SMU is on a range that is too low for the compliance limit, the full scale value of the range will instead be used.

For example, if the compliance limit setting is 5 mA, but the SMU is on the 1mA range, the actual compliance will be 1.05 mA (full scale value). When the SMU goes up to the 10 mA range, the 5 mA compliance limit goes into effect.

There are two ways to perform a measure range change; manually select a range or use autorange. A range is manually selected by using a **`rangeX`** function. When a **`rangeX`** function is not used, the SMU will use autoranging when a measure function (**`intgX`**, **`measX`**, **`avgX`**, or **`bmeasX`**) is called.

The following procedures demonstrate the proper sequence for using the limit, range, source, and measure functions.

Autoranging: For autoranged measurements, perform the following steps to perform a source-measure operation:

1. Use a **`limitX`** function to set the desired compliance limit.
2. Use a **`forceX`** function to output a source value.
3. Use a measuring function (**`intgX`**, **`measX`**, or **`avgX`**) to perform an autoranged current or voltage measurement. Before performing the measurement, the SMU will first go to the most sensitive range for the measured reading.

For source-only functionality, a dummy measurement (Step 3 above) needs to be made if an increase in the measurement range is required for a new compliance limit setting.

Manual ranging: When using a manual measurement range, to perform a source-measure operation:

1. Use a **`limitX`** function to set the compliance limit.
2. Use a **`rangeX`** function to select a manual measure range that will accommodate the compliance limit and the measured reading.
3. Use a **`forceX`** function to output a source value.
4. Use a measuring function (**`intgX`**, **`measX`** or **`avgX`**) to perform a current or voltage measurement.

For source-only operation, omit Step 4 of the above procedure.

LPT functions

In Table 8-11, function calls are grouped by function. The details about the functions for the SMUs and general operations are listed alphabetically after the table (see LPT functions for SMUs and general operations later in this section).

Details on functions for the pulse generators are described:

- For the 4205-PG2, see LPT functions for the Model 4205-PG2.
- For the 4220-PGU and 4225-PMU, see LPT functions for the Models 4220-PGU and 4225-PMU.

NOTE *The 4205-PG2, 4220-PGU, and 4225-PMU are dual-channel pulse generator cards inside the 4200-SCS. To differentiate between an internal pulse card and other supported pulse instruments, a 4205-PG2 is referred to as a VPU (voltage pulse unit). With LPT functions, the 4205-PG2 and 4220-PGU are referred to as VPU1, VPU2, and so on. A 4225-PMU (pulse-measure unit) is referred to as a PMU (PMU1, PMU2, and so on).*

NOTE *Table 15-44 provides the LPTLib function listing for the 4210-CVU. Details on the functions for the 4210-CVU follow the table.*

Table 8-11
Consolidated LPTLib function listing

Group	Function call
Instrument	devclr (Device clear) devint (Device initialize)
Matrix	addcon (Add connection) clrcon (Clear connection) conpin (Connect pin) conpth (Connect path) delcon (Delete connection)
Ranging	lorangeX (Define lowest range, `i,v`) rangeX (Range `i,v`) setauto (Re-enable autorange)
Sourcing	forceX (Force `i,v`) limitX (Limit `i,v`) mpulse (Generate pulse and measure output) pulseX (Generate pulse `i,v`)
Measuring	avgX (Average `i,v`) bmeasX (Block measure `i,v`) imeast (Immediate measure time) intgX (Integrate `i,v`) measX (Measure `i,t,v`)
Combination	asweepX (Array sweep `i,v`) bsweepX (Linear breakdown sweep `i,v`) clrscn (Clear scan) clrtrg (Clear trigger) rtfary (Return FORCE array) savgX (Sweep average `i,v`) scnmeas (Scan measure) searchX (Binary search `i,v`) sintgX (Sweep integrate `i,v`) smeasX (Measure `i,t,v`) sweepX (Linear sweep `i,v`) trigXg, trigXl (trigXg: trigger if `i,t,v` is $\geq$; trigXl: trigger if `i,t,v` is $\leq$)
Timing	adelay (Array delay) delay (Delay) disable (TIMER) (Time measurement function) enable (TIMER) (Time measurement function) rdelay (Real delay)

Table 8-11 (continued)
Consolidated LPTLib function listing

Group	Function call
GPIB	kibcmd (Send GPIB command to instrument) kibdefclr (Clear instrument on `devclr`) kibdefdelete (Delete GPIB definition strings for `devclr` and `devint`) kibdefint (Clear instrument on `devint`) kibrcv (Read device dependent string) kibsnd (Send device dependent command) kibspl (Serial poll an instrument) kibsplw (Synchronous serial poll)
RS232	kspcfg (Configure the port) kspdefclr (Define string to clear RS-232 instrument on `devclr`) kspdefdelete (Delete RS-232 definition strings for `devclr` and `devint`) kspdefint (Define string to clear RS-232 instrument on `devint`) ksprcv (Send device dependent command string) kspsnd (Read device dependent string)
General	getinstattr (Get configured instrument attributes) getinstid (Get instrument ID value from instrument name string) getinstname (Get instrument name string from instrument ID) GetKiteSite (Get KITE site number for site that is presently being tested) getstatus (Read system and instrument status information) setmode (Set operating mode) tstdsl (Test station de-select) tstsel (Test station select)
Execution*	execut (Execute) inshld (Instrument hold)
Arithmetic*	kfpabs (Floating point absolute value) kfpadd (Floating point add) kfpdiv (Floating point divide) kfpexp (Floating point raise e to a power) kfplog (Floating point return a natural logarithm) kfpmul (Floating point multiply) kfpneg (Floating point negative value) kfppwr (Floating point raise to a power) kfpsqrt (Floating point square root) kfpsub (Floating point subtract)
CVU (C-V measure)*	Table 15-44 provides the LPTLib function listing for C-V testing using the 4210-CVU. Details on using the CVU are covered in Section 15 of this manual.

Table 8-11 (continued)
Consolidated LPTLib function listing

Group	Function call
PG2 (pulse only)*	Note: See LPT functions for the Model 4205-PG2 later in this section for details on the following functions: arb_array (Define a full-arb waveform) arb_file (Load a waveform from a full-arb waveform file) pg2_init (Reset PG2 to the specified pulse mode and its default settings) pulse_burst_count (Set burst mode pulse count) pulse_current_limit (Set current limit for pulse output) pulse_dc_output (Select DC output and set level) pulse_delay (Set time delay from trigger to pulse output) pulse_fall (Set pulse fall time) pulse_halt (Stops all pulse output) pulse_init (Reset Standard pulse to its default settings) pulse_load (Set output impedance of DUT) pulse_output (Set output channel on or off) pulse_output_mode (Set output mode to normal or complement) pulse_period (Set pulse period) pulse_range (Set voltage range for pulse low or pulse high) pulse_rise (Set pulse rise time) pulse_ssrc (Control the high endurance output relays for 4205-PG2) pulse_trig (Set trigger mode and initiate (or arm) pulse output) pulse_trig_output (Set trigger output on or off) pulse_trig_polarity (Set trigger output polarity) pulse_trig_source (Set trigger source) pulse_vhigh (Set pulse V High level) pulse_vlow (Set pulse V Low level) pulse_width (Set pulse width) seg_arb_define (Define a Segment ARB® waveform) seg_arb_file (Load a waveform from a Segment ARB waveform file)
Note: Triggering transition time must be <100 ns. Trigger output Level and Trigger In Level must be TTL.	

Table 8-11 (continued)
Consolidated LPTLib function listing

Group	Function call
PGU (pulse only) and PMU (pulse and measure)*	Note: See LPT functions for the Models 4220-PGU and 4225-PMU later in this section for details on the following functions: dev_abort (aborts a pulse_exec test) PostDataDouble (posts buffer data into KITE Sheet tab) PostDataDoubleBuffer (posts buffer data into KITE Sheet tab (large data sets)) PostDataInt (Post an integer-type data point into KITE Sheet tab) PostDataString (Transfers a string to the KITE Sheet tab) pulse_chan_status (Returns the number of readings in the data buffer) pulse_conncomp (Controls connection compensation) pulse_exec (Starts test execution) pulse_exec_status (Checks the run status of the test) pulse_fetch (Retrieves enabled test data) pulse_limits (Sets voltage, current, and power thresholds) pulse_meas_sm (Configures spot mean measurements) pulse_meas_wfm (Configures waveform measurements) pulse_meas_timing (Sets the measurement windows) pulse_measrt (Returns data in pseudo real-time) pulse_ranges (Sets the pulse voltage and measure ranges) pulse_remove (Removes a channel from the test) pulse_sample_rate (Sets the sample rate) pulse_source_timing (Sets pulse source timing) pulse_step_linear (Configures pulse stepping type) pulse_sweep_linear (Configures pulse sweeping type) pulse_train (Configures a pulse train) rpm_config (Configures the Model 4225-RPM) seg_arb_sequence (Defines a Segment ARB pulse-measure sequence) seg_arb_waveform (Creates a Segment ARB pulse-measure waveform) setmode (Sets the number of iterations for LLEC (load line effect compensation))

* Provided for compatibility with other-platform versions of the LPT Library. The 4200-PG2 and 4205-PG2 are referred to as VPU1, VPU2, and so on in LPT functions. The 4220-PGU is referred to as VPU1, VPU2, and so on. The 4225-PMU is referred to as PMU1, PMU2, and so on.The 4210-CVU is referred to as CVU1, CVU2, and so on. Note that the 4225-PMU and 4225-PGU support the PG2 commands.

LPT functions for SMUs and general operations

addcon Add connection

Purpose Add connections without clearing existing connections.

Format

```
int addcon(int exist_connect, int connect1, [connectn, […]] 0);
```

`exist_connect` An instrument terminal ID. This instrument or terminal may have been, but is not required to have been, previously connected with `addcon`, `conpin`, or `conpth`.

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

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

Remarks `addcon` can be used to make additional connections on a 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. `addcon` behaves like the `conpin` command, except previous connections are never cleared.

Before making the new connections, `addcon` clears all active sources by calling `devclr`.

The value -1 will be ignored by `addcon` and is considered a valid entry in the connection list; however, `exist_connect` may not be -1.

With the row-column connection scheme, only one instrument terminal may be connected to a pin.

See also `clrcon, conpin, conpth, delcon`

adelay Array delay

Purpose Specifies an array of delay points to use with `asweepX` calls. The delay is specified in units of seconds, with a resolution of 1 ms. The minimum delay is 0 s.

Format

```
int adelay(long 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.

Remarks Each delay in the array is added to the delay specified in `asweep`. 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 `asweep` 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, long num_points, double delay_time,
double *force_array);

int asweepv(int inst_id, long 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.

Remarks

`asweepX` is used with `smeasX`, `sintgX`, or `savgX` functions.

`trigXl` and `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 `asweep`. 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 functions. 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.

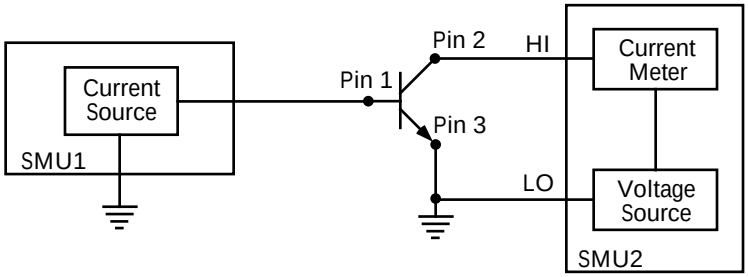
Note that changing the source mode of the SMU can modify the measure range. If the sourcing mode is changed from voltage to current sourcing (or from current to voltage sourcing), the measure range may be changed to minimize variations in the SMU output level. See Changing source mode may change measure range for recommended command order.

If `adelay` is called prior to `asweepX`, each `adelay` value is added to the `asweepX` `delay_time`. This sum is compared to the maximum delay for the configured

instrument card and if any value is larger, an error will occur. The SMU maximum delay is 2,147.483 s. The CVU maximum is 999 s.

Example 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 10 times by **smeasi**. The data gathered is then stored in the ICMEAS array.

Figure 8-87
Diagram




```
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, 10 ms */
/* apart. */
execut();
```

avgX Average

Purpose Performs a series of measurements and averages the results.

Format

```
int avgi(int inst_id, double *result, long stepno, double
steptime);
```

```
int avgv(int inst_id, double *result, long stepno, double
steptime);
```

`inst_id`	The instrument identification code of the measuring instrument.
`result`	The variable assigned the measurement's result.
`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. The minimum practical time is approximately 2.5 µs.

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 function 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.

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 10.0 V on a 4.0 V range. An overrange condition returns the value 1.0E+22 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.

Compliance limits: A compliance limit setting goes into effect when the SMU is on a measure range that can accommodate the limit value. For manual ranging, the rangeX function is used to select the range. For autoranging, the avgi or avgv function will trigger a needed range change before the measurement is performed. See Using source compliance limits earlier in this section for details.

Example

This example illustrates how the avgX command could be used to take five current readings and return the average of the measurements to the variable leakage:

```
double leakage;

.
.
limiti(SMU1, 1.0e-06);/* Limit the maximum current */
 /* to 1µA */
forcev(SMU1, 10.0);/* Force 10 V across the DUT */
delay(100);/* Delay 100 mS to allow for */
/* device settling */
avgi(SMU1, &leakage, 5, 0.01);/* Average 5 readings, delay */
/* 10 mS per measurement */
```

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, long numrdg,
double delay, int timerid, double *timerdata);

int bmeasv(int inst_id, double *results, long 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.

Remarks

This function 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 function, and the time array is returned with the result array.

Example

The example below shows how `bmeas` 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(TIMER1);/* 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 */
 TIMER1, timer);/* readings using SMU3 with */
/* 100µs delay between */
/* readings, using TIMER1 with */
/* time data labeled timer. */
```

Example

Using no timer.

This example shows how the `bmeas` function 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. */
```

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, long
num_points, double delay_time, double *result);
```

```
int bsweepv(int inst_id, double startval, double endval, long
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 32,767.
`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.

Remarks

`bsweepX` is used with `trigXg`, `trigXl`, or `trigcomp`. 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` functions. 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. The results

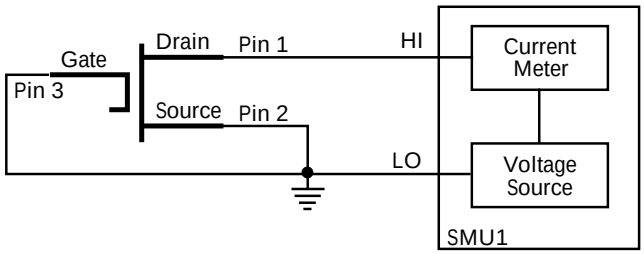
from the second bsweepX call are appended to the results of the previous bsweepX. This can cause access violation errors if the arrays were not dimensioned for the absolute total. The measurement scan table remains intact until a devint, clrscn, or execut command completes.

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.

Note that changing the source mode of the SMU can modify the measure range. If the sourcing mode is changed from voltage to current sourcing (or from current to voltage sourcing), the measure range may be changed to minimize variations in the SMU output level. See Changing source mode may change measure range for recommended command order.

Example The following example measures the drain to source breakdown voltage of a FET. A linear voltage sweep is generated from 10.0 V to 50.0 V by SMU1 using the bsweepv function. The breakdown current is set to 10 mA by using trigil. The voltage at which this current is exceeded is stored in the variable bvdss.

Figure 8-88
bsweepv function example




```
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 10 mA. */
bsweepv(SMU1, 10.0, 50.0, 40,/* Sweep from 10 to 50 V in 40 */
 10.0E-3, &bvdss);/* steps with 10 ms settling */
   /* time per step. */
```

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);
```

Remarks clrcon is called automatically by devint and execut. The first in a series of one or more connection type functions automatically calls a clrcon.

If an RPM is used, clrcon clears the physical connection to the DUT.

clrscn Clear scan table

Purpose Clears the measurement scan tables associated with a sweep.

Format

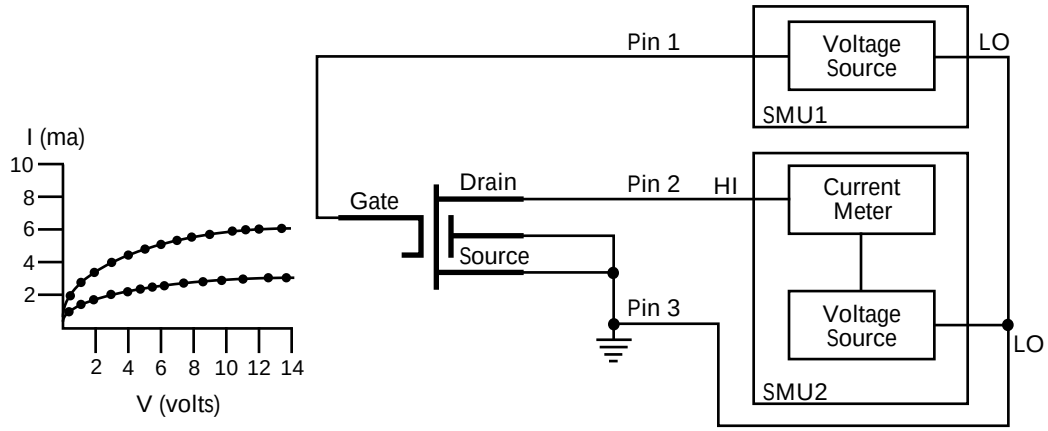
```
int clrscn(void);
```

Remarks When a single `sweepX` is used in a test sequence, there is no need to program a `clrscn` because `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 +14 V 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 5 V 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 the voltage-to-drain current characteristics of a FET gate. The program samples the current generated by SMU2 14 times. This is done in two phases: First with 4 V applied to the gate, and second with 5 V applied. The gate voltages are generated by SMU1.

Figure 8-89
sweepX



```
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 5 V 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. */
```

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.

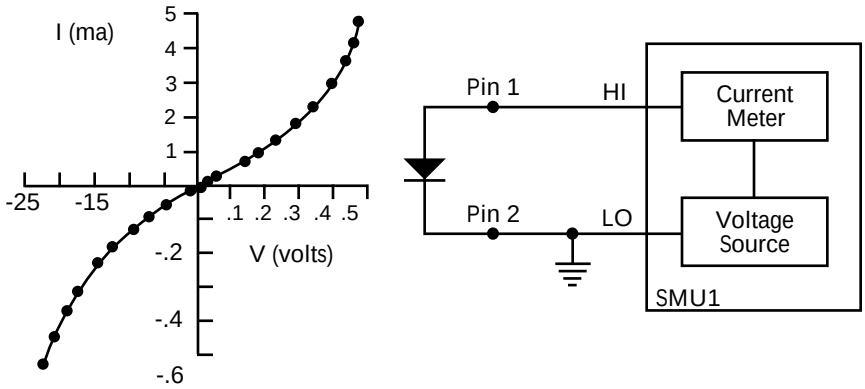
Format

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

Remarks `searchX`, `sweepX`, `asweepX`, or `bsweepX`, each with different voltage or current levels, may be used repeatedly within a function 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 multiple triggers to be programmed twice in the same test sequence. Each result is returned to a separate array.

Figure 8-90
clrtrg example



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

conpin Connect pin

Purpose Connect pins and instruments together.

Format

```
int conpin(int InstrTermID, int connect1, [connectn, […]] 0);
```

`InstrTermID` The instrument terminal ID. For example: SMU1, GNDU, or, PMU1CH1.

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

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

Remarks `conpin` connects every item in the argument list together. 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.

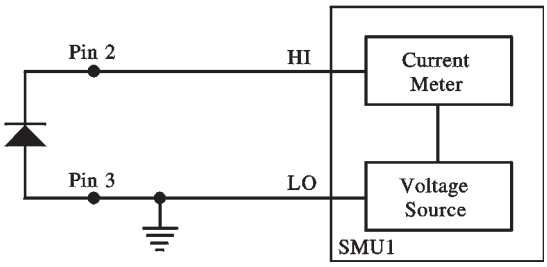
The value -1 will be ignored by `conpin` and is considered a valid entry in the connection list.

With the row-column connection scheme, only one instrument terminal may be connected to a pin.

See also `addcon, clrcon, conpth, delcon`

Example

Figure 8-91
conpin example




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

conpth Connect path

Purpose Connect pins and instruments together using a specific pathway.

Format

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

`path`	Pathway number to use for the connections.
`connect1`	A pin number or an instrument terminal ID.
`connectn`	A pin number or an instrument terminal ID.

Remarks The system can be forced to use a particular pathway by using `conpth` instead of `conpin`. This might be done to provide additional electrical isolation between two connections. The eight pathways are numbered 1 through 8.

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.

The `conpth` function is not valid in the row-column connection scheme. When the matrix is configured for remote sense, the only valid path values are 1, 3, 5, and 7.

See also `addcon, clrcon, conpin, delcon`

delay Delay

Purpose Provides a user-programmable delay within a test sequence.

Format

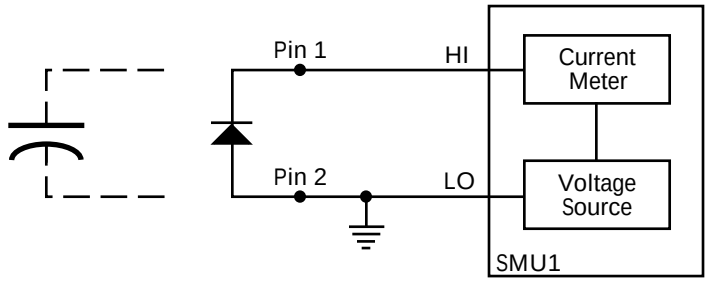
```
int delay(long n);
```

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

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

Example This example measures a variable capacitance diode's leakage current. SMU1 applies 60 V 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 20 ms delay is added. After the delay, current through SMU1 is measured and stored in the variable IR4.

Figure 8-92
delay example




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

delcon Delete connection

Purpose Remove specific matrix connections.

Format

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

`InstrTermID` The instrument terminal ID. For example: SMU1, GNDU, or PMU1CH1.

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

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

Remarks All connections to each terminal or pin listed will be disconnected. Before 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. If an SMU remains connected, GND must be reconnected using `addcon` or an error will be generated when the first LPTLib function after the connection sequence executes.

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.

Format

```
addcon, clrcon, conpin, conpth
```

Example

```
double i1, i2;
conpin(3, GND, 0);
conpin(1, SMU1, 0);
conpin(2, SMU2, 0);
forcev(SMU1, 1.0);
forcei(SMU2, .001);
measi(SMU1, &i1);
delcon(SMU2, 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);
```

Remarks This function will clear all sources sequentially in the reverse order from which they were originally forced. Before 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`, `execut`, and `tstdsl`.

If an RPM is used, use `clrcon` to clear the physical connection to the DUT.

devint Device initialize

Purpose Resets all active instruments; 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 and the default conditions and ranges for the instrumentation. To abort a running `pulse_exec` pulse test, see dev_abort.

Format

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

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

This function performs the following actions:

1) Clear all sources by calling `devclr`.
2) Clear the matrix crosspoints 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`.
6) Resets the active instrument cards in the following order
   a) SMU instrument cards
   b) CVU instrument cards
   c) Pulse instrument cards (4225-PMU, 4220-PGU, and 4205-PG2)

Note that the pulse mode is maintained. For example, if the pulse card is in Segment ARB® mode, it will still be in Segment Arb mode after the `devint` process is complete.

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

disable Disable timer

Purpose Stops the timer and sets the time value to zero (0). Timer reading is also stopped.

Format

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

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

Remarks `disable` (`TIMERn`) stops the timer and resets the time value to zero (0).

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`).

Remarks `enable` (`TIMERn`) initializes and starts the timer and allows other measurements to read the timer. The time starts at zero (0) 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);
```

Remarks This function will wait for all previous LPT commands to complete and then will issue a `devint`.

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.

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`. See Using source compliance limits for details.

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.

To force zero current with a higher voltage limit than the 20 V default, include one of the following calls ahead of the `forcei` call:

- A `measv` call, which causes the 4200-SCS to autorange to a higher voltage limit.
- A `rangev` call to an appropriate fixed voltage, which results in a fixed voltage limit.

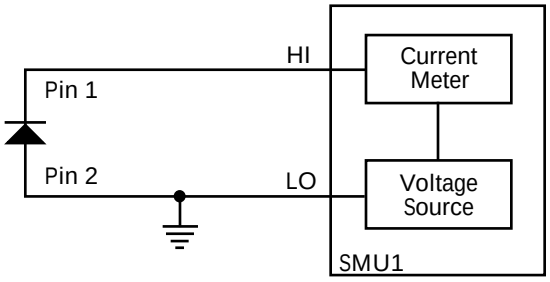
To force zero volts with a higher current limit than the 10 mA default, include one of the following calls ahead of the `forcev` call:

- A `measi` call, which causes the 4200-SCS to autorange to a higher current limit.
- A `rangei` call to an appropriate fixed current, which results in a fixed current limit.

Note that changing the source mode of the SMU can modify the measure range. If the sourcing mode is changed from voltage to current sourcing (or from current to voltage sourcing), the measure range may be changed to minimize variations in the SMU output level. See Changing source mode may change measure range for recommended command order.

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

Figure 8-93
forcei example




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

getinstattr Get configured instrument attributes

Purpose All instruments in the system configuration have specific attributes. GPIB address is an example of an attribute. The values of these attributes change as the system configuration is changed. Therefore, by getting the values of key attributes at run time, user modules can be developed in a configuration-independent manner. Given an instrument identification code and an attribute name string, this module returns the specified attribute value string.

Format

```
int getinstattr(int inst_id, char *attrstr, char *attrvalstr);
```

`inst_id` The LPTLib instrument identifier (ID).

`attrstr` The instrument attribute name string.

`attrvalstr` The value string of the requested attribute. If the requested attribute exists, the returned string will match one of the values shown in the Attribute value string column of Table 8-8. If the requested attribute does not exist, the `attrvalstr` parameter will set to a null string.

Possible values for the `getinstattr` parameters are listed in Table 8-12.

Table 8-12
getinstattr parameter values

Instrument identification code	Attribute name string	Attribute value string
GPIx	GPIBADDR	1-30
	MODELNUM	GPI 2-Terminal GPI 4-Terminal
CMTRx	GPIBADDR	1-30
	MODELNUM	HP4980 KI590
PGUx	GPIBADDR	1-30
	MODELNUM	HP8110
SMUx	MODELNUM	KI4200 KI4210
MTRX1	MODELNUM	KI707 KI708
TF1	MODELNUM	KI8006 KI8007
	NUMOFPINS	12 72
PRBR1	NUMOFPINS	2-72
	MODELNUM	FAKE PA200 MANL MM40
CVUx	MODELNUM	KICVU KICVU4210
VPUx VPUxCH1 VPUxCH2	MODELNUM	KIVPU KIVPU4205 KIVPU4220
PMUx PMUxCH1 PMUxCH2	MODELNUM	KIPMU4225

getinstid Get instrument ID

Purpose Get the instrument identifier (ID) from the instrument name string.

Format

```
int getinstid(char *inst_name, int *inst_id);
```

inst_name The instrument name string.

inst_id The returned instrument identifier.

getinstname Get instrument name

Purpose Get the instrument name string from the instrument identifier (ID).

Format

```
int getinstname(int *inst_id, char *inst_name);
```

inst_id The instrument identifier.

inst_name The returned instrument name string.

GetKiteCycle Returns KITE cycle number.

Purpose This command returns the present KITE cycle number.

Format

```
int GetKiteCycle(void);
```

Remarks If no cycling is active, `GetKiteCycle` returns 1.

GetKiteDevice Returns device being tested.

Purpose This command returns the device that KITE is presently testing.

Format

```
int GetKiteDevice(void);
```

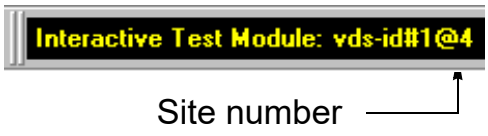
Example This example shows a user test module (UTM) that returns the present subsite, device, and test.

```
char strVal[25];
GetKiteDevice(strVal, 25);
printf("KiteDevice = %s\n", strVal);
```

GetKiteSite Get KITE site number

Purpose Get the site number for the site that KITE is presently testing: The number that KITE displays after the @ symbol in the Test/Plan Indicator box (above the Project Navigator). See Figure 8-94.

Figure 8-94
Getting the site number currently under test



Format

```
int GetKiteSite(void)
```

Remarks The `GetKiteSite` function returns the KITE site number, which is an integer that designates the relative numerical position of the presently tested site in the prober site-visit sequence. However, users normally correlate KITE site numbers with prober site coordinates; note that `GetKiteSite` does *not* return prober site coordinates.

For more information about KITE site numbers, refer to Section 6, Multi-site Project Plan execution and 'Run' execution of Project Plans.

GetKiteSubsite Returns subsite being tested.

Purpose This command returns the subsite number for the site that KITE is presently testing.

Format
```
int GetKiteSubsite(void);
```

Example This example shows a user test module (UTM) that returns the present subsite, device, and test.

```
char strVal[25];
GetKiteSubsite(strVal, 25);
printf("KiteSubsite = %s\n", strVal);
```

GetKiteTest Returns test being run.

Purpose This command returns the test that KITE is presently running.

Format
```
int GetKiteTest(void);
```

Example This example shows a user test module (UTM) that returns the present subsite, device, and test.

```
char strVal[25];
GetKiteTest(strVal, 25);
printf("KiteTest = %s\n", strVal);
```

getlpterr Get LPT error

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

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

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

Returns the first error code encountered since the last `devint`.

getstatus Get instrument status

Purpose Returns the operating state of the specified instrument.

Format

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

inst_id The instrument identifier (ID), such as SMU1, SMU2, VPU1, VPU2.

parameter The parameter of query.

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

Remarks Valid errors:

The `UT_INVLDPRM` invalid parameter error is returned from `getstatus`. The status item parameter is illegal for this device. The requested status code is invalid for selected device.

A list of supported `getstatus` query parameters for an SMU and a pulse generator card (VPU) are provided in Table 8-13 and Table 8-14.

Table 8-13
Supported SMU getstatus query parameters

SMU parameter	Returns	Comment
KI_IPVALUE	The presently programmed output value	Current value (I output value)
KI_VPVALUE		Voltage value (V output value)
KI_IPRANGE	The presently programmed range	Current range (full-scale range value, or 0.0 for autorange)
KI_VPRANGE		Voltage range (full-scale range value, or 0.0 for autorange)
KI_IARANGE	The presently active range	Current range (full-scale range value)
KI_VARANGE		Voltage range (full-scale range value)
KI_COMPLNC	Compliance status of last reading	Bitmapped values: 2 = LIMIT (at the compliance limit set by **limitX**). 4 = RANGE (at the top of the range set by **rangeX**).
KI_RANGE_COMPLIANCE	Range compliance status of the last reading	Returns 1 if in range compliance.

Table 8-14
Supported pulse generator card getstatus query parameters

Parameter		Comment
General parameters:		
KI_VPU_PERIOD	Pulse period	Pulse period value in seconds
KI_VPU_TRIG_POLARITY	Trigger polarity	Rising or falling edge
KI_VPU_CARD_STATUS		Card level status
KI_VPU_TRIG_SOURCE	Trigger source	Trigger source value
Channel based parameters:		
KI_VPU_CH1_RANGE	Source range	Channel 1 range value in volts (5.0 or 20.0)
KI_VPU_CH2_RANGE	Source range	Channel 2 range value in volts (5.0 or 20.0)
KI_VPU_CH1_RISE	Rise time	Channel 1 rise time value in seconds

Table 8-14 (continued)
Supported pulse generator card getstatus query parameters

Parameter		Comment
KI_VPU_ CH2_RISE	Rise time	Channel 2 rise time value in seconds
KI_VPU_CH1_FALL	Fall time	Channel 1 fall time value in seconds
KI_VPU_ CH2_FALL	Fall time	Channel 2 fall time value in seconds
KI_VPU_CH1_WIDTH	Pulse width	Channel 1 pulse width value in seconds
KI_VPU_ CH2_WIDTH	Pulse width	Channel 2 pulse width value in seconds
KI_VPU_CH1_VHIGH	Pulse high	Channel 1 pulse high level value in volts
KI_VPU_ CH2_ VHIGH	Pulse high	Channel 2 pulse high level value in volts
KI_VPU_CH1_VLOW	Pulse low	Channel 1 pulse low level value in volts
KI_VPU_ CH2_ VLOW	Pulse low	Channel 2 pulse low level value in volts
KI_VPU_CH1_DELAY	Pulse delay	Channel 1 pulse delay from trigger value in seconds
KI_VPU_ CH2_DELAY	Pulse delay	Channel 2 pulse delay from trigger value in seconds
KI_VPU_CH1_ILIMIT	Current limit	Channel 1 current Limit value in amps
KI_VPU_ CH2_ILIMIT	Current limit	Channel 2 current Limit value in amps
KI_VPU_CH1_BURST_COUNT	Burst count	Channel 1 burst count value
KI_VPU_ CH2_BURST_COUNT	Burst count	Channel 2 burst count value
KI_VPU_CH1_TEST_STATUS		Channel 1 test status
KI_VPU_CH2_TEST_STATUS		Channel 2 test status
KI_VPU_ CH1_DC_OUTPUT	DC output	Channel 1 DC output value
KI_VPU_ CH2_DC_OUTPUT	DC output	Channel 2 DC output value
KI_VPU_CH1_LOAD	Pulse load	Channel 1 pulse load value
KI_VPU_CH2_LOAD	Pulse load	Channel 2 pulse load value

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.

Remarks This command applies to all timers.

inshld Instrument hold

Purpose Provided for compatibility with Model S400 LPTlib.

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

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 specified period of time. The integration is automatically corrected for 50 Hz or 60 Hz power mains.

Format

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

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

inst_id The measuring instrument identifier (ID), for example, SMU1.

result The variable assigned to the result of the measurement.

Remarks

For a measurement conversion, the signal is sampled for a specific period of time. This sampling time for measurement is called the integration time. For the `intgX` function, the default integration time is set to 1 PLC. For 60Hz line power, 1 PLC = 16.67 ms (1 PLC/60 Hz). For 50 Hz line power, 1 PLC = 20 ms (1 PLC/50 Hz).

The default integration time is one AC line cycle (1 PLC). This default time can be overridden with the `KI_INTGPLC` option of `setmode`**.** The integration time can be set from 0.01 PLC to 10.0 PLC. The `devint` command resets the integration time to the one AC line cycle default value (1 PLC).

NOTE *The only difference between* `measX` *and* `intgX` *is the integration time. For* `measX`*, the integration time is fixed at 0.01PLC. For* `intgX`*, the default integration time is 1 PLC, but can set to any PLC value between 0.01 and 10.0.*

`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 10.0 V on a 4.0 V range. An overrange condition returns the value 1.0E+22 as the measurement result.

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

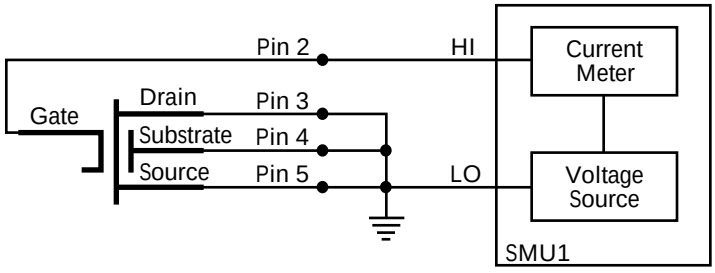
In general, measurement functions which return multiple results are more efficient than performing multiple measurement functions.

Compliance limits – A compliance limit setting goes into effect when the SMU is on a measure range that can accommodate the limit value. For manual ranging, the `rangeX` function is used to select the range. For autoranging, the `intgi` or `intgv` function will trigger a needed range change before the measurement is performed. See Using source compliance limits earlier in this section for details.

Example

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

Figure 8-95
Measuring low leakage current of a MOSFET



```
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 25 V to the gate. */
intgi(SMU1, &idss);/* Measure gate leakage; */
/* return results to idss. */
```

kfpabs Floating point absolute value

Purpose Takes a user-specified positive or negative value and converts it into a positive value that is returned to a specified variable.

Format

```
int kfpabs(double *x, double *z);
```

x	Pointer to the variable to be converted to an absolute value.
z	Pointer to the variable where the result will be stored.

Example This example takes the absolute value of a current reading. `forcev` outputs `vb1` volts from SMU1. This current is measured with `measi`, and the result is stored in location `ares2`. The absolute value of `ares2` is then calculated and stored as `ares2`.

```
double ares2, vb1;
.
.
forcev(SMU1, vb1);/* Output vb1 from SMU1. */
measi(SMU1, &ares2);/* Measure SMU1 current; */
/* store in ares2. */
kfpabs(&ares2, &ares2);/* Convert ares2 to absolute */
/* value; return result to ares2*/
```

kfpadd Keithley floating point add

Purpose Adds two real numbers and stores the result in a specified variable.

Format

```
int kfpadd(double *x, double *y, double *z);
```

x	The first of two values to add.
y	The second of two values to add.
z	A variable in which the sum `x` + `y` will be stored.

Remarks The values referenced by `x` and `y` will be summed and the result will be stored in the location pointed to by `z`. If an overflow occurs, the result will be `±Inf`. If an underflow occurs, the result will be zero (0).

Example This example adds the data in `res1` to the data in `res2`. The result is stored in the `resia` variable.

```
double res1, res2, resia;
.
.
measv(SMU1, &res1);/* Measure SMU1 voltage; store */
/* in res1. */
measi(SMU2, &res2);/* Measure SMU2 current; store */
/* in res2. */
kfpadd(&res1, &res2, &resia);/* Adds res1 and res2; return */
/* result to resia. */
```

.
.

kfpdiv Keithley floating point divide

Purpose Divides two real numbers and stores the result in a specified variable.

Format

```
int kfpdiv(double *x, double *y, double *z);
```

`x`	The dividend.
`y`	The divisor.
`z`	A variable where the result of `x/y` will be stored.

Remarks The value referenced by `x` will be divided by the value referenced by `y`, and the result will be stored in the location pointed to by `z`. If an overflow occurs, the result will be `±Inf`. If an underflow occurs, the result will be zero (0).

Example This example divides the data in `res1` by the data in `res2`. The result is stored in the `resia` variable.

```
double res1, res2, resia;
.
.
measv(SMU1, &res1);/* Measure SMU1 voltage; store */
/* in res1. */
measi(SMU2, &res2);/* Measure SMU2 current; store */
/* in res2. */
kfpdiv(&res1, &res2, &resia);/* Divide res1 by res2; return */
/* result to resia. */
.
.
```

kfpexp Keithley floating point exponential

Purpose Supplies the base of natural logarithms (e) raised to a specified power and stores the result as a variable.

Format

```
int kfpexp(double *x, double *z);
```

x	The exponent.
z	The variable where the result of e^x will be stored.

Remarks e raised to the power of the value referenced by x will be stored in the location pointed to by z. If an overflow occurs, the result will be ±Inf. If an underflow occurs, the result will be zero (0).

Example In this example, kfpexp raises the base of natural logarithms to the power specified by the exponent res4. The result is stored in res4e.

```
double res4, res4e;
.
.
measv(SMU1, &res4);/* Raise the base of natural */
/* logarithms e to the power */
/* res4; */
kfpexp(&res4, &res4e;/* return the result to res4e. */
.
.
```

kfplog Keithley floating point logarithm

Purpose Returns the natural logarithm of a real number to the specified variable.

Format

```
int kfplog(double *x, double *z);
```

x	A variable containing a floating point number.
z	A variable where the result of ln (x) will be stored.

Remarks This function returns a natural logarithm, not a common logarithm. The natural logarithm of the value referenced by x will be stored in the location pointed to by z.

If a negative value or zero (0) is supplied for x, a log of negative value or zero (0) error will be generated and the result will be NaN (not a number).

Example This example calculates the natural logarithm of a real number (res1). The result is stored in logres.

```
double res1, logres;
.
.
measv(SMU1, &res1);/* Measure SMU1; store in res1. */
kfplog(&res1, &logres);/* Convert res1 to a natural */
/* LOG and store in logres. */
.
```

kfpmul Keithley floating point multiply

Purpose Multiplies two real numbers and stores the result as a specified variable.

Format

```
int kfpmul(double *x, double *y, double *z);
```

x A variable containing the multiplicand.

y A variable containing the multiplier.

z The variable where the result of x *y will be stored.

Remarks The value referenced by x will be multiplied by the value referenced by y, and the result will be stored in the location pointed to by z. If an overflow occurs, the result will be ±Inf. If an underflow occurs, the result will be zero (0).

Example This example multiplies variables res1 and res2 and stores the result in variable pwr2.

```
double res1, res2, pwr2;
.
.
measi(SMU1, &res1);/* Measure SMU1 current; */
/* store in res1. */
measv(SMU1, &res2);/* Measure SMU1 voltage; */
/* store in res2. */
kfpmul(&res1, &res2, &pwr2);/* Multiply res1 by res2; */
/* return result to pwr2. */
.
.
```

kfpneg Keithley floating point negative value

Purpose Changes the sign of a value and stores the result as a specified variable.

Format

```
int kfpneg(double *x, double *z);
```

x A variable containing the number to be converted.

z A variable where the result of -x will be stored.

Remarks If the value is positive, it is converted to a negative; if the value is negative, it is converted to a positive.

Example This example changes the sign of a positive voltage reading. forcev outputs a positive 10 V from SMU1. The current is measured with measi, and the result is stored as res4. kfpneg reads res4 and converts the data to a negative value. res4 is then overwritten with the converted value.

```
double res4;
.
.
forcev(SMU1, 10.0);/* Output 10 V from SMU1. */
measi(SMU1, &res4);/* Measure SMU1 current; store */
/* in res4. */
kfpneg(&res4, &res4);/* Convert sign of res4; */
./* return results to res4. */
.
```

kfppwr Keithley floating point power

Purpose Raises a real number to a specified power and assigns the result to a specified variable.

Format

```
int kfppwr(double *x, double *y, double *z);
```

x	A variable containing a floating point number.
y	A variable containing the exponent.
z	A variable where the result of x^y will be stored.

Remarks The value referenced by `x` will be raised to the power of the value referenced by `y`, and the result will be stored in the location pointed to by ***z***. If an overflow occurs, the result will be `±Inf`. If an underflow occurs, the result will be zero (0).

If `x` points to a negative number, a power of a negative number error will be generated, and the result returned will be `-Inf`.

If `x` points to a value of zero (0) and `y` points to a negative number, a divide by zero (0) error will be generated, and the result returned will be `+Inf`.

If `x` points to a value of 1.0, the result will be 1.0, regardless of the exponent.

Example The following example raises the variable `res2` by the power of three. The result is stored in `pwres2`.

```
double res2, pwres2, power=3.0;
.
.
measv(SMU1, &res2);/* Measure SMU1; store */
/* result in res2. */
kfppwr(&res2, &power,
 &pwres2);/* res2 to the third power; */
/* return result to pwres2. */
.
```

kfpsqrt Keithley floating point square root

Purpose Performs a square root operation on a real number and returns the result to the specified variable.

Format

```
int kfpsqrt(double *x, double *z);
```

`x`	A variable containing a floating point number.
`z`	A variable where the result, the square root of `x`, will be stored.

Remarks The square root of the value referenced by `x` will be stored in the location pointed to by `z`.

If `x` points to a negative number, a square root of negative number error will be generated, and the result will be `NaN` (not a number).

Example This example converts a real number (`res1`) into its square root. The result is stored in `sqres2`.

```
double res1, sqres2;
.
.
measv(SMU1, &res1);/* Measure SMU1; store result */
./* in res1. */
kfpsqrt(&res1, &sqres2);/* Find square root of res1; */
/* return result to sqres2. */
.
```

kfpsub Keithley floating point subtract

Purpose Subtracts two real numbers and stores their difference in a specified variable.

Format

```
int kfpsub(double *x, double *y, double *z);
```

x A variable containing the minuend.

y A variable containing the subtrahend.

z The variable where the result of *x* - *y* will be stored.

Remarks The value referenced by *y* will be subtracted from the value referenced by *x*, and the result will be stored in the location pointed to by *z*. If an overflow occurs, the result will be ±`Inf`. If an underflow occurs, the result will be zero (0).

Example This example subtracts `res2` from `res1`. The result is returned to `diff2`.

```
double res1, res2, diff2;
.
.
measv(SMU1, &res1);/* Measure SMU1; store result */
/* in res1. */
measv(SMU2, &res2);/* Measure SMU2; store result */
/* in res2. */
kfpsub(&res1, &res2, &diff2);/* Subtract res2 from res1; */
./* return the place with */
/* result to diff2. */
.
```

kibcmd Keithley GPIB command

Purpose Enables universal, addressed, and unaddressed GPIB bus commands to be sent through the GPIB interface. These commands can consist of any command that is valid with the ATN line asserted, such as `DCL`,`SDC`,`GET`, and so on. The following table lists these GPIB commands.

Format

```
int kibcmd(unsigned int timeout, unsigned int numbytes, char*
cmdbuffer);
```

timeout The timeout for transfer (in 100 ms ticks).

numbytes	The number of BYTES in `cmdbuffer` to send with the ATN line asserted.
`cmdbuffer`	The array containing the BYTES to transfer over the GPIB interface.

Table 8-15
GPIB command list

GPIB command	Data byte (Hex)	Comments
Universal		
`LLO` (local lockout)	11	Locks-out front panel controls.
`DCL` (device clear)	14	Returns instrument to default conditions.
`SPE` (serial poll enable)	18	Enables serial polling.
`SPD` (serial poll disable)	19	Disables serial polling.
Addressed		
`SDC` (selective device clear)	04	Returns instrument to default conditions.
`GTL` (go to local)	01	Sends go to local.
`GET` (group execute trigger)	08	Triggers instrument for reading.
Un-addressed		
`UNL` (unlisten)	3F	Removes all listeners from GPIB bus.
`UNT` (untalk)	5F	Removes any talkers from GPIB bus.
`LAG` (listen address group)	20-3E	Place instrument at this primary address (0 through 30) in listen mode.
`TAG` (talk address group)	40-5E	Place instrument at this primary address (0 through 30) in talk mode.
`SCG` (secondary command group)	60-7E	Place instrument at this secondary address (0 through 30) in listen mode.

Remarks `kibcmd` performs the following steps:

1. Asserts attention (ATN).
2. Sends byte string (command buffer).
3. De-asserts ATN.

Example This example illustrates how the `kibcmd` command could be used to issue a GPIB bus trigger command to a GPIB instrument located at address 15:

```
int status;
char GPIBtrigger[5] = {0x3F, 0x2F, 0x08, 0x3F, 0x00};
/* Unlisten = 3F (UNL) */
/* Listen address = 32 + 15 = 2F */
/* Group Execute Trigger (GET) = 08 */
/* UNL */
/* Terminate string with NULL */
.
.
.
status = kibcmd(30, strlen(GPIBtrigger),GPIBtrigger);
/* Use 3s timeout */
```

kibdefclr Keithley GPIB define device clear

Purpose

Defines the device-dependent command sent to an instrument connected to the GPIB 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 1 through 30 are valid (the controller uses address 31).

sec_addr The secondary address of the instrument; numbers 1 through 30 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 100 ms units (for example, timeout = 40 = 4.0 s).

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.

Remarks

Each call to `kibdefclr` copies parameters into a data structure within the tester memory. These data structures are allocated dynamically. After the execution of the command buffer using `execut`, these tables are cleared. 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 effect on `bsweep`, because `bsweep` issues a `devclr` to clear active sources. It is not recommended to use GPIB instruments when performing `bsweep` tests.

kibdefdelete Delete GPIB definition strings for devclr and devint

Deletes all command definitions previously made with the `kibdefclr` (Keithley GPIB define device clear) and `kibdefint` (Keithley GPIB define device initialize) commands. Once this command is issued, any previous definitions made using `kibdefclr` or `kibdefint` will no longer occur at `devint` or `devclr` time.

Format

```
int kibdefdelete( void );
```

Remarks

This function can be overridden by re-issuing the `kibdefint` and `kibdefclr` commands.

kibdefint Keithley GPIB define device initialize

Purpose

Defines a device-dependent command sent to an instrument connected to the GPIB 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; 1 through 30 is valid (the GPIB uses address 31).

`sec_addr` The secondary address of the instrument; 1 through 30 is valid. If the instrument device does not support secondary addressing, this parameter must be -1.

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

`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 is allowed.

Remarks

Each call to `kibdefclr` and `kibdefint` copies parameters to a data structure within tester memory. These data structures are allocated dynamically. After the execution of the command buffer using `execut`, these tables are cleared, and 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 before `devclr` and `devint` execution. This may be a problem when communicating with any Keithley-supported GPIB instruments. This may also have an effect on `bsweep`, because `bsweep` issues a `devclr` to clear ALL active sources. It is not recommended to use GPIB instruments when performing `bsweep` 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, unsigned int *rcv_len, char
*rcv_buff);
```

`pri_addr`	The primary address of the instrument; 1 through 30 is valid (the GPIB controller uses address 31).
`sec_addr`	The secondary address of the instrument; 1 through 30 is valid. If the instrument device does not support secondary addressing, this parameter must be -1.
`term`	The ASCII delimiter character of the returned string. This is the byte used for terminating data buffer read.
`timeout`	The GPIB transfer timeout. This timeout is in 100 ms units (for example, timeout = 40 = 4.0 s).
`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.

Remarks

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

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

`kibrcv`: `rcv_size` defines the maximum number of bytes physically allowed in the buffer. If `rcv_size` is greater than the byte string returned by the instrument, then the device is short-cycled and only the maximum number of bytes is 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; 1 through 30 is valid. The controller uses GPIB address 31.

`sec_addr` The secondary address of the instrument; 1 through 30 is valid. If the instrument device does not support secondary addressing, this parameter must be -1.

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

`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.

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 *statusbyte);
```

`pri_addr` The primary address of the instrument; 1 through 30 is valid (the controller uses GPIB address 31).

`sec_addr` The secondary address of the instrument; 1 through 30 is valid. If the instrument device does not support secondary addressing, this parameter must be -1.

`timeout` The GPIB polling timeout. This timeout is in l00 ms units (for example, timeout = 40 = 4.0 s).

`statusbyte` The serial poll status byte returned by the device presently being polled. The `statusbyte` variable must be an integer.

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) Poll GPIB interface until data is available.
8) Read Serial_Poll_BYTE from device (if data is available), else.
9) Serial_Poll_BYTE = 0 (indicating error; device not SRQing).
10) Assert ATN.
11) Send serial poll disable (SPD).
12) Send UNTalk (UNT).
13) Send UNListen (UNL).
14) De-assert ATN.

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 status byte is returned.

Format

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

`pri_addr`	The primary address of the instrument; 2 through 31 is valid.
`sec_addr`	The secondary address of the instrument; 1 through 31 is valid. If the instrument device does not support secondary addressing, this parameter must be -1.
`timeout`	The GPIB polling timeout. The timeout is in l00 ms units (for example, timeout = 40 = 4.0 s).
`stausbyte`	The serial poll status byte variable name returned by the device presently being polled.

Remarks `kibsplw` performs the following steps:

15) Wait with timeout for general SRQ assertion on the GPIB.
16) Call `kibspl`.

kspcfg Configure the port

Purpose Configures and allocates a serial port for RS-232 communications.

Format

```
int kspcfg(int port, int baud, int databits, int parity, int
stopbits, int flowctl);
```

`port`	The RS-232 port to be used. Currently only port 1 is supported.
`baud`	The transmission rate that will be used. Valid rates are: 2400, 4800, 9600, 14400, and 19200 baud.

`databits`	The number of data bits that will be used. Valid inputs are 7 or 8 bits.
`parity`	Determines whether or not parity bits will be transmitted. Valid inputs are: 0 (no parity), 1 (odd parity), or 2 (even parity).
`stopbits`	Sets the number of stop bits to be transmitted. Valid inputs are: 1 or 2.
`flowctl`	Determines the type of flow control that will be used. Valid inputs are: 0 (no flow control), 1 (XON/XOFF flow control), or 2 (hardware).

Remarks Port 1 must not be allocated to another program or utility when using the `ksp` (Keithley Serial Port) commands.

- The `databits`, `parity`, `stopbits`, and `flowctl` settings must match those on the instrument or device that you wish to control.
- Using a flow control setting of 0 may result in buffer overruns if the device or instrument that you are controlling has a high data rate.
- If you use a flow-control setting of 2 (hardware), you must make sure that the RS-232 cable has enough wires to handle the RTS/CTS signals.

Example Here is an example of how you would use `kspcfg` to set port 1 to 19200 baud, 8 data bits, odd parity, 1 stop bit, and xon/xoff flow control:

```
int status;
.
.
.
status = kspcfg(1, 19200, 8, 1, 1, 1);/* port 1, 19200 baud,
   8 bits, odd parity,
   1 stop bit, and
   xon-xoff flow ctl */
```

kspdefclr Define string to clear RS-232 instrument on devclr

Purpose Defines a device-dependent character string sent to an instrument connected to a serial port. This string is sent during the normal tester `devclr` process. It ensures that if the tester is calling `devclr` internally, any device connected to the configured serial port will be cleared with the given string.

Format

```
int kspdefclr(int port, double timeout, double delay, int
buffsize, char *buffer);
```

port	The RS-232 port to be used. Currently only port 1 is supported. This port must have been previously configured for communications with the `kspcfg` command.
timeout	The serial communications timeout. The valid input range is 0 to 600 s.
delay	The amount of time to delay after sending the string to the serial device. The valid input range is 0 to 600 s.
buffsize	The length of the string to send to the serial device.

buffer A character string containing the data to send to the serial device.

Remarks Before issuing this command, you must configure the serial port using the `kspcfg` command.

- The commands sent to the serial device are issued in the order in which they were defined using the `kspdefclr` command.
- The `kspdefdelete` command can be used to delete any previous definitions.
- The `kspdefclr` and `kspdefint` command strings are sent prior to normal (for example, an SMU) instrument `devclr` and `devint` execution.

kspdefdelete Delete RS-232 definition strings for devclr and devint

Purpose Deletes all command definitions previously made with the `kspdefclr` (Keithley Serial Define Device Clear) and `kspdefint` (Keithley Serial Define Device Initialize) commands. Once this command is issued, any previous definitions made using `kspdefclr` or `kspdefint` will no longer occur at `devint` or `devclr` time.

Format

```
int kspdefdelete( void );
```

Remarks This function can be overridden by re-issuing the original `kspdefint` and `kspdefclr` commands.

kspdefint Define string to clear RS-232 instrument on devint

Purpose :Defines a device-dependent character string sent to an instrument connected to a serial port. This string is sent during the normal tester `devint` process. It ensures that if the tester is calling `devint` internally, any device connected to the configured serial port will be cleared with the given string.

Format

```
int kspdefint(int port, double timeout, double delay, int
buffsize, char *buffer);
```

port The RS-232 port to be used. Currently only port 1 is supported. This port must have been previously configured for communications with the `kspcfg` command.

timeout The serial communications timeout. The valid input range is 0 to 600 seconds.

delay The amount of time to delay after sending the string to the serial device. The valid input range is 0 to 600 seconds.

buffsize The length of the string to send to the serial device.

buffer A character string containing the data to send to the serial device.

Remarks Before issuing this command, you must configure the serial port using the `kspcfg` command.

- The commands sent to the serial device are issued in the order in which they were defined using the `kspdefclr` command.
- The `kspdefdelete` command can be used to delete any previous definitions.
- The `kspdefclr` and `kspdefint` command strings are sent prior to normal (for example, an SMU) instrument `devclr` and `devint` execution.

ksprcv Receive device-dependent command string

Purpose Used to read data from an instrument connected to a serial port.

Format

```
int ksprcv(int port, char terminator, double timeout,
int rcvsize, int *rcv_len, char *rcv_buffer);
```

port	The RS-232 port to be used. Currently only port 1 is supported. This port must have been previously configured for communications with the `kspcfg` command.
terminator	The ASCII terminator for the received data. This character is used to terminate the read.
timeout	The serial communications timeout. The valid input range is 0 to 600 seconds.
rcvsize	The physical buffer size. This is used to control the maximum number of characters that can be read from the device.
rcv_len	The actual number of characters read from the device. This value is returned to the `ksprcv` command by the software.
rcv_buffer	A character array in which to store the data returned from the serial device.

kspsnd Send device-dependent string

Purpose Sends a device-dependent command to an instrument attached to a RS-232 serial port.

Format

```
int kspsnd(int port, double timeout, int cmdlen, char *cmd);
```

port	The RS-232 port to be used. Currently only port 1 is supported. This port must have been previously configured for communications with the `kspcfg` command.
timeout	Serial communications timeout. The valid input range is 0 to 600 s.
cmdlen	The number of characters that you are sending out the serial port.
cmd	The character array containing the data that you want sent out of the serial port.

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 ±10.0 V. A `limiti (SMU1, 1.5E-3)` limits the current of the voltage source SMU1 to ±I.5 mA.

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.*

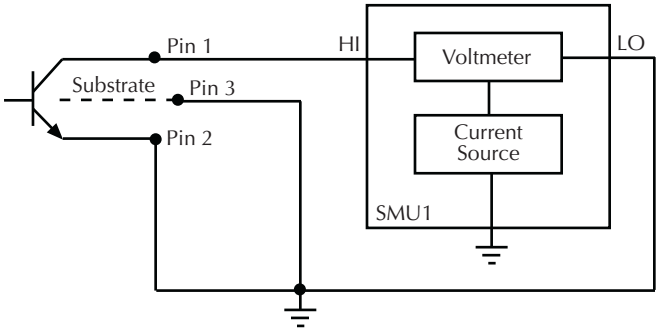
This function must be called in the test sequence before the associated `forceX`, `pulseX`, `bsweepx`, `sweepX`, or `searchX` function is used to generate the voltage or current. `limitX` also sets the top measurement range of an autoranged measurement.

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

If you need a voltage limit greater than 20 V at an SMU that has been set to force zero current, use a `measv` call to set the SMU to autorange to a higher range, OR use `rangev` to set a higher voltage range. Similarly, if you need a current limit of greater than 10 mA at an SMU that has been set to force zero volts, use a `measi` call to set the SMU to autorange to a higher range OR use **rangev** to set a higher current range.

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

Figure 8-96
Measuring device breakdown voltage




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

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 or amperes.

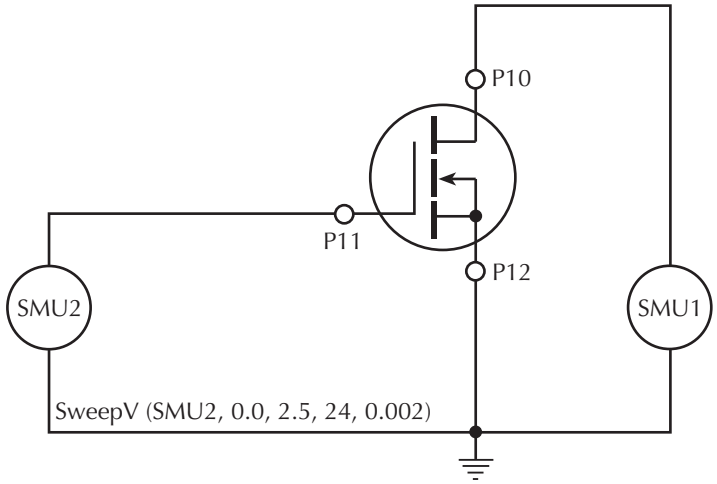
Remarks `lorange` is used with autoranging to limit the number of range changes, and thus saves test time.

If the instrument was on a range lower than the one specified by `lorange`, the range is changed. The 4200-SCS automatically provides any range change settling delay that may be necessary due to this potential range change.

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

Example This example illustrates how you would select the bottom autorange limit.

Figure 8-97
Defining bottom autorange limit




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

measX Measure

Purpose Allows the measurement of voltage, current, or time.

Format

```
int measi(int inst_id, double *result);
int meast(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.

Remarks For a measurement conversion, the signal is sampled for a specific period of time. This sampling time for measurement is called the integration time. For the measX function, the integration time is fixed at 0.01 PLC. For 60 Hz line power, 0.01 PLC = 166.67 µs (0.01 PLC/60 Hz). For 50 Hz line power, 0.01 PLC = 200 µs (0.01 PLC/50 Hz).

NOTE *The only difference between* measX *and* intgX *is the integration time. For* measX*, the integration time is fixed at 0.01 PLC. For* intgX*, the default integration time is 1 PLC, but can set to any PLC value between 0.01 and 10.0.*

After the function 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 function. 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 10.0 V on a 4.0 V range. An overrange condition returns the value 1.0E+22 as the result of the measurement.

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

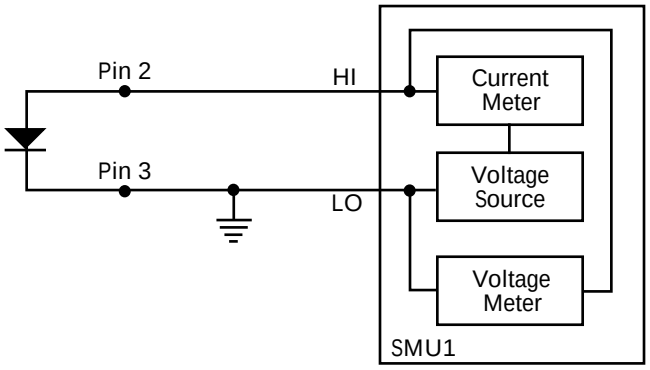
All measurements except meast invoke a timer snapshot measurement to be taken by all enabled timers. This timer snapshot can then be read with the meast command.

In general, measurement functions which return multiple results are more efficient than performing multiple measurement functions.

Compliance limits: A compliance limit setting goes into effect when the SMU is on a measure range that can accommodate the limit value. For manual ranging, the rangeX function is used to select the range. For autoranging, the measi or measv function will trigger a needed range change before the measurement is performed. See Using source compliance limits for details.

Example In this example, the diode's forward bias voltage is obtained from a single SMU.

Figure 8-98
measX




```
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 50 mA. */
measv(SMU1, &vf47);/* Measure forward bias; */
/* return result to vf47. */
```

mpulse Measure pulse

Purpose This function uses an SMU to force a voltage pulse and measures both the voltage and current for exact device loading.

Format

```
int mpulse(int inst_id, double pulse_amplitude, double
pulse_duration, double *v_meas, double *i_meas);
```

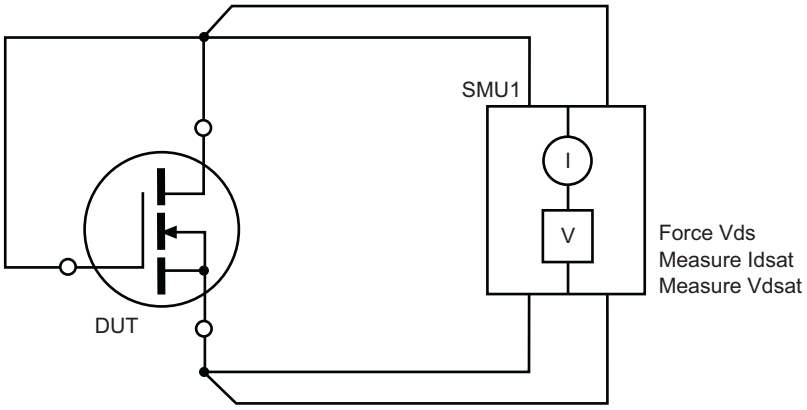
inst_id	The name of the instrument under control.
pulse_amplitude	The pulse height in volts.
pulse_duration	The pulse width in seconds. The measurements will be taken at the end of the pulse before the `mpulse` is shutdown.
v_meas	The variable used to receive the voltage on the output of the instrument at the time the pulse terminates.
i_meas	The variable used to receive the current drawn from the instrument. 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.

Pulsing is useful for devices that exhibit self-heating, which could damage the device or shift operating characteristics. Examples are high-power GaAs transistors or BJTs, but even some silicon devices exhibit self-heating.

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 ms. Voltage across the MOS transistor, Vdsat, and drain current, Idsat, are measured.

Figure 8-99
mpulse




```
double vdsat, idsat, vds;
.
.
mpulse(SMU1, vds, 1.0E-3,/* Pulse output of SMU1. */
 &vdsat, &idsat);
```

pulseX Pulse of a voltage or current

Purpose This function directs a SMU to force a voltage or current at a specific level for a predetermined length of time.

Format

```
int pulsei(int inst_id, double forceval, double time);
int pulsev(int inst_id, double forceval, double time);
```

inst_id The instrument identification code.

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, 10 e-3)` generates +10.0 V for 10 ms, and a `pulsei (SMU1, -1.5E-3, 10 e-3)` generates -1.5 mA for 10 ms.

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

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

Remarks After `pulseX` is executed, the output is turned off. In order to perform measurements, the output must be turned back on. `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 made using the `pulseX` and `measX` functions are made after the pulse has completed.

NOTE *When the source is not operating, measurements are not allowed.*

Whenever `pulseX` is executed, either a default or a programmed current or voltage limit is in effect. Refer to the limit command for additional information.

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

Note that changing the source mode of the SMU can modify the measure range. If the sourcing mode is changed from voltage to current sourcing (or from current to voltage sourcing), the measure range may be changed to minimize variations in the SMU output level. See Changing source mode may change measure range for recommended command order.

Example

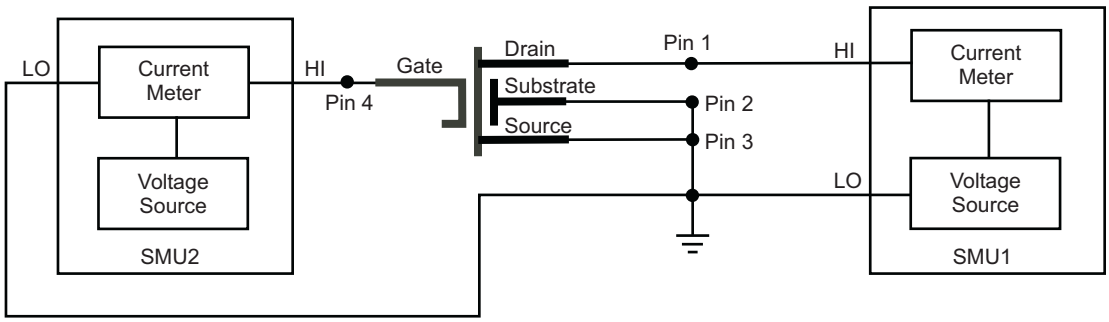
Here the threshold voltage shift of an FET is measured by performing two `searchv` functions, as follows:

a) `searchv` measures the gate voltage required to initiate a drain current of 10 µA.

b) `searchv` measures 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 because `clrtrg`, clear trigger, was not used.

Figure 8-100
pulseX




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

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 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.

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 10.0 V on a 2.0 V range. The value `1.0E+22` is returned when this occurs.

`rangeX` can also reference a source, because an SMU can be either of the following:

- Simultaneously a voltage source, voltmeter, and current meter.
- Simultaneously a current source, current meter, and voltmeter.

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

Compliance limits – 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. See Using source compliance limits earlier in this section for more information.

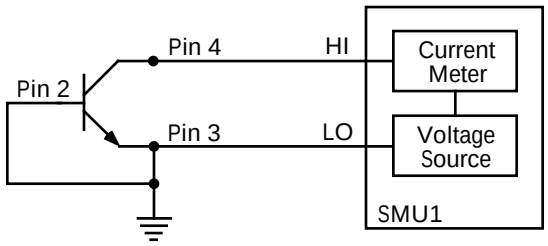
When changing the instrument range, be careful not to overrange the instrument. For example, a test initially performed in the 10 mA range with a 5 mA limit is changed to test in the 1 mA range with a 1 mA limit. Notice that the limit is lowered from 5 mA to 1 mA to avoid overranging the 1 mA setting.

Changing source mode may change measure range

When changing the source mode of the SMU, the measure range may change. This change minimizes variations in the SMU output level. The source mode of the SMU refers to its voltage sourcing or current sourcing capability. Changing the source mode means using a command (such as `forceX`) to change the SMU mode from forcing voltage to forcing current (or from forcing current to forcing voltage). For example, if the SMU is programmed to force voltage (`forcev`), and then is programmed with to force current (`forcei`), to ensure a consistent output signal the previously programmed current measure range may change. Make sure the desired measure range is set by sending the `rangeX` command after switching the source mode. The commands that can change the source mode are `asweepX`, `bsweepX`, `forceX`, `pulseX`, `searchX`, and `sweepX`.

Example

Figure 8-101
rangeX




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

rdelay Realtime delay

Purpose A user-programmable delay in seconds.

Format

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

n The desired delay duration in seconds.

Example The following example measures a variable capacitance diode's leakage current. SMU1 presets 60 V 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 20 ms 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 60 V from SMU1. */
rdelay(0.02);/* Pause for 20 ms. */
measi(SMU1, &ir4);/* Measure current; return */
/* result to ir4. */
```

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.

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

When the test sequence is executed, the sweep function 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.

Example Refer to the examples for the `smeasx` and `sweepx` functions.

savgX Sweep average

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

Format

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

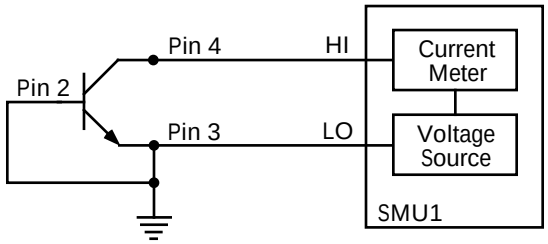
Remarks This function is used to create an entry in the measurement scan table. During any of the sweeping functions, a measurement scan is performed for every force point in the sweep. During each scan, a measurement will be made for every entry in the scan table. The measurements are made in the same order in 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 that 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 -50 V. Capacitance is measured 26 times during the sweep. The results are stored in an array called `res1`.

Figure 8-102
savgX




```
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 -50 V over 25 steps.*/
```

scnmeas Scan measure

Purpose To perform a single measurement on multiple instruments at the same time.

Format

```
int scnmeas(void);
```

Remarks This function behaves like a single point sweep. It performs a single measurement on multiple instruments at the same time. Any forcing or delaying must be done prior to calling `scnmeas`.

`smeasX`, `sintgX`, or `savgX` must be used to set up result arrays just as is done for a sweep call. Each call to `scanmeas` will add one element to the end of each array.

Calls to `scnmeas` may be mixed with calls to `sweepX`, and all results will be appended to the result arrays in the same way multiple `sweepX` calls behave.

searchX Binary search measurement

Purpose Used to determine the voltage or current required to obtain a desired current or voltage. 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, long
iterate_no, double iterate_time, double *result);
```

```
int searchv(int inst_id, double min_val, double max_val, long
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.

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:

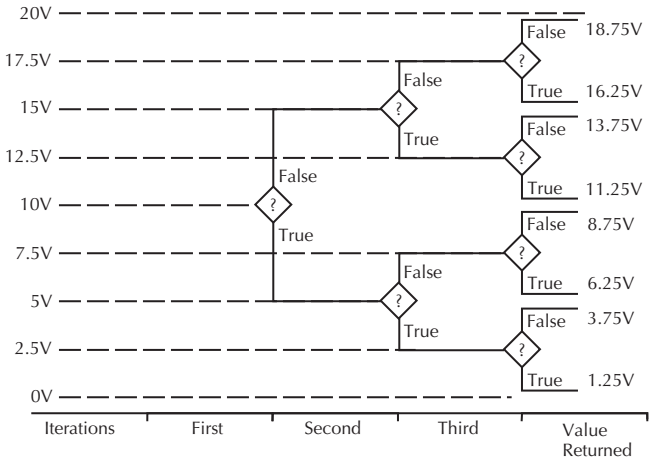
- A voltage or current is applied to a circuit node of the DUT.
- All triggers are evaluated.
- If the triggers evaluate true, the source value is moved toward the `min_val`. Otherwise, the source value is moved toward `max_val`. The source range is then divided in half for the next iteration.

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 20 V, 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, 10 V is 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 8-103
searchX



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

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

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

- Call `limitX` and `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.

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

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

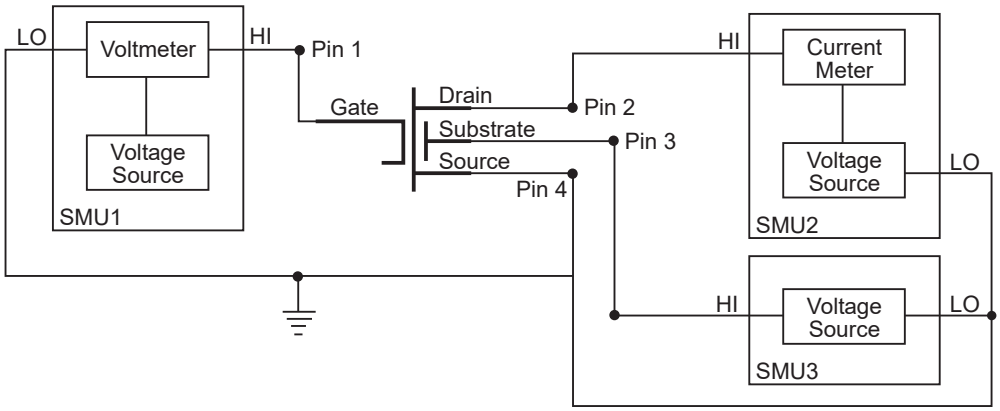
$$\frac{20}{2^{(16 + 1)}} = 1.2\text{mV}$$

Note that changing the source mode of the SMU can modify the measure range. If the sourcing mode is changed from voltage to current sourcing (or from current to voltage sourcing), the measure range may be changed to minimize variations in the SMU output level. See Changing source mode may change measure range for recommended command order.

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 V through 1.7 V 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 8-104
searchv



```
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 1µA. */
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. */
```

setauto Re-enable Autoranging

Purpose

Re-enables autoranging and cancels any previous `rangeX` command for the specified instrument.

Format

```
int setauto(int inst_id);
```

`inst_id` The instrument identification code.

Remarks

When an instrument is returned to the autorange mode, it will remain in its present range for measurement purposes. The source range will change immediately.

Due to the dual mode operation of the SMU (v versus i), `setauto` places both voltage and current ranges in autorange mode.

Example

```
double icer1;
double idatvg[25];
.
.
rangei(SMU1, 2.0E-9);/* Select manual range. */
delay(200);/* Delay after range change. */
measi(SMU1, &icer1);/* Measure leakage. */
.
.
setauto(SMU1);/* Enable autorange mode. */
lorangei(SMU1, 2.0E-6);/* Select 2µA 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 2.5 V.
*/
```

setmode Set component mode

Purpose

Set instrument-specific operating mode parameters.

Format

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

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

`modifier` Instrument specific operating characteristic to change. See Table 8-16.

`value` Value to set the operating parameter to.

Remarks

`Setmode` allows control over certain instrument-specific operating characteristics. Refer to the appendix for the specific instrument for more information on what each instrument supports.

A special instrument ID called KI_SYSTEM is used to set operating characteristics of the system.

For modifier values, refer to Table 8-16.

Table 8-16
Modifiers

Support	Parameters			Comment
	instr_id	*modifier*	*value*	
Supported	KI_SYSTEM	KI_TRIGMODE	KI_MEASX KI_INTEGRATE KI_AVERAGE KI_ABSOLUTE KI_NORMAL	Redefines all existing triggers to use a new method of measurement.
		KI_AVGNUMBER	<value>	Number of readings to take when KI_TRIGMODE is set to KI_AVERAGE.
		KI_AVGTIME	<value> (in units of seconds)	Time between readings when KI_TRIGMODE is set to KI_AVERAGE.
No-Op (Accepted but not responded to)[1]		KI_MX_DEFMODE	KI_HIGH KI_LOW	Sets the default matrix mode to high current mode or low current mode. This setting will remain in effect until the end of the current session and is not reset by **devint**.
		KI_HICURRENT	KI_ON	Forces the matrix into high current mode. The mode will revert to the default at the next **devint** unless the configuration file sets this parameter to reset on a **clrcon**.
		KI_CC_AUTO	KI_ON KI_OFF	Turns automatic compliance clear processing on or off. **devint** will reset this value to KI_ON.
		KI_CC_SRC_DLY	<value>	The minimum time after a source value change before a compliance clear scan may start. This represents the time after a source value change it takes the circuit under test to settle and prevent false compliance detection due to transients.
		KI_CC_COMP_DLY	<value>	The time between compliance scans while processing **compclr**. This also represents the time after a source value change it takes the circuit under test to settle and prevent false compliance detection due to transients, but the source value changes are only due to removing instrument from an artificial compliance state.
		KI_CC_MEAS_DLY	<value>	The minimum time after the last source value change before a measurement can be made. This represents the time it takes the circuit under test to settle to the level desired for the subsequent measurements.
Supported	SMUn	KI_INTGPLC	<value> (in units of line cycles)	Specifies the integration time the SMU will use for the **intgx** and **sintgx** commands. The default **devint** value is 1.0. The valid range is 0.01 to 10.0.
		KI_AVGMODE	KI_MEASX KI_INTEGRATE	Controls what kind of readings are taken for **avgX** calls. The **devint** default value is KI_MEASX. When KI_INTEGRATE is specified, the integration time used is that specified by the KI_INTGPLC **setmode** call.

Table 8-16 (continued)
Modifiers

Support	Parameters			Comment
	instr_id	*modifier*	*value*	
	SMUn (continued)	KI_DELAY_FACTOR	<value>	This factor scales the internal delay times used by the SMU. A value larger than one increases the delays; a value less than one decreases the delays. A minimum delay will be enforced by the SMU. NOTE: This command should not be used when setting the SMU speed to FAST, NORMAL, or QUIET modes; the delay factor is set internally by these modes so changing the value while using one of the predefined modes corrupts the speed settings or the delay factor.
No-Op (Accepted but not responded to)[1]	SMUn (continued)	KI_IMTR		Sets up the SMU as a current meter. The ranges used are representative of the type of instrument being simulated. NOTE: This **setmode** will turn the source on.
			KI_S400	Sets the SMU to use ranges equivalent to the Model S400.
			KI_DMM	Sets the SMU to use ranges equivalent to a DMM (lowest range = 100 µa). Provides a lower resolution, fast measurement. Used for high current applications.
			KI_ELECTROMETER	Sets the SMU to use ranges equivalent to an electrometer. Provides best measurement resolution, but has a slower measurement time. Used for low current measurements.
		KI_LIM_INDCTR	Any	Controls what measure value is returned if the SMU is at its programmed limit. The **devint** default is SOURCE_LIMIT (7.0 e22). NOTE: The SMU always returns INST_OVERRANGE (1.0 e22) 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 an indicator value.
No-Op (Accepted but not responded to)[1]	SMUn (continued)	KI_VMTR		Sets up the SMU as a volt meter. The ranges used are representative of the type of instrument being simulated. Note: This **setmode** will turn the source on.
			KI_S400	Sets the SMU to use ranges equivalent to the Model S400.
			KI_DMM	Sets the SMU to use ranges equivalent to a DMM. Provides a low impedance, fast measurement. Used for low voltage applications.
			KI_ELECTROMETER	Sets the SMU to use ranges equivalent to an electrometer. Provides a high input impedance, but has a slower measurement time. Used for high resistance measurements.

1. These modifiers perform no operations in the 4200-SCS. They are included only for compatibility, so that existing S600 programs using the setmode function can be ported to the 4200-SCS without problems.

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.

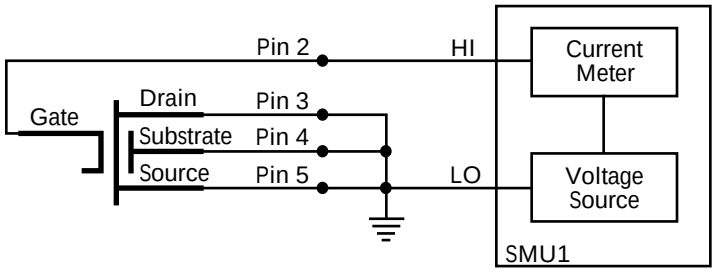
`results` The floating point array where the results are stored.

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 25.0 V.

Figure 8-105
sintgX



```
double 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 */
./* 25 V; each step 1ms in */
./* duration. */
```

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.

Remarks

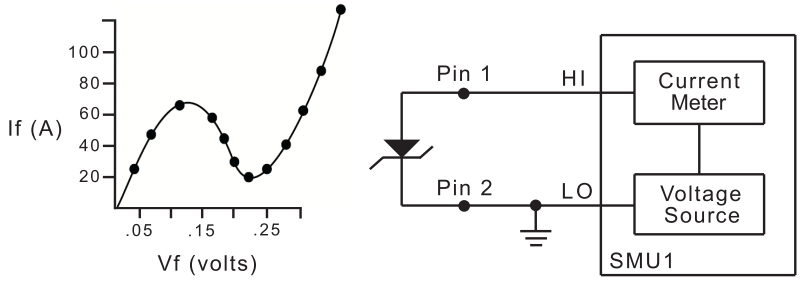
This function is used to create an entry in the measurement scan table. During any of the sweeping functions, a measurement scan is performed for every force point in the sweep. During each scan, a measurement will be made for every entry in the scan table. The measurements are made in the same order in 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 that 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 0.3 V. The current through the diode is sampled 13 times with a duration of 25 ms at each step. The results are stored in an array called `resi`.

Figure 8-106
smeasX




```
double resi[13];/* Defines array. */
double vf [13];
.
.
.
conpin(SMU1, 1, 0);
conpin(GND, 2, 0);
rtfary (vf);/* Return the voltage force array*/
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. */
```

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, long
stepno, double step_delay);
```

```
int sweepv(int inst_id, double startval, double endval, long
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.

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`, `trigXl`, and `trigcomp` 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:

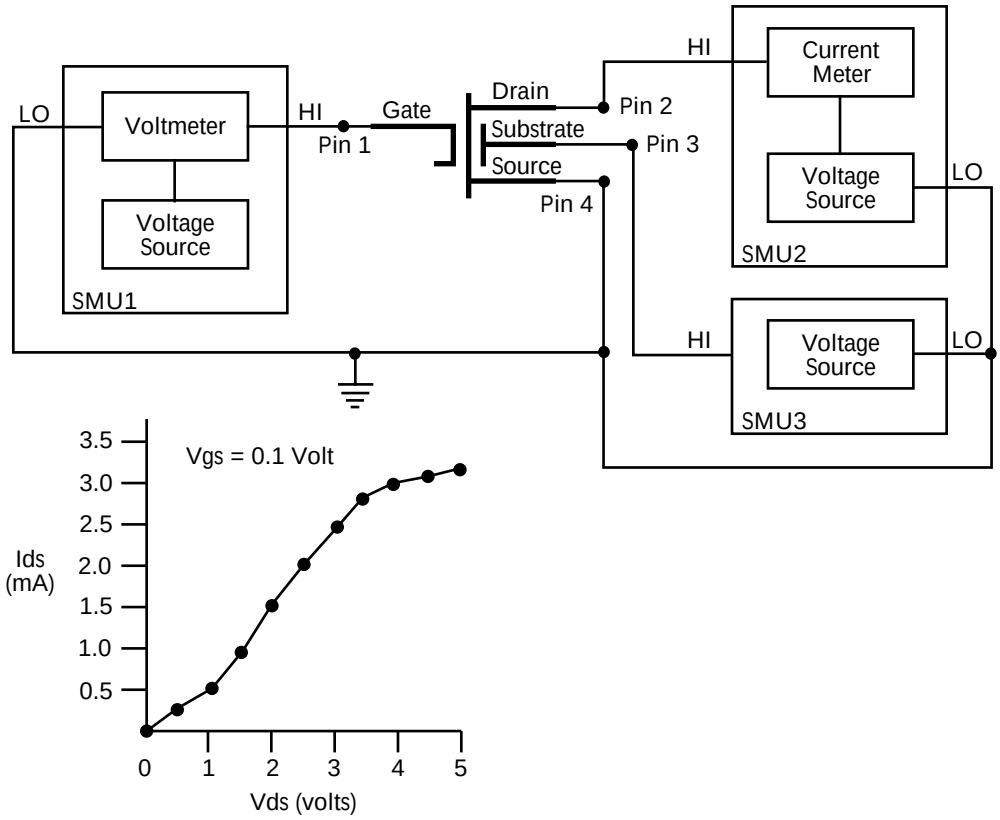
```
double ires[6];
conpin(SMU1, 10, 0);
conpin(2, GND 0);
forcev(SMU1, 5.0);/* Force 5 V to charge. */
delay(10);/* Wait for precharge. */
smeasi(SMU1, ires);/* Set up measurement. */
sweepv(SMU1, 5.0, 10.0, 5, 2.5E-3);/* Do the real measurement.
*/
```

Note that changing the source mode of the SMU can modify the measure range. If the sourcing mode is changed from voltage to current sourcing (or from current to voltage sourcing), the measure range may be changed to minimize variations in the SMU output level. See Changing source mode may change measure range for recommended command order.

Example

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

Figure 8-107
sweepX




```
double resi[11], ssbiasv;
double vds[11];
.
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. */
rtfary(vds);/* Return force array*/
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 2.5 ms duration, */
/* ranging from 0 to 5 V. */
```

trigcomp Trigger on compliance

Purpose This function will cause a trigger when an instrument goes in or out of compliance.

Format

```
int trigcomp(int instr_id, int mode);
```

`instr_id`	The ID of the instrument the trigger is set to.
`mode`	Specifies whether to trigger when an instrument is in or out of compliance. Use 1 to trigger when in compliance. Use 0 to trigger when out of compliance.

Remarks This function will cause LPT to monitor the given instrument for compliance. A trigger can be set when the instrument is either in compliance or out of compliance, based on the specified mode.

trigXg, trigXl trigXg: trigger if greater than; trigXl: trigger if less than

Purpose Monitors for a predetermined level of voltage, current, 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, 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`.

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.

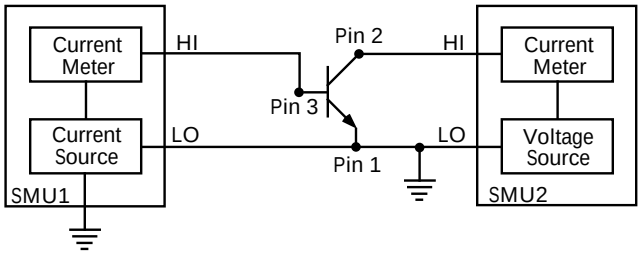
The specified trigger point is automatically cleared when a `clrtrg`, `devint`, or `execut` is executed.

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 5 mA 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 µA and 200 µA in 15 iterations. `trigig` continuously monitors the current through SMU1. The base current supplied by SMU1 is stored as the result `res22`.

Figure 8-108
trigXg, trgX1




```
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 5 mA. */
searchi(SMU1, 5.0E-5, 2.0E-4,
 15, 1.0E-3, &res22);/* Generate a current ranging */
/* from 50 to 200µA in */
/* 15 iterations. Return the */
/* current resulting from the */
/* last iteration as res22. */
```

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 V to 0.0 V in 18 steps with a 1 ms duration at each step. `trigil` stops the ramping initiated by `sweepv` when the current through the diode is greater than +4 mA. `smeasi` measures the current applied by SMU1 at each voltage step and stores the results in array `res1`.

```
double res1[20];
.
.
conpin(SMU1, 1, 0);
conpin(GND,2, 0);
trigil(SMU1, +4.0E-3);/* If less than +4mA, */
/* stop ramping. */

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

tstdsl Test station deselect

Purpose Used to deselect a test station.

Format
```
tstdsl (void);
```

Remarks To relinquish control of an individual test station, a new test station must now be selected using `tstsel` before any subsequent test control functions are run.

The `tstdsl` command has the same effect as the `tstsel (0)` command.

NOTE *`tstdsl` is not required for use in a UTM.*

Example
```
tstdsl ( );  /* Disables test station.*/
```

See also `tstsel`

tstsel Test station select

Purpose Used to enable or disable a test station.

Format
```
tstsel (int x);
```

Where `x` is the test station number, 0 or 1.

Remarks `tstsel` is normally called at the beginning of a test program.

`tstsel (1)` will select the first test station and load the instrumentation configuration.

NOTE *`tstsel` is not required for use in a UTM.*

See also `tstdsl`

LPT functions for the Model 4205-PG2

The following information explains the functions included in the Keithley LPTLib (Linear Parametric Test Library) for the 4205-PG2 pulse generator card. The functions are summarized in Table 8-11.

NOTE *All pulse functions are supported by the 4205-PG2 pulse generator card. Most pulse functions are also supported by the 4200-PG2 if the firmware is upgraded to KITE 6.2. Instructions for upgrading firmware are available by clicking the Complete Reference icon on the 4200-SCS desktop. Follow the links for Release Notes, then look for the Firmware Upgrade Procedure for the pulse card firmware.*

NOTE *The terms "pulse generator card" and "PG2" is used for functions that pertain to both the 4205-PG2 and 4200-PG2. Operations that are not supported by the 4200-PG2 are explained.*

arb_array Defines a full-arb waveform

Purpose This function is used to define a full-arb waveform and name the file.

Format

```
int arb_array(INSTR_ID instr_id, long ch, double TimePerPt, long
length, double *levelArr, char *fname);
```

instr_id	Instrument ID of the PG2: VPU1, VPU2, and so on.
`ch`	The PG2 channel: 1 or 2.
`TimePerPt`	Sets the time interval between waveform points: 20 ns to 1 s.
`length`	The number of points (values): 262,144 maximum.
`levelArr`	An array of voltage values for each point in the waveform.
`fname`	A name for the full-arb waveform.

Remarks This function is used to define the number of points in a waveform, the time interval between points, and the voltage value at each point. The maximum number of waveform points per channel is 262,144.

The load time for a full-arb waveform is proportional to the number of points. The total time to load full-size full-arb waveforms for both channels is around one minute.

Once loaded, use pulse_output to turn on the appropriate channels, and then use pulse_trig to select the trigger mode and start (or arm) pulse output.

Refer to Full Arb in Section 11 for details on this pulse mode. The following voltage level array is required for the example full-arb waveform shown in Table 8-17.

Table 8-17
arb_array

Level array	Level array (continued)
levelArr(0) = 0.5	levelArr(41) = 19.5
levelArr(1) = 1.0	levelArr(42) = 19.0
levelArr(2) = 1.5	levelArr(43) = 18.5
•	•
•	•
•	•
levelArr(39) = 19.5	levelArr(79) = 0.5

.kaf waveform file for KPulse: The arbitrary waveform data defined by the arb_file function can be copied into a `.kaf` file. Use a text editor to properly format the file. The `.kaf` file can then be imported into KPulse. By default, `.kaf` waveform files for KPulse are saved in the `ArbFiles` folder at the following command path location: `C:\S4200\kiuser\KPulse\ArbFiles`. Refer to Section 13 for details on using KPulse.

See also **`seg_arb_define`**

arb_file Loads a waveform from a full-arb waveform file

Purpose

This function is used to load a waveform from an existing full-arb waveform file.

Format

```
int arb_file(INSTR_ID instr_id, long ch, char *fname)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`ch`	The PG2 channel: 1 or 2.
`fname`	The name of the waveform file.

Remarks

This function is used to load a waveform from an existing full-arb `.kaf` waveform file into the pulse generator card. A full-arb waveform can be loaded for each channel of the pulse generator card. Once loaded, use pulse_output to turn on the appropriate channel, and then use pulse_trig to select the trigger mode and start (or arm) pulse output.

When specifying the `fname`, include the full command path with the file name. Existing `.kaf` waveforms are typically saved in the `ArbFiles` folder at the following command path location:

```
C:\S4200\kiuser\KPulse\ArbFiles
```

A full-arb waveform can be created using KPulse, and then saved as a `.kaf` waveform file (refer to Section 13 for details).

A waveform in an existing `.kaf` file can be modified in two ways:

- Use a text editor to modify.
- Import into KPulse and then modify.

See also

arb_array, **seg_arb_file**, **seg_arb_define**

Example

The following function loads a full-arb file named `SINE.kaf` (saved in the `ArbFiles` folder) into the pulse generator card for Channel 1:

```
arb_file(VPU1, 1, "C:\\S4200\\kiuser\\KPulse\\ArbFiles\\SINE.kaf")
```

pg2_init Resets PG2 to default settings for specified pulse mode

Purpose Use this function to change the pulse mode. It resets the pulse generator card to the specified pulse mode (standard, full-arb, or Segment ARB) and its default conditions

Table 8-18
pg2_init

Standard pulse defaults	Full Arb and Segment ARB pulse defaults
Pulse high and pulse low = 0 V	Source range = 5 V; fast speed
Source range = 5 V; fast speed	Pulse count = 1
Pulse period = 1 µs	Pulse delay = 0 s
Pulse width = 500 ns	Pulse load = 50 Ω
Pulse count = 1	Pulse trigger source = Software
Rise and fall time = 100 ns	Trigger mode = Continuous
Pulse delay = 0 s	Pulse trigger output = Off
Pulse load = 50 Ω	Trigger polarity = Positive
Pulse trigger source = Software	Current limit = 105 mA
Trigger mode = Continuous	Pulse output = Off
Pulse trigger output = Off	
Trigger polarity = Positive	
Complement mode = Normal Pulse	
Current limit = 105 mA	
Pulse output = Off	

Format

```
int pg2_init(INSTR_ID instr_id, long mode)
```

`instr_id` Instrument ID of the PG2: VPU1, VPU2, and so on.

`mode` Pulse mode: 0 (standard pulse), 1 (Segment Arb) or 2 (full-arb).

Remarks This function resets both channels of the PG2 to the default settings of the specified pulse mode. The default setting for each parameter is listed in the Format section for each LPT function.

The pulse_init function can be used to reset the pulse generator card to the default settings for the presently selected pulse mode.

Example This function resets the PG2 to the Segment ARB pulse mode and its default settings:

```
pg2_init(VPU1, 1)
```

pulse_burst_count Sets count for pulse burst mode

Purpose For the burst mode, this function sets the number of pulses to output during a burst sequence.

Format

```
int pulse_burst_count(INSTR_ID instr_id, long chan, unsigned
long count)
```

`instr_id` Instrument ID of the PG2: VPU1, VPU2, and so on.

`chan` Channel number of the PG2: 1 or 2.

`count` Number of pulses to output: 1 to (2^{32}-1).

Default： `1`

Remarks Each channel of the PG2 can have a unique burst count. When a burst sequence is triggered, the PG2 will output the specified number of pulses and then stop. The pulse_trig function is used to start (or arm) the burst sequence (`Burst` or `Trig Burst`).

NOTE With an external trigger source selected, the burst count for channel 1 cannot be less than the burst count for channel 2. Setting the burst count for channel 2 higher than the burst count for channel 1 may cause your system to stop responding when pulse output is triggered to start. Also, when using one channel, set the unused channel to the same burst count value. See pulse_trig_source for details on selecting an external trigger source.

Example The following function sets the burst count for the PG2 channel 1 to 10:

```
pulse_burst_count(VPU1, 1, 10)
```

pulse_current_limit Sets current limit for the PG2

Purpose This function sets the current limit of the PG2.

Format

```
int pulse_current_limit(INSTR_ID instr_id, long chan, double ilimit)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`ilimit`	Current limit value (in amps, range and load dependent): 5 V range: -0.2 to +0.2 20 V range: -0.8 to +0.8 Default: `0.105 (5 V range)`

Remarks Current limit can be independently set for each PG2 channel. Current limit values are range dependent and can be set from -0.2 A to +0.2 A (5 V range, 50 Ω load) or -0.4 A to +0.4 A (20 V range, 50 Ω load). Current limit is used to protect the DUT by using the specified DUT load to calculate the voltage required to reach the current limit. A PG2 channel will not exceed the voltage required to reach the set current limit value at the specified DUT load.

See also **`pulse_load`**

Example The following function sets the current limit of PG2 channel 1 to 1 mA:

```
pulse_current_limit(VPU1, 1, 1e-3)
```

pulse_dc_output Selects DC output and sets voltage level

CAUTION **The** `pulse_vlow`**,** `pulse_vhigh`**, and** `pulse_dc_output` **commands set the voltage value output by the pulse channel when it is turned on (using** `pulse_output`**). If the output is already enabled, these commands will change the voltage level immediately, even before the pulsing is started with a** `pulse_trig` **command.**

Purpose This function selects the DC output mode and sets the voltage level.

Format

```
int pulse_dc_output(INSTR_ID instr_id, long chan, double dcvalue)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`dcvalue`	DC voltage output value (in volts, range, and load dependent): 5 V range: -5 to +5 20 V range: -20 to +20 Default: N/A

Remarks Each PG2 channel can be set to output a fixed DC voltage level, rather than pulses. The DC output for each PG2 channel can be set from -5 V to +5 V (5 V range) or -20 V to +20 V (20 V range) for 50 Ω load.

See also `pulse_load`

Example The following function selects channel 1 DCV output and sets voltage to +10 V:

```
pulse_dc_output(VPU1, 1, 10)
```

pulse_delay Sets time delay from trigger to pulse output

Purpose This function sets the delay period from trigger to when pulse output starts.

Format

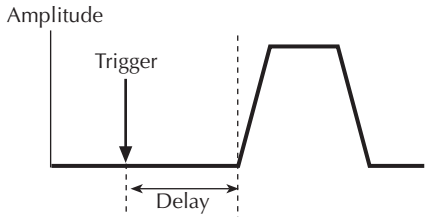
```
int pulse_delay(INSTR_ID instr_id, long chan, double delay)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`delay`	Time delay in seconds: Fast speed: 0 to (Period - 10 e-9) Slow speed: 0 to (Period - 10 e-9) Default: 0

Remarks Pulse delay can be set independently for each PG2 channel. For both speeds, pulse delay can be set from 0 ns to (Period - 10 ns). The pulse_range function is used to set pulse speed.

As shown below, pulse delay is the time from pulse trigger initiation to the start of the rise transition time.

Figure 8-109
pulse_delay



The maximum pulse delay that can be set depends on the presently set period for the pulse. For example, if the period is set for 500 ns, the maximum pulse delay that can be set is 490 ns (500 ns - 10 ns = 490 ns).

NOTE *Use the pulse_source_timing function to set the pulse delay time for the Models 4220-PGU and 4225-PMU.*

See also `pulse_period`, `pulse_trig`

Example The following function sets the pulse delay for channel 1 to 300 ns:

```
pulse_delay(VPU1, 1, 300e-9)
```

pulse_fall Sets pulse fall time

Purpose This function sets the fall transition time for the PG2 pulse output.

Format

```
int pulse_fall(INSTR_ID instr_id, long chan, double fallt)
```

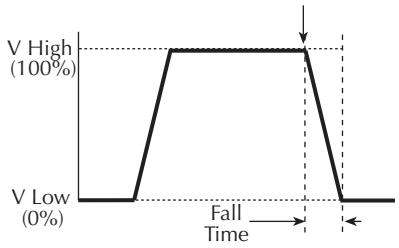
`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`fallt`	Pulse fall time in seconds (floating point number) Fast speed: 10 e-9 to 33 e-3 Slow speed[7]: 50 e-9 to 33 e-3 Default: 100 e-9

Remarks Rise and fall transition time can be set independently for each PG2 channel. There is a minimum slew rate for both the rise and fall transitions. For the fast speed range, the minimum is 362 µV/µs, or 1 V/2.7 ms. For the high voltage range, the minimum slew rate is 1.8 mV/µs, or 1 V/500 µs.The `pulse_range` function is used to set pulse speed.

As shown below, the pulse fall time occurs between the 100 percent and 0 percent amplitude points on the falling edge of the pulse, where the amplitude is the difference between the V High and V Low pulse values.

7. Note that the minimum transition time for pulse source only (no measurement) on the 40 V range is 50 ns for the 4225-PMU and 4220-PGU. The 4200-PG2 and 4205-PG2 have a minimum transition time of 100 ns.

Figure 8-110
pulse_fall



The pulse fall time setting takes effect immediately during continuous pulse output. Otherwise, the fall time setting takes effect when the next trigger is initiated. The `pulse_trig` function is used to trigger continuous or burst output.

NOTE *Use the* `pulse_source_timing` *function to set the pulse fall time for the Models 4220-PGU and 4225-PMU.*

See also **`pulse_rise`**

Example For fast speed, the following function sets the pulse fall time for channel 1 of the PG2 to 50 ns:

```
pulse_fall(VPU1, 1, 50e-9)
```

pulse_halt Stops pulse output

Purpose This function stops all pulse output from the PG2.

Format

```
int pulse_halt(INSTR_ID instr_id)
```

`instr_id` Instrument ID of the PG2: VPU1, VPU2, and so on.

Remarks This function stops all pulse output from the PG2 and turns the PG2 channels off. Pulse output can be restarted by first turning the outputs back on with `pulse_output` and then using the `pulse_trig` function to restart the test.

See also **`pulse_output`**

Example The following function stops PG2 pulse output:

```
pulse_halt(VPU1)
```

pulse_init Resets PG2 to default settings for the present pulse mode

Purpose This function resets the PG2 to the default settings for whichever pulse mode (standard, full-arb, or Segment ARB) is presently selected:

Table 8-19
pulse_init

Standard pulse defaults	Full Arb and Segment ARB pulse defaults
Pulse high and pulse low = 0 V	Source range = 5 V: Fast speed
Source range = 5 V: fast speed	Pulse count = 1
Pulse period = 1 µs	Pulse delay = 0 s
Pulse width = 500 ns	Pulse load = 50 Ω
Pulse count = 1	Pulse trigger source = Software
Rise and Fall time = 100 ns	Pulse trigger mode = Continuous
Pulse delay = 0 s	Pulse trigger output = Off
Pulse load = 50 Ω	Trigger polarity = Positive
Pulse trigger source = Software	Current limit = 105 mA
Pulse trigger mode = Continuous	Pulse output = Off
Pulse trigger output = Off	
Trigger polarity = Positive	
Complement mode = Normal pulse	
Current limit = 105 mA	
Pulse output = Off	

Format

```
int pulse_init(INSTR_ID instr_id)
```

`instr_id` Instrument ID of the PG2: VPU1, VPU2, and so on.

Remarks This function resets both channels of the PG2 to the default settings. The default setting for each parameter is listed in the Format section for each LPT function.

The `pg2_init` function can be used to specify which pulse mode to reset. Using this function selects the specified pulse mode and its default settings.

See also **pg2_init**

Example The following function resets the PG2 to the default settings for the presently selected pulse mode:

```
pulse_init(VPU1)
```

pulse_load Sets output impedance for the load

Purpose This function sets the output impedance for the load (DUT).

Format

```
int pulse_load(INSTR_ID instr_id, long chan, double load)
```

`instr_id` Instrument ID of the PG2: VPU1, VPU2, and so on.

`chan` Channel number of the PG2: 1 or 2.

`load` Output impedance (in ohms): 1 to 10 e6

Default: 50

Remarks DUT impedance can be independently set for each PG2 channel. The DUT impedance can be set from 1 Ω to 10 M Ω, depending on the programmed pulse high and low values.

Maximum power transfer is achieved when the DUT impedance matches the output impedance of the PG2. For example, if the DUT impedance is set to 1 MΩ, the voltage output settings will change to account for the higher DUT impedance, ensuring that the voltage at the DUT will not be double the voltage setting (caused by reflection due to load mismatching).

Example The following function sets the output impedance of PG2 channel 1 to 100 Ω:

```
pulse_load(VPU1, 1, 100).
```

pulse_output Sets pulse output on or off

Purpose This function sets the pulse output of a PG2, PGU, or PMU (without RPM) channel on or off. This command also prepares the pulse source when using a PMU with RPMs. See the first note in this function's remarks section for details.

Format

```
int pulse_output(INSTR_ID instr_id, long chan, long out_state)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`out_state`	Pulse output state: 0 (off) or 1 (on). Default: 0 (off)

Remarks When a PG2 channel is off, the output is in a high-impedance (open) state. The command configures the channel to output and close the output relay. This connects the source to the device under test (DUT). When using the PG2 commands (see Table 8-16 on page 8-146), the pulse channel starts pulsing when a trigger is initiated. Note that if a pulse delay has been set, pulse output will start after the delay period expires. Also, `devclr()` resets the VPU source and disconnects the source from the DUT.

NOTE *When using a 4225-RPM with a PMU, the pulse_output command is necessary to prepare the channel for sourcing, but does not close the output relay. Use the rpm_config command to enable the RPM output.*

This function does not control the high-endurance output relays (HEORs) on the Model 4205-PG2 card. The `pulse_ssrc` function controls (open/close) the HEORs, and the `seg_arb_define` function is used to define a Segment ARB® waveform, which includes HEOR control.

NOTE *It is good practice to routinely turn off the outputs of the PG2 after a test has been completed.*

See also **`pulse_delay`**, **`pulse_trig`**, **`pulse_current_limit`**

Example The following function turns off the output for PG2 channel 1:

```
pulse_output(VPU1, 1, 0)
```

pulse_output_mode Sets pulse output mode

Purpose This function sets the pulse output mode of a PG2 channel.

Format

```
int pulse_output_mode(INSTR_ID instr_id, long chan, long mode)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`mode`	Pulse output state: NORMAL (0) or COMPLEMENT (1). Default: NORMAL

Remarks When a PG2 channel is in COMPLEMENT mode, the V Low and V High voltage settings are swapped.

Example The following function sets the output mode for PG2 channel 1 to COMPLEMENT:

```
pulse_output_mode(VPU1, 1, COMPLEMENT)
```

pulse_period Sets pulse period

Purpose This function sets the period for pulse output.

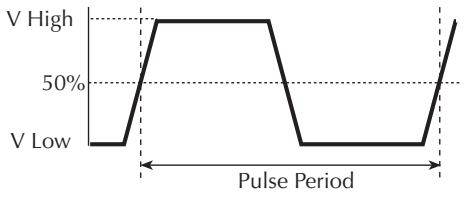
Format

```
int pulse_period(INSTR_ID instr_id, double period)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`period`	Pulse period (in seconds): 5 V range: 20 e-9 to 1 20 V range: 500 e-9 to 1 Default: 1 e-6

Remarks This function sets the pulse period for both channels of the PG2. As shown below, the pulse period is measured at the median point (50 percent between the high and low pulse values) from the rising edge of a pulse to the rising edge of the next pulse.

Figure 8-111
pulse_period



The pulse period setting takes effect immediately during continuous pulse output. Otherwise, the period setting takes effect when the next trigger is initiated. The `pulse_trig` function is used to trigger continuous or burst output.

Example The following function sets the pulse period of the PG2 to 200 ns:

```
pulse_period(VPU1, 200e-9)
```

pulse_range Sets pulse voltage range (low or high)

Purpose

Sets a PG2 channel for low voltage (fast speed) or high voltage (slow speed).

Format

```
int pulse_range(INSTR_ID instr_id, long chan, double range)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`range`	Pulse range (in volts): 5 or 20. Default: 5 V

Remarks

Setting the pulse range of a PG2 channel to 5 V selects the low-voltage range. Selecting the low-voltage range also selects fast speed for pulse output. For fast speed, the minimum pulse width that can be set is 10 ns, and minimum rise / fall times can be set to 10 ns.

Setting the pulse range of a PG2 channel to 20 V selects the high-voltage range. Selecting the high-voltage range also selects slow speed for pulse output. For slow speed, the minimum pulse width that can be set is 250 ns, and the minimum rise / fall times can be set to 100 ns.

This setting takes effect when the next trigger is initiated. The following pulse parameters are then checked: period, width, rise time, fall time, and high and low voltage levels. If any of these parameters is out of bounds, it is reset to the default value.

NOTE *When using the* `pulse_range` *function, changing the source range after setting voltage levels, in any pulse mode, may result in voltage levels invalid for the new range setting. Therefore, use* `pulse_range` *before setting the voltage levels.*

NOTE *This function (*`pulse_range`*) can also be used to set the voltage source range of the 4220-PGU and 4225-PMU. Use the* `pulse_ranges` *function to set the source and measure ranges of the Model 4225-PMU.*

See also

`pulse_fall`, **`pulse_vhigh`**, **`pulse_vlow`**, **`pulse_period`**, **`pulse_rise`**, **`pulse_width`**

Example

The following function selects the high-voltage (slow speed) range for PG2 channel 1:

```
pulse_range(VPU1, 1, 20).
```

pulse_rise Sets pulse rise time

Purpose This function sets the rise transition time for the PG2 pulse output.

Format

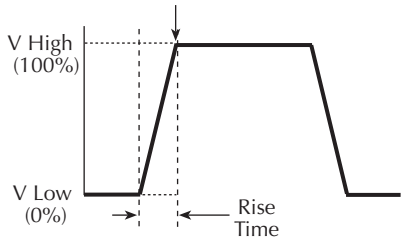
```
int pulse_rise(INSTR_ID instr_id, long chan, double riset)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`riset`	Pulse rise time in seconds (floating point number) Fast speed: 10 e-9 to 33 e-3 Slow speed[8]: 50 e-9 to 33 e-3 Default: 100 e-9

Remarks Rise and fall transition time can be set independently for each PG2 channel. There is a minimum slew rate for both the rise and fall transitions. For the fast speed range, the minimum is 362 µV/µs, or 1 V/2.7 ms. For the high-voltage range, the minimum slew rate is 1.8 mV/µs, or 1 V/500 µs.The `pulse_range` function is used to set pulse speed.

As shown below, the pulse rise time is measured from the 0 percent and 100 percent amplitude points on the rising edge of the pulse, where the amplitude is the difference between the V High and V Low values.

Figure 8-112
pulse_rise



The pulse rise time setting takes effect immediately during continuous pulse output. Otherwise, the rise time setting takes effect when the next trigger is initiated. The `pulse_trig` function is used to trigger continuous or burst output.

NOTE *Use the `pulse_source_timing` function to set the pulse rise time for the 4220-PGU and 4225-PMU.*

See also **`pulse_fall`**

Example For fast speed, the following function sets the pulse rise time for channel 1 of the PG2 to 50 ns:

```
pulse_rise(VPU1, 1, 50e-9)
```

8. Note that the minimum transition time for pulse source only (no measurement) on the 40 V range is 50 ns for the 4225-PMU and 4220-PGU. The 4200-PG2 and 4205-PG2 have a minimum transition time of 100 ns.

pulse_ssrc Control the high endurance output relays on the Model 4205-PG2

Purpose

This function controls the high endurance output relay (HEOR) for each output channel of the Model 4205-PG2.

Format

```
int pulse_ssrc(INSTR_ID instr_id, long chan, long state, long ctrl)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`state`	0 (Open) or 1 (Close)
`ctrl`	0 (Auto), 1 (Manual) or 2 (Trigger out driven) Default: 1 (Close), 0 (Auto)

Remarks

When a high-endurance output relay (HEOR) for a Model 4205-PG2 channel is opened, the output for that channel will be electrically isolated from the DUT. Note that this setting is independent of the output relay (see `pulse_output`).

The `ctrl` parameter determines how the HEOR will be controlled. When set to 0 (auto), the Segment ARB pulse mode will control the HEOR. When set to 2 (trigger out driven), the relay state will follow the trigger output. When `ctrl` is set to 1 (manual), the `state` parameter will open (0) or close (1) the HEOR.

The 4200-PG2 does not have high endurance output relays. Calling the `pulse_ssrc` function for the 4200-PG2 will return an error.

Example

The following function selects Manual control and opens the HEOR:

```
pulse_ssrc(VPU1, 1, 0, 1)
```

pulse_trig Select trigger mode and initiate or arm pulse output

Purpose

This function selects the trigger mode (continuous, burst, or trig burst) and initiates the start of pulse output or arms the pulse generator card.

Format

```
int pulse_trig(INSTR_ID instr_id, long mode)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`mode`	Trigger mode: 0 (burst), 1 (continuous) or 2 (trig burst)

Remarks

With the software trigger source selected, this function will set the trigger mode (continuous, burst, or trig burst) for both pulse generator card channels, and initiate the start of pulse output.

Model 4205-PG2 only: With an external trigger source selected, this function will set the trigger mode, and arm the Model 4205-PG2. Pulse output will start when an external trigger is applied to the Trigger In connector. The 4200-PG2 does support input triggering (no Trigger In connector).

The `pulse_trig_source` function is used to select the trigger source for the Model 4205-PG2.

In the continuous trigger mode, the PG2 will output pulses continuously. For burst and trig burst, the PG2 will output the programmed number of pulses and then stop.

NOTE *See Triggering in Section 11 for details on triggering.*

If pulse delay is set to zero (0), pulse output will start immediately after it is triggered. If pulse delay is >0, pulse output will start after the delay period expires.

See also `pulse_burst_count`, `pulse_delay`

Example The following function initiates (triggers) burst pulse output:

```
pulse_trig(VPU1, 0)
```

pulse_trig_output Sets output trigger on or off

Purpose This function sets the output trigger on or off.

Format

```
int pulse_trig_output(INSTR_ID instr_id, long state)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`state`	Output trigger state: 0 (off) or 1 (on) Default: 1 for standard pulse, 0 for Segment ARB and full arb

Remarks This function turns the TTL level trigger output pulse on or off. The pulse used to synchronize pulse output with the operations of an external instrument. When connected to a scope, each output pulse of the PG2 will trigger a scope waveform measurement.

See also `pulse_trig_polarity`

Example The following function sets the PG2 trigger output on:

```
pulse_trig_output(VPU1, 1)
```

pulse_trig_polarity Sets polarity of the output trigger

Purpose This function sets the polarity (positive or negative) of the PG2 output trigger.

Format

```
int pulse_trig_polarity(INSTR_ID instr_id, long polarity)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`polarity`	Output trigger polarity: 0 (negative) or 1 (positive) Default: 1 (positive, rising edge)

Remarks Trigger output provides a TTL level output that is at the same frequency (period) as the PG2 output channels. It is used to synchronize pulse output with the operations of an external instrument. When connected to a scope, each output pulse of the PG2 will trigger a scope waveform measurement.

The external instrument that is connected to the PG2 external trigger may require a positive-going (rising-edge) pulse or a negative-going (falling-edge) pulse for triggering. If using the scope card, the PG2 output trigger must be set for positive polarity.

If a `polarity` value other than 0 or 1 is sent, it will map to 0 or 1 in the following manner:

```
if(polarity <= 0)
pol = NEGATIVE;
else
pol = POSITIVE;
```

NOTE *4220-PGU and 4225-PMU: Do not use the two external **falling** trigger sources (see `pulse_trig_source` function) with the positive trigger output polarity on the master card that triggers itself and other subordinate cards. These two falling trigger sources should only be used when an external piece of equipment is being used to supply the trigger pulses to the 4220-PGU and 4225-PMU. This applies to all three pulse modes (standard pulse, Segment ARB and full arb).*

NOTE *When triggering multiple Model 4205-PG2 cards in a master/subordinate configuration, changing the master trigger output polarity (`pulse_trigger_polarity` function) will result in a transition in the trigger output levels that may be interpreted as a trigger pulse by the other cards.*

See also **`pulse_trig_output`**

Example The following function sets the PG2 trigger output for negative polarity:

```
pulse_trig_polarity(VPU1, 0)
```

pulse_trig_source Sets trigger source

Purpose This function sets the trigger source

Format

```
int pulse_trig_source(INSTR_ID instr_id, long source)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`source`	Trigger source: 0 (software) 1 (external – initial trigger only – rising) 2 (external – initial trigger only – falling) 3 (external – trigger per pulse – rising) 4 (external – trigger per pulse – falling) 5 (internal trigger bus) Default: Software

Remarks This function sets the trigger source that will be used to trigger the Model 4205-PG2 to start its output. With the software trigger source selected, the `pulse_trig` function will select the trigger mode (continuous, burst, or trig burst), and initiate the start of pulse output.

NOTE *Models 4220-PGU and 4225-PMU: Do not use the two external* ***falling*** *trigger sources with the positive trigger output polarity (see* `pulse_trig_polarity` *function) on the master card that triggers itself and other subordinate cards. These two falling trigger sources should only be used when an external piece of equipment is being used to supply the trigger pulses to the Models 4220-PGU and 4225-PMU. This applies to all three pulse modes (standard pulse, Segment ARB and full arb).*

NOTE *The 4200-PG2 does not support input triggering. Calling the* `pulse_trig_source` *function for the 4200-PG2 will return an error.*

NOTE *Because trigger source is really a card-level setting and not a channel setting, using channel 1 or 2 will set the card to the specified source card 1. Similarly, channel 3 or 4 will set the source for card 2.*

With an external trigger source selected, the `pulse_trig` function will select the trigger mode and arm pulse output. Pulse output will start when the required external trigger pulse is applied to the Trigger In connector. There is a trigger-in delay of 560 ns. This is the delay from the trigger-in pulse to the time of the rising edge of the output pulse.

For an initial trigger only setting, only the first rising or falling trigger pulse will start and control pulse output.

For a trigger per pulse setting, rising or falling edge trigger pulses will start and control pulse output. After the initial pulse, the pulse output, either continuous or burst, will be output based on the internal pulse generator clock. If pulse-to-pulse synchronization is required over higher count pulse trains, use the trigger per pulse mode. For details, refer to External triggering in Section 11.

The internal bus trigger source is used for synchronizing multiple PMU/PGU cards for standard pulse using the legacy pulse functions (`pulse_vhigh`, `pulse_vlow`, `pulse_width`, and so on). This trigger source is used only by the Models 4220-PGU and 4225-PMU.

See also **`pulse_trig`**

Example The following function sets the trigger source to external – initial trigger only – rising:

```
pulse_trig_source(VPU1, 1)
```

pulse_vhigh Sets pulse V High value

Purpose This function sets the pulse V High level.

Format

```
int pulse_vhigh(INSTR_ID instr_id, long chan, double vhigh)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`vhigh`	Pulse V High value in volts (floating point number) Fast speed: -5 to +5 Slow speed: -20 to +20 Default: 0

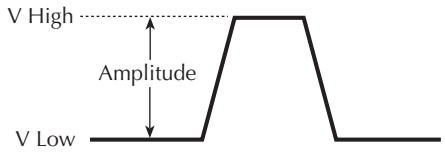
Remarks Pulse V High can be set independently for each pulse card channel. For lower voltages and faster transitions, it can be set from -5 V to 5 V for 50 Ω DUT load. For higher voltages and slower transitions, it can be set from -20 V to 20 V for 50 Ω DUT load. The `pulse_range` function sets the pulse voltage range.

NOTE *When using the* `pulse_range` *function, changing the source range after setting voltage levels (in any pulse mode) will result in voltage levels invalid for the new range setting. Therefore, use* `pulse_range` *before setting the voltage levels.*

As shown below, the pulse V High is typically set as the greater pulse voltage value. However, V High can be any valid voltage value. That means pulse V High can be less than V Low. When started, the pulse transitions from V Low to V High and then back to V Low. The voltage remains at V Low for the remainder of the pulse period.

The pulse V High setting takes effect immediately during continuous pulse output. Otherwise, the V High setting takes effect when the next trigger is initiated. The pulse_trig function is used to trigger continuous or burst output.

Figure 8-113
pulse_vhigh



See also **`pulse_vlow`**

Example The following function sets the pulse V High value for channel 1 of the PG2 to 2.5 V:

```
pulse_vhigh(VPU1, 1, 2.5)
```

pulse_vlow Sets pulse V Low value

CAUTION **The `pulse_vlow` and `pulse_dc_output` commands set the voltage value output by the pulse channel when it is turned on (using `pulse_output`). If the output is already enabled, these commands will change the voltage level immediately, even before the pulsing is started with a `pulse_trig` command.**

Purpose

This function sets the pulse V Low value.

Format

```
int pulse_vlow(INSTR_ID instr_id, long chan, double vlow)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`vlow`	Pulse V Low value in volts (floating point number) Fast speed: -5 to +5 Slow speed: -20 to +20 Default: 0

Remarks

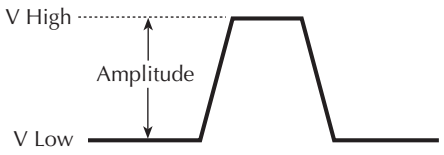
Pulse V Low can be set independently for each pulse card channel. For lower voltages and faster transitions, it can be set from -5 V to 5 V for 50 Ω DUT load. For higher voltages and slower transitions, it can be set from -20 V to 20 V for 50 Ω DUT load. The `pulse_range` function sets the pulse voltage range.

NOTE *When using the `pulse_range` function, changing the source range after setting voltage levels, in any pulse mode, will result in voltage levels invalid for the new range setting. Therefore, use `pulse_range` before setting the voltage levels.*

As shown below, the pulse V Low is typically set as the lesser pulse voltage value. However, V Low can be any valid voltage value. That means pulse V Low can be higher than V High. When started, the pulse transitions from V Low to V High and then back to V Low. The voltage remains at V Low for the remainder of the pulse period.

The pulse V Low setting takes effect immediately during continuous pulse output. Otherwise, the V Low setting takes effect when the next trigger is initiated. The pulse_trig function is used to trigger continuous or burst output.

Figure 8-114
pulse_vlow



See also **`pulse_vhigh`**

Example

The following function sets the pulse V Low value for channel 1 of the PG2 to 0.5 V:

```
pulse_vlow(VPU1, 1, 0.5)
```

pulse_width Sets pulse width

Purpose This function sets the pulse width for pulse output.

Format

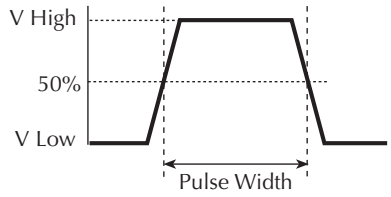
```
int pulse_width(INSTR_ID instr_id, long chan, double width)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`chan`	Channel number of the PG2: 1 or 2.
`width`	Pulse width in seconds Fast speed: 10 e-9 to (Period - 10 e-9) Slow speed: 250 e-9 to (Period - 10 e-9) Default: 500 e-9

Remarks Pulse width can be independently set for each PG2 channel. For fast speed, pulse width can be set from 10 ns to (Period - 10 ns). For slow speed, pulse width can be set from 250 ns to (Period - 10 ns). The `pulse_range` function is used to set pulse speed.

PG2 pulse width is based on the full width, half max method (FWHM). As shown below, the pulse width is measured at the median (50 percent amplitude) point from the rising edge of the pulse to the falling edge of the pulse.

Figure 8-115
pulse_width



The maximum pulse width that can be set depends on the presently set period for the pulse. For example, if the period is set for 500 ns, the maximum pulse width that can be set is 490 ns (500 ns - 10 ns = 490 ns).

The pulse width setting takes effect immediately during continuous pulse output. Otherwise, the width setting takes effect when the next trigger is initiated. The `pulse_trig` function is used to trigger continuous or burst output.

NOTE *Use the `pulse_source_timing` function to set the pulse width for the Models 4220-PGU and 4225-PMU.*

See also **`pulse_period`**

Example The following function sets the pulse width for channel 1 to 250 ns:

```
pulse_width(VPU1, 1, 250e-9)
```

seg_arb_define Defines a Segment ARB waveform

Purpose This function defines the parameters for a Segment ARB® waveform.

Format

```
int seg_arb_define(INSTR_ID inst_id, long ch, long nsegments,
double *startvals, double *stopvals, double *timevals, long
*triggervals, long *outputRelayVals);
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`ch`	The PG2 channel: 1 or 2.
`nsegments`	The number of values in each of the arrays (1024 maximum).
`startvals`	An array of start voltage values for each segment (in volts).
`stopvals`	An array of stop voltage values for each segment (in volts).
`timevals`	An array of time values for each segment (20 ns minimum).
`triggervals`	An array of trigger values: 0 (trigger low) or 1 (trigger high).
`outputRelayVals`	An array of values to control the high endurance output relay: 0 (open) or 1 (closed).

Remarks Each channel of the 4205-PG2 can be configured to output its own unique Segment Arb waveform. A Segment Arb waveform is made up of user-defined segments (up to 1024). Each segment can have a unique time interval, start value, stop value, output trigger level (TTL high or low) and output relay state (open or closed).

NOTE *The 4200-PG2 does not have high-endurance output relays (HEOR). When this function is used for an upgraded 4200-PG2, the values for the relays are ignored.*

See Segment ARB in Section 11 for details on this pulse mode. For triggering, refer to Pulse-measure synchronization and Pulse output synchronization.

To configure each channel to output a unique Segment Arb® waveform, refer to seg_arb_sequence.

The following arrays are required for the example Segment ARB® waveform shown in Figure 11-3:

Table 8-20
seg_arb_define

Start	Stop	Time	Trigger	Output relay
startvals(0) = 0.0	stopvals(0) = 1.0	timevals(0) = 50 e-9	triggervals(0) = 1	outputRelayVals(0) = 0
startvals(1) = 1.0	stopvals(1) = 1.0	timevals(1) = 100 e-9	triggervals(1) = 1	outputRelayVals(1) = 0
startvals(2) = 1.0	stopvals(2) = 1.5	timevals(2) = 20 e-9	triggervals(2) = 1	outputRelayVals(2) = 0
startvals(3) = 1.5	stopvals(3) = 1.5	timevals(3) = 150 e-9	triggervals(3) = 0	outputRelayVals(3) = 0
startvals(4) = 1.5	stopvals(4) = 0.0	timevals(4) = 50 e-9	triggervals(4) = 0	outputRelayVals(4) = 0
startvals(5) = 0.0	stopvals(5) = 0.0	timevals(5) = 500 e-9	triggervals(5) = 0	outputRelayVals(5) = 0
startvals(6) = 0.0	stopvals(6) = 0.0	timevals(6) = 130 e-9	triggervals(6) = 0	outputRelayVals(6) = 1

See also `arb_file`, `arb_array`, `seg_arb_file`

seg_arb_file Loads a waveform from a Segment ARB waveform file

Purpose

This function is used to load a waveform from an existing Segment ARB® waveform file.

Format

```
int seg_arb_file(INSTR_ID instr_id, long ch, char *fname)
```

`instr_id`	Instrument ID of the PG2: VPU1, VPU2, and so on.
`ch`	The PG2 channel: 1 or 2.
`fname`	The name of the waveform file.

Remarks

This function is used to load a waveform from an existing Segment ARB `.ksf` waveform file into the pulse generator card. A Segment ARB waveform can be loaded for each channel of the pulse generator card. Once loaded, use `pulse_output` to turn on the appropriate channel, and then use `pulse_trig` to select the trigger mode and start (or arm) pulse output.

When specifying the `fname`, include the full command path with the file name. Existing `.ksf` waveforms are typically saved in the `SarbFiles` folder at the following command path location:

`C:\S4200\kiuser\KPulse\SarbFiles`

A Segment ARB waveform can be created using KPulse, and then saved as a `.ksf` waveform file (refer to Section 13 for details).

A waveform in an existing `.ksf` file can be modified using a text editor.

NOTE *The 4200-PG2 does not have high-endurance output relays (HEOR). When this function is used for an upgraded 4200-PG2, the file will load, but the values for the relays will be ignored when the output is turned on.*

See also

`arb_array`, **`arb_file`**, **`seg_arb_define`**

Example

The following function loads a Segment ARB file named `sarb3.ksf` (saved in the `SarbFiles` folder) into the pulse generator card for channel 1:

```
seg_arb_file(VPU1, 1,
C:\\S4200\\kiuser\\KPulse\\SarbFiles\\sarb3.ksf)
```

LPT functions for the Models 4220-PGU and 4225-PMU

NOTE *The 4200-SCS has built-in project plan tests that use functions documented below. The PMU-DUT-Examples project in Section 16 provides simple examples for coding a PMU user test module (UTM).*

The following information explains the functions included in the Keithley Linear Parametric Test Library (LPTLib) for the 4220-PGU and 4225-PMU. To simplify the descriptions of these two pulse cards in this documentation, the model names are abbreviated as PGU (pulse-generator unit) and PMU (pulse-measure unit).

The difference between these two pulse cards is that the PGU functions only as a pulse generator, but the PMU has both pulse and measurement capabilities. See PGU (pulse only) and PMU (pulse and measure)* for a summary of the functions for the PGU and PMU.

The pulse commands for the 4220-PGU and 4225-PMU require pulse_exec to execute. In addition, they support external triggering, but do not support trigger input from external input signals or instruments. Older PG2 commands do support output and input trigger for 2-level pulse bursts and PG2-style Segment Arb output (see LPT functions for the Model 4205-PG2).

NOTE *The PGU and PMU support all the pulsing functions that the 4205-PG2 can perform. See PG2 (pulse only)* for a summary of the PG2 functions.*

The PGU and PMU support the following pulse modes:

- Standard pulse mode: For this two-level pulse mode, the user defines an amplitude and base level for the pulse output.
- Segment ARB pulse mode: For this multi-level pulse mode, you define a pulse waveform that consists of three or more line segments. Segment ARB pulse mode for the PGU and PMU also includes sequencing and sequence looping (see seg_arb_sequence and seg_arb_waveform for the PGU and PMU).
- Full arb pulse mode: For this multilevel pulse mode, the waveform consists of a number of user-defined points (see arb_array and arb_file).

Use the following instrument ID (identification) for LPT functions for the PGU and PMU:

- 4220-PGU: The instrument ID is VPU (VPU1, VPU2, and so on)
- 4225-PMU: The instrument ID is PMU (PMU1, PMU2, and so on)

The LPT functions for the PGU and PMU, which are described in detail on the following pages, include:

dev_abort (PGU, PMU)
PostDataDouble (PMU)*
PostDataDoubleBuffer (PMU)
pulse_chan_status (PMU)
pulse_conncomp (PMU)
pulse_exec (PGU, PMU)
pulse_exec_status (PGU, PMU)
pulse_fetch (PMU)
pulse_chan_status (PMU)
pulse_limits (PMU)
pulse_meas_sm (PMU)
pulse_meas_wfm (PMU)
pulse_meas_timing (PMU)
pulse_measrt (PMU)
pulse_ranges (PGU, PMU)
pulse_remove (PGU, PMU)
pulse_sample_rate (PMU)
pulse_source_timing (PGU, PMU)
pulse_step_linear (PGU, PMU)
pulse_sweep_linear (PGU, PMU)
pulse_train (PGU, PMU)
rpm_config (PMU)
seg_arb_sequence (PGU, PMU)
seg_arb_waveform (PGU, PMU)
setmode (PMU)

* The PostDataDouble function is also used with the SMU and CVU (see Enabling real time plotting for UTMs).

dev_abort Aborts a pulse_exec test

Purpose This function is used to programmatically end a test from within the user module (abort a test) that was started with the `pulse_exec` command.

Pulsers 4220-PGU (VPU), 4225-PMU

Pulse mode Standard and Segment ARB

Format

```
int dev_abort (NULL);
```

Remarks Use this function to abort a PGU or PMU pulse test from within a user module. This command is useful during a longer `pulse_exec` test. Note that `pulse_exec` is nonblocking, so it is possible to fetch data during a longer test. An example of this is a long stress/measure test, using `seg_arb_sequence` and `seg_arb_waveform`, that

evaluates degradation data intra-test. Evaluating this data from within the user module may determine that a test should end. For example, if the degradation is greater than ten percent (> 10%), then end the test (saves test time). This command is necessary to properly abort any test that uses `pulse_exec`, from within the user module.

Example The following code example illustrate placement of this command in a code a fragment. Note that after the `dev_abort` command is sent it is still necessary to use `pulse_exec_status` to poll and wait for the test to be ended.

```
// Place code to configure the PMU test here
//
// Start the test (for seg-arb testing, or for standard pulsing
// with no ranging, LLEC, or I/V/P threshold detection)
status = pulse_exec(PULSE_MODE_SIMPLE);
         if ( status )
         {
              // Minimal error handling, release memory used to
              // fetch results and stop test
              Free_Used_Arrays();
              return status;
         }
// loop to fetch data, while waiting for test complete
abort_sent = 0;
while(pulse_exec_status(&elapsedt) == 1)
      {
           // Code to fetch and evaluate data here
           if (abort_sent == 0)
           {
                // Code to fetch PMU data
                // Code to evaluate data
                // Code to determine if an abort is required
           }

           // If the test must be aborted, send dev_abort
           if (abort_required && abort_sent == 0)
           {
                dev_abort(NULL);
                abort_sent = 1;
           }
           Sleep(100);
      }
```

PostDataDouble Post buffer data into KITE Sheet tab
PostDataDoubleBuffer Post buffer data into KITE Sheet tab (for large data sets)

Purpose

`PostDataDouble`	Post data retrieved from the buffer into the KITE Sheet tab.
`PostDataDoubleBuffer`	Post PMU data retrieved from the buffer into the KITE Sheet tab (large data sets).

Pulsers 4225-PMU

Pulse mode Standard and Segment ARB

Formats

PostDataDouble

```
int PostDataDouble(char *ColName, double *array);
```

`*ColName`	Column name for the data array into the KITE Sheet tab.
`array`	An array of data values for the KITE Sheet tab

PostDataDoubleBuffer

```
int PostDataDoubleBuffer(char *ColName, double *array, int
length);
```

`*ColName`	Column name for the data array in the KITE Sheet tab.
`array`	An array of data values for the KITE Sheet tab.
`length`	Number of data points (up to 65,535) to post into the KITE Sheet tab.

Remarks The `PostDataDouble` and `PostDataDoubleBuffer` functions are used to post double precision, floating point PMU data into the KITE Sheet tab. Up to 65,535 points (rows) of data can be posted into the Sheet tab.

NOTE *The PostDataDouble function can also be used with the SMU and CVU (see Enabling real time plotting for UTMs).*

NOTE *The PostDataDoubleBuffer function is not compatible using KXCI to call user libraries remotely (see Calling KULT user libraries remotely in Section 9. Use PostDataDouble for user routines (UTMs) that will be called via KXCI.*

Either of these two functions can be used to post PMU data into the Sheet tab. However, the `PostDataDoubleBuffer` function should be used to post large data sets typically generated by PMU waveform measurements.

The `PostDataDouble` function uses two parameters. The `array` parameter is the array of data values to be posted into the Sheet tab. The **ColName* parameter is used to name the column for the data array into the Sheet tab.

The `PostDataDoubleBuffer` function is similar but uses a third parameter. The `length` parameter specifies the number of data points to post into the Sheet tab.

The following sequence summarizes the process to post data into the Sheet tab:

- Run a test.
- Use the pulse_fetch function to retrieve the data from the buffer. If desired, the retrieved data can be analyzed and/or manipulated.
- Use the `PostDataDouble` or `PostDataDoubleBuffer` function to post data into the KITE Sheet tab.

When using the `pulse_fetch` function, you can wait until the test is finished before retrieving data or you can retrieve blocks of data while the test is running, which is useful for a test that takes a long time. Instead of waiting for the entire test to finish, you can retrieve blocks of data at prescribed intervals. For details, see Data retrieval options for pulse_fetch.

NOTE *If you do not need to analyze and/or manipulate the test data before posting it into the Sheet tab in KITE, you can use the pulse_measrt function. The* `pulse_measrt` *function retrieves all the test data in pseudo real-time and automatically posts it into the KITE Sheet tab.*

Examples

Two examples are provided. One uses the `PostDataDouble` function to post spot mean measurement data into the KITE Sheet tab. The second example uses the `PostDataDoubleBuffer` function to post waveform measurement data into the Sheet tab.

PostDataDouble example

This example assumes that a PMU spot mean test is configured to perform 100 (or more) voltage and current measurements for pulse high and low. Use the pulse_meas_sm function to configure the spot mean test.

The code below performs the following tasks:

- Starts the configured test.
- Uses a while loop to allow the spot mean test to finish.
- Retrieves voltage and current readings (100 data points) from the buffer.
- Separates the voltage and current readings for high (amplitude) and low (base).
- Posts the high measurement data into the KITE Sheet tab. Low measurement data is not posted into the Sheet tab.

```
// Code to configure the PMU test here

// Start the test (no analysis)
pulse_exec(0);
// While loop (continues while test is still running), with delay
// (30 ms)
while(pulse_exec_status(&elapsedt) == 1)
{
   Sleep(30);
}
// Retrieve V and I data (no timestamp or status)
status = pulse_fetch(PMU1, 1, 0, 100, Vmeas, Imeas, NULL, NULL);
// Separate V & I measurements for high (amplitude) and
// low (base)
for (i = 0; i<100; i++)
{
   VmeasHi_sheet[i] = Vmeas[2*i];
   ImeasHi_sheet[i] = Imeas[2*i];
   VmeasLo_sheet[i] = Vmeas[2*i+1];
   ImeasLo_sheet[i] = Imeas[2*i+1];
   PostDataDouble("DrainVmeas", VmeasHi_sheet[i]);
   PostDataDouble("DrainImeas", ImeasHi_sheet[i]);
}
```

PostDataDoubleBuffer example

This example assumes that a PMU waveform test is configured to perform 20,000 (or more) voltage and current measurements. Use the pulse_meas_wfm function to configure the waveform test.

The code below performs the following tasks:

- Starts the configured test.
- Uses a while loop to allow the waveform test to finish.
- Retrieves voltage, current, and timestamp readings (20,000 data points) from the buffer.
- Separates the voltage, current, and timestamp readings.
- Posts the measurement data into the KITE Sheet tab.

```
// Code to configure the PMU test here
// Start the test (no analysis)
status = pulse_exec(0);
// While loop (continues while test is still running), with
// delay (30 ms)
while (pulse_exec_status(&elapsedt) == 1)
{
   Sleep(30);
}
// Retrieve V, I, and timestamp data (no status)
status = pulse_fetch(PMU1, 1, 0, 20e3, Vmeas, Imeas, Tstamp, NULL);
// Separate V, I, and timestamp measurements
for (i = 0; i<20e3; i++)
{
   Vmeas_sheet[i] = Vmeas[2*i];
   Imeas_sheet[i] = Imeas[2*i];
   Tstamp_sheet[i] = Tstamp[2*i];
}
PostDataDoubleBuffer("DrainVmeas", Vmeas_sheet, 20e3);
PostDataDoubleBuffer("DrainImeas", Imeas_sheet, 20e3);
PostDataDoubleBuffer("Timestamp", Tstamp_sheet, 20e3);
```

PostDataInt Post an integer-type data point into KITE Sheet tab

Purpose

This command posts an integer-type data point from memory to the KITE Sheet tab in the user test module and plots it on the graph.

Format

```
PostDataInt(char *variableName, int *variableValue);
```

variableName	The variable name.
variableValue	The value of the variable to be transferred.

Remarks

The first parameter is the variable name, defined as `char *`. For example, if the output variable name is `DrainI`, then `DrainI` (with quotes) is the first parameter.

The second parameter is the value of the variable to be transferred. For example, if `DrainI[10]` is transferred, then you call `PostDataInt("DrainI", DrainI[10])`.

PostDataString Transfers a string to the KITE Sheet tab

Purpose

This command transfers a string from memory to the KITE Sheet tab in the user test module and plots it on the graph.

Format

```
PostDataString(char *variableName, int *variableValue);
```

variableName	The variable name.
variableValue	The value of the variable to be transferred.

Remarks

The first parameter is the variable name, defined as `char *`. For example, if the output variable name is `DrainI`, then `DrainI` (with quotes) is the first parameter.

The second parameter is the value of the variable to be transferred. For example, if `DrainI[10]` is transferred, then you call `PostDataInt("DrainI", DrainI[10])`.

pulse_chan_status Returns the number of readings in the data buffer

Purpose

This function is used to determine how many readings are stored in the data buffer.

Pulsers

4225-PMU

Pulse mode

Standard and Segment ARB

Format

```
int pulse_chan_status(INSTR_ID instr_id, int chan, int
*buffersize);
```

instr_id	Instrument ID: PMU1, PMU2, and so on.
chan	PMU channel: 1 or 2.
*buffersize	User-defined name for the returned value.

Remarks

Use this function to return the number of readings presently stored in the data buffer. This function can be called while a sweep is in progress or after it is completed.

For a short sweep (test time in seconds to a minute or more), this function is typically called after the sweep is completed to determine the total number of readings stored in the buffer. For a long test, you can use this function to track the progress of the test. A long test is typically Segment ARB with test time in minutes, hours, or days).

Example This function returns the number of readings stored in the buffer for channel 1:

```
pulse_chan_status(PMU1, 1, buffersize);
```

pulse_conncomp Control connection compensation

Purpose This function controls (enables or disables) connection compensation.

Pulsers Model 4225-PMU

Pulse mode Standard and Segment ARB

Format

```
int pulse_conncomp(INSTR_ID instr_id, int chan, int type, int
index);
```

`instr_id`	Instrument ID: PMU1, PMU2, and so on.
`chan`	PMU channel: 1 or 2.
`type`	Type of compensation to enable: 1: Short 2: Delay
`index`	0: Disable (for both types) Short: 1: Chx-Short-Res-2m-PMU 2: Chx-Short-Res-2m-RPM 3: Chx-Short-Res-2m-Custom Delay: 1: Chx-Delay-2m-PMU 2: Chx-Delay-2m-RPM 3: Chx-Delay-2m-Custom Where: x is the channel number (1 or 2)

Remarks Errors caused by the connections and cable length between the Model 4225-PMU and the device under test (DUT) can be corrected by using connection compensation. When connection compensation is enabled, each DUT measurement factors in either the default or measured (custom) compensation values.

Use this function to control connection compensation. There are two types of compensation using this function: Short and delay. Short compensation corrects for the measured resistance of the cabling and connections; delay compensation measures and corrects for cable delay (which is the time it takes a signal to transit the cable).

You have the option to use either default connection compensation values (PMU or RPM) or custom connection compensation values. The default values provide compensation for simple connection setups that use the supplied cables. The custom connection compensated values are generated when connection compensation is performed. The custom values provide optimum compensation.

`index` parameter values:

- 0: Disables all connection compensation.
- 1: Selects the default connection compensation values for a setup that uses the PMU only.
- 2: Selects the default connection compensation values for a setup that uses the PMU and the RPM.
- 3: Selects the custom connection compensation values.

Custom connection compensation is a two-part process:

1. Perform connection compensation from the KITE interface (see Performing connection compensation in Section 16). Connection compensation data is generated for short and delay conditions. The compensation values are stored in tables.
2. Use this function (`pulse_conncomp`) to select (enable) the custom connection compensation values.

When a test is run, each measurement will factor in the enabled compensation values. If connection compensation is disabled, the compensation values will not be used by the test.

Example Assuming connection compensation was performed from the KITE interface, this function selects (enables) short connection compensation using the custom correction values:

```
pulse_conncomp(PMU1, 1, 1, 3);
```

pulse_exec Starts test execution

Purpose This function is used to validate the test configuration and start test execution.

Pulsers 4220-PGU (VPU) and 4225-PMU

Pulse mode Standard and Segment ARB

Format

```
int pulse_exec(long mode);
```

`mode`

`PULSE_MODE_SIMPLE` or 0: No analysis performed during testing; no ranging, no load line effect compensation, and no threshold checking.

`PULSE_MODE_ADVANCED` or 1: Enables the analytical sweep engine and incorporates the use of any combination of the various options for the standard (2-level) pulse mode.

Remarks Use this function to validate the test configuration, select the simple or advanced mode, and execute the test. If there are any problems with the test configuration, the validation will stop and the test will not be executed.

The `pulse_exec` function is nonblocking, which means that if this function is called to execute the test, the program will continue and not wait for the test to finish. Therefore, after calling pulse_exec, the pulse_exec_status command must be called in a while loop to ensure the test is complete before fetching data or exiting the UTM.

When using pulse_fetch to retrieve data, you need to pause the program to allow time for the buffer to fill. You can use the sleep function to pause for a specified period of time, or you can use the pulse_exec_status function in a `while` loop to wait until the test is completed.

Noe that the newer PMU and PGU commands require the use of the pulse_exec command. A trigger input to start a test is not available.

There are two functions that will affect a pulse test while it is running:

- The pulse_remove function removes a PMU channel from the test.
- The dev_abort function aborts the test.

NOTE *The "Internal Trigger Bus" trigger source (see pulse_trig_source function) is used only by the 4220-PGU and 4225-PMU for triggering. The pulse_exec command automatically uses the internal trigger bus.*

CAUTION **Do not exit the user module while the test is still running, or incorrect readings or device damage may result.**

Example Program fragment 1 shows how to use `pulse_exec` which validates the test configuration, sets the execution type to simple two-level pulse operation (no analysis), and executes the test.

pulse_exec_status Checks the run status of the test

Purpose This function is used to determine if a test is running or completed.

Pulsers Models 4220-PGU (VPU) and 4225-PMU

Pulse mode Standard and Segment ARB

Format

```
int pulse_exec_status(double *elapsedt);
```

`*elapsedt` Name of the user-defined pointer for elapsed time.

Remarks This function is required to determine when a test is complete or what is occurring during a test. The return value indicates whether the test is still running (`PMU_TEST_STATUS_RUNNING` or `1`) or completed (`PMU_TEST_STATUS_IDLE` or `0`). The primary use of this command is to ensure that the test is completed before fetching PMU data or ending the test. See the code example link below. The parameter for this function is a pointer to indicate elapsed time.

The elapsed time is the KITE test time, not the PMU or VPU card test time. For short test times, the returned elapsed time will be longer than the actual time required on-card.

This function is typically used in a `while` loop to allow the test to finish before retrieving the data using the pulse_fetch function (see Program fragment 1).

It is the responsibility of the UTM programmer to ensure that the pulse test is complete before exiting the UTM. If the UTM program ends before the test is complete, KITE will respond with two messages. These messages are displayed in the KITE messages window. Five seconds after the UTM ends prematurely (before the pulse test is finished), the message "UTMname ended before the test was complete. Waiting for test to finish (max wait = 5 minutes).". KITE will continue to wait for the UTM to finish, interrupting further test execution. After the default of five minutes, the UTM is terminated and the following message is displayed, "UTMname did not finish before the maximum wait period. UTM aborted." After this five minute wait, KITE will release control to the GUI, or the next test in the project (if using repeat executing or looping).

See also **`pulse_exec`**

Example Program fragment 1 shows how to use `pulse_exec_status` in a `while` loop.

pulse_fetch Retrieves enabled test data

Purpose This function retrieves enabled test data and temporarily stores it in the data buffer.

Pulsers Model 4225-PMU

Pulse mode Standard and Segment ARB

Format

```
int pulse_fetch(INSTR_ID instr_id, int chan, int StartIndex, int
StopIndex, double *Vmeas, double *Imeas, double *Timestamp,
unsigned long *Status);
```

`instr_id`	Instrument ID: PMU1, PMU2, and so on.
`chan`	PMU channel: 1 or 2.
`StartIndex`	Start index point for data (within the overall set of data).
`StopIndex`	Final index point to be retrieved.
`*Vmeas`	Name of the user-defined array for retrieved voltage measure readings. This is a single-dimension array.
`*Imeas`	Name of the user-defined array for retrieved current measure readings. This is a single-dimension array.
`*Timestamp`	Name of the user-defined array for retrieved time stamps. This is a single-dimension array.
`*Status`	Name of the user-defined array for retrieved status for the channel.

Remarks Use this function to retrieve a block of newly generated test data in pseudo real-time and temporarily store it in the data buffer. The stored data can then be analyzed and manipulated as needed before posting it to the Sheet tab in KITE.

Typically, this function is used with the `pulse_exec_status` function to allow the test to finish before retrieving the data. For details, see Wait until the test is completed before retrieving data.

The block of data to be retrieved is set by the `StartIndex` and `StopIndex` parameters. The start index parameter specifies the first index number in the buffer, and the stop index parameter specifies the final index number. For example, assume there are 1000 data test points for a test, and you want to retrieve the first 50 data points. The start index value is set to zero (0) and the stop index is set to 49.

The `*Vmeas`, `*Imeas`, `*Timestamp`, and `*Status` parameters are array names defined by the user. If you do not want to retrieve the time stamp or status, `NULL` can be passed as a valid parameter for this field.

NOTE *The return of all readings must be enabled by the pulse_meas_sm function. If disabled, the arrays will not be retrieved.*

For spot mean measurements, amplitude and base level readings are returned in the same array buffer area and must be separated (or parsed) after the measurement cycle is complete. See pulse_meas_sm function for details on spot mean measurements. Note that number of measurements returned is determined by the spot means enabled in pulse_meas_sm. With both amplitude and base measurements enabled, there will be two voltage and two current readings for each pulse (with spot mean discrete) or

each pulse burst (with spot mean average). Voltage and current readings are returned in individual arrays: Vmeas, Imeas. When both amplitude and base readings are enabled, the readings are alternated. For example, the Vmeas array: Vampl_1, Vbase_1, Vampl_2, Vbase_2, Vampl_3, Vbase_3, etc. To plot the amplitude values, separate the amplitude and base measurements into individual arrays before using PostDataDouble to post the measurements to the sheet.

The time stamps pertain to either per spot mean reading or per sample. Status is returned as a 32-bit word. The status code bit map is shown in Table 8-21.

Table 8-21
Status-code bit map for pulse_fetch

Bit	Summary or description	Value (bit pattern)
31	Reserved	Reserved bit for future use
30	Sweep skipped	0 = Not skipped 1 = Skipped
29	Load line effect compensation (LLEC) enabled (only valid when LLEC is enabled)	0 = Failed 1 = Successful
28	LLEC status	0 = Disabled 1 = Enabled
27 - 24	RPM mode settings	0 (0000) = No RPM 1 (0001) = RPM 2 (0010) = Bypass; PMU 3 (0011) = Bypass; SMU 4 (0100) = Bypass; CVU All other values (bit patterns) reserved
23 - 20	Reserved	Reserved bits for future use
19 - 16	Measurement type	1 (0001) = Spot mean 2 (0010) = Waveform All other values (bit patterns) reserved
15 -12	Current threshold, voltage threshold, power threshold, and source compliance	0 (0000) = None 1 (0001) = Source compliance 2 (0010) = Current threshold reached or surpassed 4 (0100) = Voltage threshold reached or surpassed 8 (1000) = Power threshold reached or surpassed
11 - 10	Current measure overflow	0 (00) = No overflow 1 (01) = Negative overflow 2 (10) = Positive overflow
9 - 8	Voltage measure overflow	0 (00) = No overflow 1 (01) = Negative overflow 2 (10) = Positive overflow
7 - 4	Current measure range	0 (0000) = 100 nA (RPM only) 1 (0001) = 1 µA (RPM only) 2 (0010) = 10 µA (RPM only) 3 (0011) = 100 µA 4 (0100) = 1 mA (RPM only) 5 (0101) = 10 mA 6 (0110) = 200 mA 7 (0111) = 800 mA All other values (bit patterns) reserved
3 - 2	Voltage measure range	0 (00) = 10 V 1 (01) = 40 V
1 - 0	Channel number	1 (01) = Ch1 2 (10) = Ch2 Value 0 (00) not used

NOTE *If you do not need to analyze and/or manipulate the test data before posting it to the Sheet tab in KITE, you can use the pulse_measrt function. The* `pulse_measrt` *function retrieves all the test data in pseudo real-time and automatically posts it into the KITE Sheet tab.*

Data retrieval options for pulse_fetch

There are two options to retrieve data: 1) Wait until the test is completed, or 2) retrieve blocks of data while the test is running.

Because pulse_exec is a non-blocking function, the running user test module (UTM) will continue after it is called to start the test. This means that the program will not automatically pause to allow the pulse-measure test to finish.

CAUTION **The programmer must ensure that the test program does not finish or return to KITE before the test is complete, or erroneous results and damage to test devices may occur.**

If `pulse_fetch` is inadvertently called before the test is completed, the data buffer may not fill with all the requested readings. Array entries are designated as zero for test data that is not yet available.

Wait until the test is completed before retrieving data

An effective method to pause the program is to monitor the status of the test by using a `while` loop to check the returned value of pulse_exec_status. When the test is completed, the program drops out of the loop and calls `pulse_fetch` to retrieve all the test data. The following program fragment shows how to use a `while` loop.

Program fragment 1

```
// Code to configure the PMU test here

// Start the test (no analysis)
pulse_exec(0);

// while loop and short delay (10 ms)

while (pulse_exec_status(&elapsedt) == 1)

{

   Sleep(10);

}

// Retrieve all data
status = pulse_fetch(PMU1, 1, 0, 49, Drain_Vmeas, Drain_Imeas,
NULL, NULL);

// Code for data handling here
```

After all the data is retrieved, it can be analyzed, manipulated and then posted into the KITE Sheet tab. Use the PostDataDouble or PostDataDoubleBuffer function to post the data.

Retrieve blocks of data while test is running

An advantage of the `pulse_exec` function being non-blocking is that it allows you to retrieve test data before the test is completed, which is useful for a test that takes a long time. Instead of waiting for the entire test to finish, you can retrieve blocks of data at prescribed intervals. The interval can be controlled by using the `sleep` function as shown in the following program fragment.

Program fragment 2

```
// Code to initialize the data arrays
for i (i = 0; i < array_size; i++)
{
   Drain_Vmeas = 0.0;
   Drain_Imeas = 0.0;
}


// Code to configure the PMU test here
// Start the test and pause for 20 seconds
pulse_exec(0);
Sleep(20000);
// Retrieve a block of test data:
pulse_fetch(PMU1, 1, 0, 10e3, Drain_Vmeas, Drain_Imeas, 1,
NULL);
// Code for data handling here
```

After retrieving a block of data, loop back to the sleep function to allow the next block of data to become available before fetching it. Repeat this loop until all the data is retrieved.

The `pulse_fetch` command will return all data available at the time of the call. The remaining array space will not be modified. To determine how much data was retrieved, it is recommended to initialize the arrays. **Program fragment 2** initializes the results arrays to 0.0, but other values may be used. After the retrieving the data, search the array for the first entry with this initialized value.

Retrieved blocks of data can be analyzed and manipulated while the test is still running. After data handling is completed, use the PostDataDoubleBuffer function to post the data to the KITE Sheet tab.

Example

This function retrieves 50 points of data from the buffer:

```
pulse_fetch(PMU1, 1, 0, 49, Drain_Vmeas, Drain_Imeas, T_Stamp,
NULL);
```

Where:

`Instr_id` = `PMU1`
`chan` = `1` (channel 1)
`StartIndex` = `0`
`StopIndex` = `49`
`Vmeas` = `Drain_Vmeas` (name of array)
`Imeas` = `Drain_Imeas` (name of array)
`Timestamp` = `T_Stamp` (name of array)
`Status` = `NULL` (not retrieved)

pulse_limits Sets voltage, current, and power thresholds

Purpose Sets measured voltage and current thresholds at the DUT. Also sets the power threshold for each channel.

Pulsers Model 4225-PMU

Pulse mode Standard

Format

```
int pulse_limits(INSTR_ID instr_id, int chan, double V_Limit,
double I_Limit, double Power_Limit);
```

`instr_id`	Instrument ID: PMU1, PMU2, and so on.
`chan`	Pulse generator channel: 1 or 2.
`V_Limit`	Measured voltage (V) threshold at the DUT.
`I_Limit`	Measured current (A) threshold at the DUT.
`Power_Limit`	Power (W) threshold for the channel (Power = Vmeas x Imeas).

Remarks This feature is different from a SMU compliance setting, in that threshold checking is performed after each burst of pulses, using the spot mean values to compare to the specified thresholds. The thresholds are checked against all enabled measurements for the channel. If a threshold is reached or exceeded, the present sweep is stopped and testing continues with any subsequent sweeps.

This feature doesn't prevent the set thresholds from being reached or exceeded. After detecting a threshold breach, it simply aborts the sweep.

Maximum power for each PMU source range:

High-speed voltage source (10 V) range: Maximum power = 5 V x 0.1 A = 0.5 W
High-voltage source (40 V) range: Maximum power = 20 V x 0.4 A = 8 W

Example This function sets thresholds for channel 1 of the PMU:

```
pulse_limits(PMU1, 1, 42, 1, 10);
```

`Instr_id` = PMU1
`chan` = 1 (channel 1)
`V_Limit` = 42 V
`I_Limit` = 1 A
`Power_Limit` = 10 W

pulse_meas_sm Configures spot mean measurements
pulse_meas_wfm Configures waveform measurements
pulse_meas_timing Sets the measurement windows

Purpose

`pulse_meas_sm`	This function configures spot mean measurements.
`pulse_meas_wfm`	This function configures waveform measurements.
`pulse_meas_timing`	This function sets the measurement windows.

Pulsers Model 4225-PMU

Pulse mode Standard

Formats

pulse_meas_sm

```
int pulse_meas_sm(INSTR_ID instr_id, int chan, Int AcquireType,
int AcquireMeasVAmpl, int AcquireMeasVBase, int
AcquireMeasIAmpl, int AquireMeasIBase, int AquireTimeStamp, int
LLEComp);
```

`instr_id`	Instrument ID: PMU1, PMU2, and so on.
`chan`	PMU channel: 1 or 2.
`AcquireType`	Acquisition type: 0 = Discrete 1 = Average
`AcquireMeasVAmpl`	Enable (1) or disable (0) return of amplitude voltage measurements.
`AcquireMeasVBase`	Enable (1) or disable (0) return of base level voltage measurements.
`AcquireMeasIAmpl`	Enable (1) or disable (0) return of amplitude current measurements.
`AcquireMeasIBase`	Enable (1) or disable (0) return of base current level measurements.
`AcquireTimeStamp`	Enable (1) or disable (0) return of time stamp readings.
`LLEComp`	Load line effect compensation (LLEC): 0 = All LLEC disabled 1 = Voltage LLEC on for pulse amplitude only LLEC is only performed for standard pulse IV testing using PMU measure ranges. It is not performed when using Model 4225-RPM measure ranges. The active RPM circuitry provides its own analog LLEC (assuming a short cable from the RPM to the DUT).

pulse_meas_wfm

```
int pulse_meas_wfm(INSTR_ID instr_id, int chan, int AcquireType,
int AcquireMeasV, int AcquireMeasI, int AquireTimeStamp, int
LLEComp);
```

`instr_id`	Instrument ID: PMU1, PMU2, and so on.
`chan`	PMU channel: 1 or 2.
`AcquireType`	Acquisition type: 0 = Discrete 1 = Average
`AcquireMeasV`	Enable (1) or disable (0) return of voltage measurements.
`AcquireMeasI`	Enable (1) or disable (0) return of current measurements.
`AcquireTimeStamp`	Enable (1) or disable (0) return of time stamp readings. Time stamp must be enabled in order to measure waveforms.
`LLEComp`	Load line effect compensation (LLEC): 0 = LLEC disabled 1 = LLEC enabled LLEC is only performed for standard pulse IV testing using PMU measure ranges. It is not performed when using 4225-RPM measure ranges. The active RPM circuitry

provides its own analog LLEC (assuming a short cable from the RPM to the DUT).

pulse_meas_timing

```
int pulse_meas_timing(INSTR_ID instr_id, int Chan, double,
StartPercent, double StopPercent, int NumPulses);
```

`instr_id`	Instrument ID: PMU1, PMU2, and so on.
`chan`	PMU channel: 1 or 2
`StartPercent`	Spot mean: Start location for spot mean measurements, specified as a percentage of the widths for the amplitude and base level (see Figure 8-116). Waveform: Pre-data for the amplitude, specified as a percentage of the amplitude pulse duration (see Figure 8-120).
`StopPercent`	Spot mean: Stop location of the spot mean measurements, specified as a percentage of the widths for the amplitude and base level (see Figure 8-116). Waveform: Post-data for the amplitude, specified as a percentage of the amplitude pulse duration (see Figure 8-120).
`NumPulses`	Number of pulses to output and measure (1 to 10,000).

NOTE *Use the pulse_sample_rate function to set the sampling rate for pulse measurements.*

Remarks

There are four types of pulse measurements that are performed by the Model 4225-PMU: Spot mean discrete, spot mean average, waveform discrete, and waveform average. Use the following pulse generator functions to configure pulse measurements:

- Use the `pulse_meas_sm` function to configure spot mean measurements; select the data acquisition type, set the readings to be returned, enable or disable time stamp, and set LLEC. See Spot mean measurements for details.
- Use the `pulse_meas_wfm` function to configure waveform measurements; select the data acquisition type, set the readings to be returned, enable or disable time stamp, and set LLEC. See Waveform measurements for details.
- Use the `pulse_meas_timing` function to set measurement timing. For spot mean measurements, portions of the amplitude and base levels are specified for sampling. For pre-data and post-data waveform measurements, a percentage of the entire pulse duration is specified. See Measurement timing for details on pulse measurement timing.

NOTE *Before calling the* `pulse_meas_timing` *function, use the* `pulse_meas_sm` *or* `pulse_meas_wfm` *function to configure the measurement type.*

Measurement types

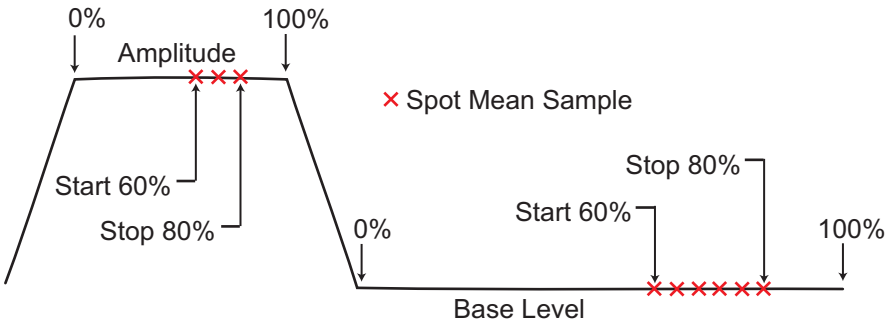
There are two types of pulse measurements: Spot mean and waveform. The pulse_meas_sm function is used to configure Spot mean measurements and the pulse_meas_wfm function is used to configure Waveform measurements.

Spot mean measurements

Spot mean measurements sample a portion of the amplitude and a portion of the base level. The measured samples are then averaged to yield a single voltage and current reading for the amplitude and base low levels. The `NumPulses` (number of pulses) parameter is used to specify the number of pulses to be output and sampled. To return individual spot mean for each pulse, see Spot mean discrete readings. To have a single spot mean for all NumPulses, see Spot mean average readings. With both amplitude and base spot means enabled, each pulse will have 2 voltage measurements and two current measurements. See pulse_fetch for details.

In Figure 8-116, three measured samples are taken on the amplitude and six samples are taken on the base level. The start and stop percentage values shown in Figure 8-116 indicate the portions of the pulse that are sampled. See Spot mean measurement timing for details on measurement timing.

Figure 8-116
Spot mean measurements example

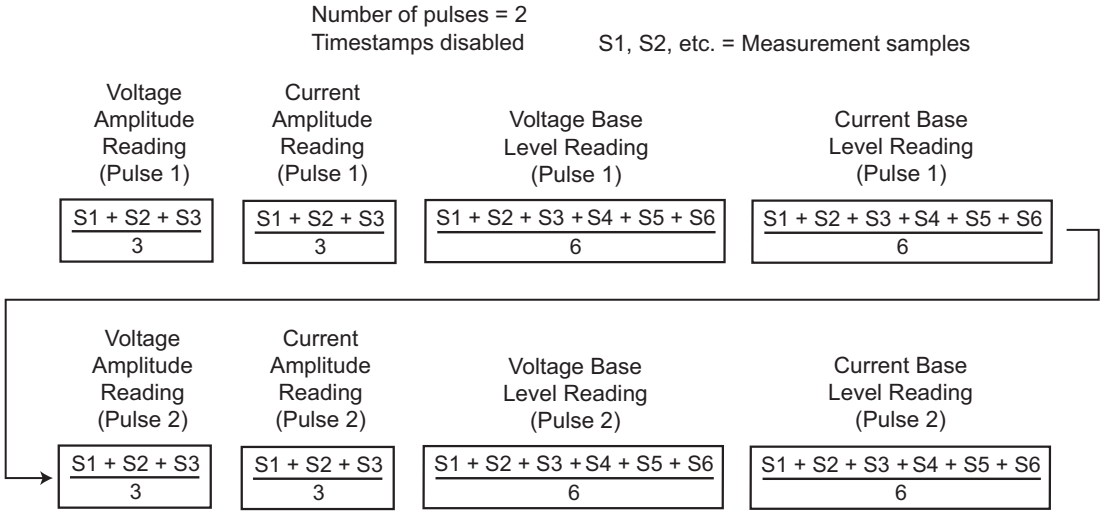


There are two data acquisition types of spot mean measurements; Spot mean discrete readings and Spot mean average readings.

Spot mean discrete readings

The averaged voltage and/or current readings for every sampled pulse period are returned in a single data set. Figure 8-117 shows how spot mean discrete readings are returned as a data set for two pulse periods. With all voltage and current readings enabled, four readings are returned for each pulse. The measured samples are averaged to yield the mean average readings. When time stamps are enabled, a time stamp is included in the data set after each mean reading.

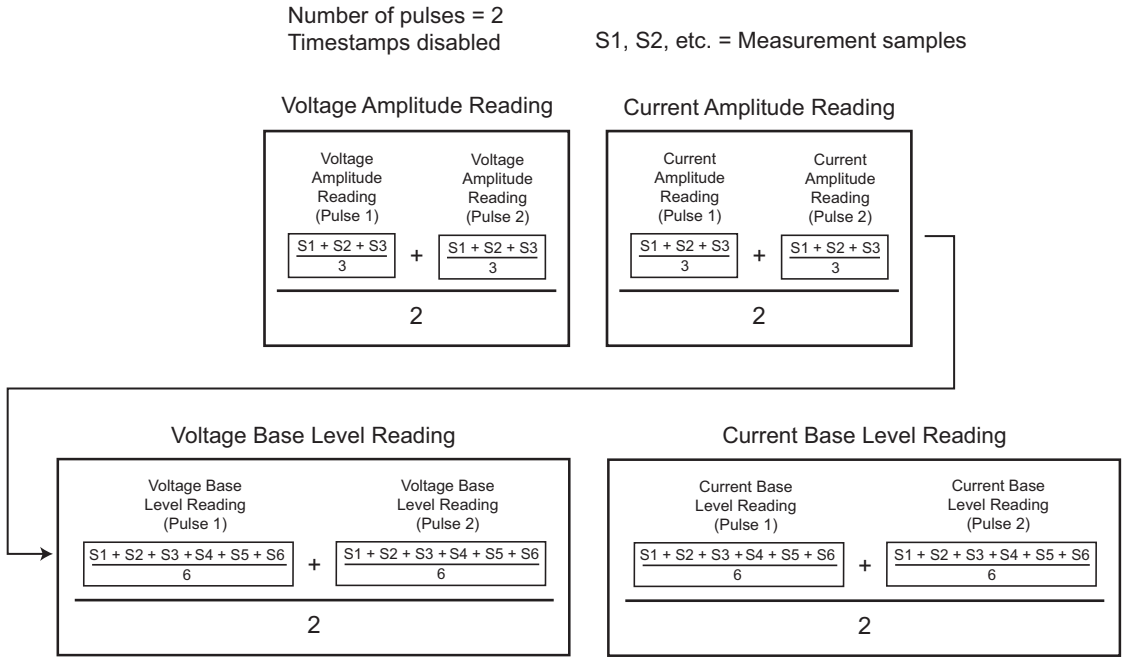
Figure 8-117
Returned data set for spot mean discrete readings



Spot mean average readings

For this data acquisition type, each returned reading is a mean-of-the-means. Spot mean average averages the mean readings for all the pulses in the burst. In Figure 8-117, each mean reading for Pulse 1 is averaged with each corresponding mean reading for Pulse 2 to yield the mean-of-the-means readings shown in Figure 8-118. With all voltage and current readings enabled, four readings are returned for the burst. When time stamps are enabled, a time stamp is included in the data set after each mean-of-the-means reading.

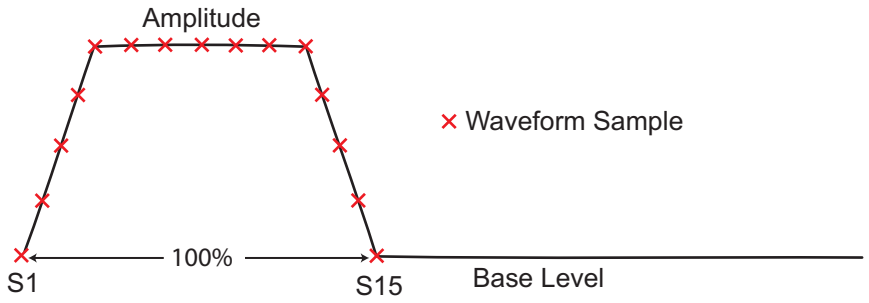
Figure 8-118
Returned data set for spot mean average readings



Waveform measurements

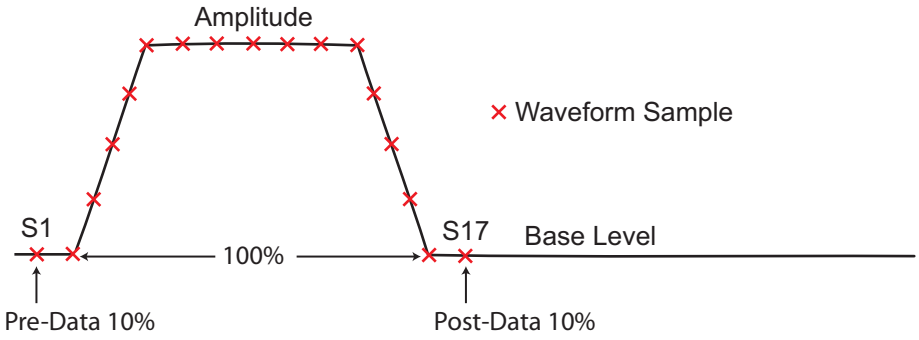
Waveform measurement readings sample the entire pulse. Sampling is performed on the rise time, top width, and fall time portions of the pulse. In Figure 8-119, 15 samples are performed on the pulse waveform. Enabled voltage and current readings and time stamps are returned for every sample taken on the pulse. The `NumPulses` (number of pulses) parameter is used to specify the number of pulses to be output and sampled.

Figure 8-119
Waveform measurements



A waveform measurement can include pre-data and post-data. Pre-data is extra data taken before the rise time of the pulse; post-data is extra data taken after the fall time (see Figure 8-120). See Waveform measurement timing for details on measurement timing for pre-data and post-data.

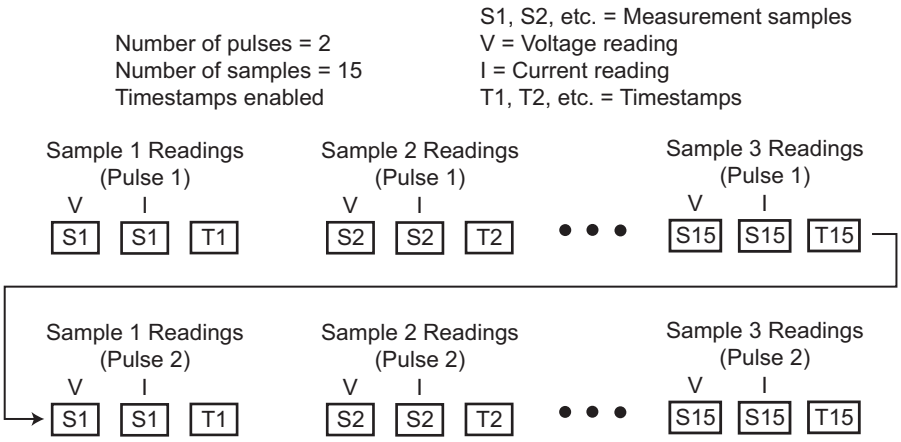
Figure 8-120
Waveform measurements (with pre-data and post-data)



Waveform discrete readings

Enabled voltage and/or current readings and time stamps for every sample of the waveform are returned in a single data set. Figure 8-121 shows how waveform discrete readings are returned as a data set for two pulse periods with voltage, current, and time stamp readings enabled.

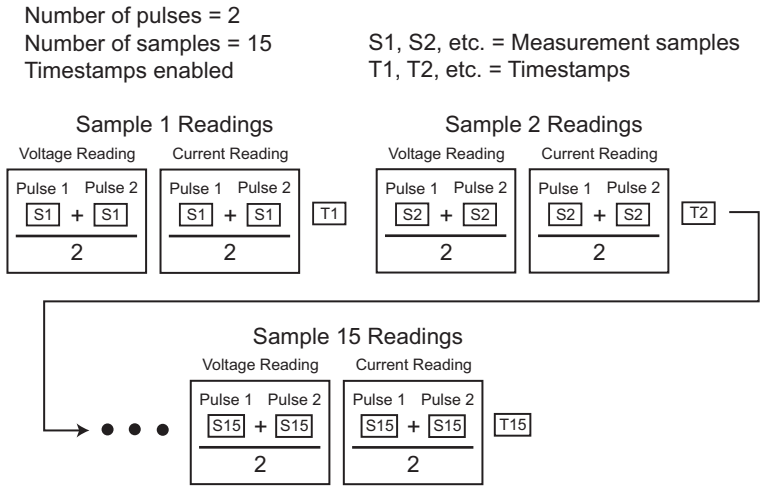
Figure 8-121
Returned data set for waveform discrete readings



Waveform average readings

For this data acquisition type, each returned reading is a mean average of the corresponding samples for all the pulses in the burst. For example, in Figure 8-121 the V and I readings for Sample 1, Pulse 1 would be averaged with the V and I readings for Sample 1, Pulse 2. Figure 8-122 shows how waveform average readings are returned as a data set for two pulse periods with voltage, current, and time stamp readings enabled.

Figure 8-122
Returned data set for waveform average readings



Load line effect compensation (LLEC)

The basic pulse test system is a series circuit that consists of the pulse card resistance (fixed at 50 Ω), interconnect resistance, and the resistance of the DUT. In this series circuit, the sum of the voltage drops across these three components is equal to the output voltage of the pulse generator. Therefore, if the resistance of the DUT changes, the voltage seen at the DUT also changes. This effect is called the “load line effect.” See DUT resistance determines pulse voltage across DUT in Section 11 for details on how the resistance of the DUT affects the voltage across it.

To counter load line effect, the Model 4225-PMU pulse generator has built-in load line effect compensation (LLEC). When LLEC is active, the pulse generator adjusts its output (within its limits) such that the programmed output voltage appears at the DUT.

LLEC uses a software algorithm to adjust the output of the pulser. When enabled, the algorithm performs a set number of iterations in an attempt to output the programmed output voltage to the DUT. The setmode function is used to set the number of iterations for the software algorithm.

NOTE *See Load line effect compensation (LLEC) for the PMU* in Section 16 for details on load line effect compensation.

The `LLEComp` parameter for the `pulse_meas_sm` and `pulse_meas_wfm` functions enables or disables LLEC. The following values can be set for the `LLEComp` parameter to control LLEC:

- Value 1 enables LLEC for amplitude only. It is disabled for base level.
- Value 0 disables LLEC for both amplitude and base level.

The magnitude of the pulse steps affects speed. The larger the magnitude, the longer it takes to perform LLEC.

There may be some high-impedance devices that will not test properly with LLEC enabled. In this case, you can disable LLEC (value 0).

NOTE *Another advantage of using LLEC is that it maintains linear voltage spacing during the test. For example, if the pulse sweep uses 200 mV steps, measurements will be performed at every 200 mV step. Data that is generated using linear voltage spacing is ready to be fed into a mathematical model. For details, see LLEC maintains even voltage spacing in Section 16.*

Measurement timing

The pulse_meas_timing function is used to configure pulse measurement timing

Spot mean measurement timing

The spot mean measurement type samples a portion of the amplitude and a portion of the base level. The portions to be sampled are specified as a percentage.

An example of spot mean measurements is shown in Figure 8-116. As shown, the beginning of the amplitude and base level are designated as the zero (0) percent points; the ends are designated as the 100 percent points. The start and stop points for amplitude and base-level sampling are expressed as a percentage between 0 and 100 percent. In Figure 8-116, sampling for the amplitude and base levels starts at 60 percent (0.6) and ends at 80 percent (0.8).

The number of samples taken on the amplitude and base level is dependent on the size of the portions to be sampled and the sampling rate. Use the pulse_sample_rate function to set the sampling rate for pulse measurements.

Waveform measurement timing

The waveform measurement type samples the pulse as shown in Figure 8-119. Sampling is performed on the entire duration (100 percent) of the pulse. This includes the rise time, amplitude, and fall time portions of the pulse. A voltage and/or current reading is returned for every sample.

A waveform measurement can include pre-data and post-data. Pre-data is extra data taken before the rise time of the pulse; post-data is extra data taken after the fall time.

Figure 8-120 shows an example where 10 percent (0.1) pre-data and 10 percent (0.1) post-data is taken.

The number of samples taken on the pulse is dependent on the size of the pulse to be sampled and the sampling rate. Use the pulse_sample_rate function to set the sampling rate for pulse measurements.

NOTE *In the function, a percentage must be expressed as its decimal equivalent. For example, specify 50 percent as 0.5 in the function.*

Examples

pulse_meas_sm

This function sets channel 1 of the PMU for the spot mean discrete measure type to acquire the voltage amplitude measurement, the current base level measurement, and the time stamps. It also enables LLEC for amplitude:

```
pulse_meas_sm(PMU1, 1, 0, 1, 0, 1, 0, 1, 1);
```

Where:

`Instr_id` = PMU1
`chan` = 1 (channel 1)
`AcquireType` = 0 (discrete)
`AcquireMeasVAmpl` = 1 (enable)
`AcquireMeasVBase` = 0 (disable)
`AcquireMeasIAmpl` = 1 (enable)
`AcquireMeasIBase` = 0 (disable)
`AcquireTimeStamp` = 1 (enable)
`LLEComp` = 1 (enable)

pulse_meas_wfm

This function sets channel 1 of the PMU for the waveform discrete measure type to acquire the voltage/current readings for the waveform and the time stamps. It also enables LLEC.

```
pulse_meas_wfm(PMU1, 1, 0, 1, 1, 1, 1);
```

Where:

`Instr_id` = PMU1
`chan` = 1 (channel 1)
`AcquireType` = 0 (discrete)
`AcquireMeasV` = 1 (enable)
`AcquireMeasI`= 1 (enable)
`AcquireTimeStamp` = 1 (enable)
`LLEComp` = 1 (enable)

pulse_meas_timing

This function sets the following pulse measure timing settings for five spot mean measurements for channel 1 of PMU1:

```
pulse_meas_timing(PMU1, 1, 0.6, 0.8, 5);
```

Where:

`Instr_id` = PMU1
`chan` = 1 (channel 1)
`StartPercent` = 0.6 (60 percent)
`StopPercent` = 0.8 (80 percent)
`NumPulses` = 5 (output one pulse)

pulse_measrt Returns data in pseudo real-time

Purpose This function returns pulse source and measure data in pseudo real-time.

Pulsers Model 4225-PMU

Pulse mode Standard and Segment ARB

Format

```
int pulse_measrt(INSTR_ID instr_id, int chan, char
*VMeasColName, char *IMeasColName, char *TimeStampColName, char
*StatusColName);
```

`instr_id`	Instrument ID: PMU1, PMU2, and so on.
`chan`	Channel number: 1 or 2.
`*VMeasColName`	Column name for V-measure data in the KITE Sheet tab.
`*ImeasColName`	Column name for I-measure data in the KITE Sheet tab.
`*TimeStampColName`	Column name for the time stamp data in the KITE Sheet tab.
`*StatusColName`	Column name for the status data in the KITE Sheet tab.

Remarks Use this function to return and display test data in pseudo real-time. As measurements are performed, the data is automatically placed in the KITE Sheet tab. This function is also used to name the columns in the KITE Sheet tab. This function must be called prior to calling pulse_exec to start the test.

> ***NOTE*** *The pulse_measrt function is not compatible using KXCI to call user libraries remotely (see Calling KULT user libraries remotely in Section 9. Use PostDataDouble for user routines (UTMs) that will be called via KXCI.*

See also `pulse_fetch`

Example The following function configures channel 1 of PMU1 to return data in pseudo real-time:

```
pulse_measrt(PMU1, 1, "V-Measure", "I-Measure", "Timestamp",
"Status");
```

pulse_ranges Sets the pulse voltage and measure ranges

Purpose This function sets the voltage pulse range and voltage/current measure ranges.

Pulsers 4220-PGU and 4225-PMU

Pulse mode Standard and Segment ARB

Format

```
int pulse_ranges(INSTR_ID instr_id, int Chan, double VSrcRange,
int Vrange_type, double Vrange, int Irange_type, double IRange);
```

`instr_id`	Instrument ID: VPU1, VPU2, PMU1, PMU2, and so on.
`Chan`	Channel number: 1 or 2.
`VSrcRange`	Source range: 5 or 20 (into 50 Ω) 10 or 40 (into ≥1 MΩ)

`Vrange_type` Voltage measure range type (PMU):

0 = Auto
1 = Limited auto
2 = Fixed

`Vrange_type` parameter is ignored by the PGU. Autorange (0) and limited autorange (1) are not valid for the Segment ARB pulse mode.

`Vrange` Voltage measure range (PMU)

`Vrange` parameter is ignored by the PGU and is ignored by the PMU when autorange is selected.

`Irange_type` Current measure range type (PMU):

0 = Auto
1 = Limited auto
2 = Fixed

`Irange_type` parameter is ignored by the PGU. Autorange (0) and limited autorange (1) are not valid for the Segment ARB pulse mode.

`Irange` Current measure range (PMU)

`Irange` parameter is ignored by the PGU and is ignored by the PMU when autorange is selected.

Remarks

For the PMU, use this function to set the voltage source range for pulse output, set the voltage and current measure range types, and set the actual voltage and current measure ranges. For the PGU, use this function to set the voltage source range for pulse output.

Measure range types for the PMU:

- Fixed: Use this range type to specify a fixed measure range (`Vrange` or `Irange`).
- Limited Auto: Select this range type to use the fixed measure as the lowest range that will be used for automatic ranging.
- Auto: Use this range type to automatically select the optimum measure range. The specified fixed measure range (`Vrange` or `Irange`) is not used in autorange mode, but must be a valid range.
- The current ranges available depend on the source range and whether the system includes a 4225-RPM, as shown in the next table.

Table 8-22
Available current ranges

Current measure range (A)	PMU source range (V)	RPM source range (V)
0.8	n/a	40
0.2	10	n/a
0.01	10	10 or 40
0.001	n/a	10
0.0001	n/a	10 or 40
0.00001	n/a	10
0.000001	n/a	10
0.0000001	n/a	10

Auto or limited autoranging is available only when using the advanced mode in the pulse_exec function. Ranging is controlled per-channel and may be combined with load

line effect compensation (LLEC) and thresholds (see pulse_limits function for thresholds).

The Segment ARB pulse mode does not allow range changes (no autorange) within a Segment ARB® waveform definition. Only fixed ranging is available for the Segment ARB pulse mode.

See also **PMU and RPM measure ranges are not source ranges**

Example The following function sets the source-measure ranges for channel 1 of PMU1:

```
pulse_ranges(PMU1, 1, 10, 0, 10, 0, 0.2);
```

Where:

`Instr_id` = PMU1
`chan` = 1 (channel 1)
`VSrcRange` = 10 V
`Vrange_type` = Auto (0)
`Vrange` = 10 V (value ignored because V-measure autorange is set)
`Irange_type` = Auto (0)
`Irange` = 200 mA (value ignored because I-measure autorange is set)

pulse_remove Removes a channel from the test

Purpose This function removes a pulse channel from the test.

Pulsers 4220-PGU (VPU) and 4225-PMU

Pulse mode Standard and Segment ARB

Format

```
int pulse_remove(INSTR_ID instr_id, int chan, double voltage,
unsigned long state);
```

`instr_id`	Instrument ID: VPU1, VPU2, PMU1, PMU2, and so on.
`chan`	Channel number: 1 or 2.
`voltage`	Voltage to output when removing a channel.
`state`	Output relay state: PULSE OUTPUT OFF or 0 = Open (disconnected) PULSE OUTPUT ON or 1 = Close (connected)

Remarks Use this function to remove a channel from a test. It is useful when there needs to be one less channel for a pulse test that already exists. The `pulse_remove` function has two behaviors, depending on when it is used.

Use the `voltage` and `state` parameters to remove a channel from a test that is running. Use the `voltage` parameter to set the output voltage. For example, you may want to set the output voltage to zero (0) when removing the channel. Use the `state` parameter to connect or disconnect the channel. The output relay for the PMU is shown in Figure 16-2.

When removing a channel from a test that is not running, the `voltage` and `state` parameters are ignored.

Example The following function removes channel 1 for PMU2, sets the voltage to 0 V, and opens the output relay:

```
pulse_remove(PMU2, 1, 0, 0);
```

pulse_sample_rate Sets the sample rate

Purpose This function sets the measurement sample rate.

Pulsers 4225-PMU

Pulse mode Standard and Segment ARB

Format

```
int pulse_sample_rate(INSTR_ID instr_id, double Sample_rate);
```

`instr_id`	Instrument ID: PMU1, PMU2, and so on.
`Sample_rate`	Sample rate: 200E6, 100E6, 50E6, 40E6, 33E6, 29E6, … 1E3

Remarks Use this card-based function to set the measurement sample rate. The sample rate is the number of measurements (per second) that are performed by the PMU. The sample rate can be set from 200E6 to 200E6/n, where n = 1 to 200,000. The minimum sampling rate is 1E3 samples per second. The sample rate is a fixed rate (not adjustable within a test). For multi-card tests, set all cards to the same sample rate.

If a requested sample rate does not match an available rate, the next higher rate will be used. For example, if 90E6 samples per second is sent, the sampling rate will set to 100E6 samples per second (200E6/2).

Example The following function sets the sampling rate of the PMU to 100E6 samples per second:

```
pulse_sample_rate(PMU1, 100E6);
```

pulse_source_timing Sets pulse source timing

Purpose This function sets the pulse period, pulse width, rise time, fall time, and delay time.

Pulsers 4220-PGU (VPU) and 4225-PMU

Pulse mode Standard

Format

```
int pulse_source_timing(INSTR_ID instr_id, int chan, double
period, double delay, double width, double rise, double fall);
```

`instr_id`	Instrument ID: VPU1, VPU2, PMU1, PMU2, and so on.
`chan`	Channel number: 1 or 2.
`period`	Pulse period (in seconds) for both channels.
`delay`	Delay time (in seconds) for the selected channel.
`width`	Pulse width (in seconds for the selected channel.
`rise`	Rise time (in seconds) for the selected channel.
`fall`	Fall time (in seconds) for the selected channel.

Remarks Use this function to set the timing parameters for the test. Pulse width, rise time, fall time, and delay are individually set for the selected channel. The pulse period setting applies to both channels. See Pulse parameter definitions for more information on these pulse parameters.

This function returns errors if there is an invalid setting or combination of settings.The rise time of a pulse cannot be longer than the pulse width. The minimum time allowed for parameters width, rise, and fall is 20 ns. The minimum value for delay is 0 ns. When setting timing for a sample (waveform capture), setting the delay to a small value allows the PMU to better capture the rising edge of the pulse. This value is sample rate dependent, but for the 200 MSa/s rate, a pulse delay of 20 ns to 100 ns will allow the rising edge of the pulse to be captured.

Another internally enforced limit is the minimum off time. This is calculated as:

minimum off time = period - delay - width - 0.5 x (rise + fall)

The minimum off time may not be less than 40 ns. To see the whole pulse transition to high when capturing waveform data, use a small non-zero value like 10 ns for `pulse_delay`.

When a source timing parameter is already set to step or sweep, the step or sweep parameter overrides the timing parameter set by this function. For details, see pulse_step_linear and pulse_sweep_linear.

For example, if the `SWEEP_PERIOD_SP` parameter type is selected for the `pulse_sweep_linear` function, the period values for the sweep will override the period setting for this function.

Example

This function sets the following pulse source timing settings for the PMU:

```
pulse_source_timing(PMU1, 1, 0.02, 0.005 0.01, 0.001, 0.001);
```

Where:

`instr_id` = PMU1
`chan` = 1
`period` = 0.02 (20 ms)
`delay` = 0.005 (5 ms)
`width` = 0.01 (10 ms)
`rise` = 0.001 (1 ms)
`fall` = 0.001 (1 ms)

pulse_step_linear Configures pulse stepping type
pulse_sweep_linear Configures pulse sweeping type

Purpose

These functions configure the pulse stepping type and pulse sweeping type.

Pulsers

Models 4220-PGU (VPU) and 4225-PMU

Pulse mode

Standard

Format

```
int pulse_step_linear(INSTR_ID instr_id, int chan, int StepType,
double start, double stop, double step);
```

```
int pulse_sweep_linear(INSTR_ID instr_id, int chan int
SweepType, double start, double stop, double step);
```

`instr_id`	Instrument ID: VPU1, VPU2, PMU1, PMU2, and so on.	
`chan`	Pulse generator channel: 1 or 2.	
`StepType/`	`PULSE_AMPLITUDE_SP`	Sweeps pulse voltage amplitude.

SweepType	PULSE_BASE_SP	Sweeps base voltage level.
	PULSE_DC_SP	Sweeps DC voltage level.
	PULSE_PERIOD_SP	Sweeps pulse period.
	PULSE_RISE_SP	Sweeps pulse rise time.
	PULSE_FALL_SP	Sweeps pulse fall time.
	PULSE_WIDTH_SP	Sweeps FWHM (full-width half-maximum) pulse width.
	PULSE_DUAL_BASE_SP	Dual sweeps base voltage level
	PULSE_DUAL_AMPLITUDE_SP	Dual sweeps pulse voltage amplitude
	PULSE_DUAL_DC_SP	Dual sweeps DC voltage level
start	Initial value for stepping/sweeping.	
stop	Final value for stepping/sweeping.	
step	Step size for stepping/sweeping.	

Remarks

Use the pulse_step_linear function to configure stepping. Use the pulse_sweep_linear function to configure sweeping.

The relationship between a step function and a sweep function for SMUs is illustrated in Figure 6-153. The step/sweep relationship for pulsing is similar (see Figure 16-32). While a terminal of a device is at a pulse step, a pulse sweep is performed on another terminal. A pulse_step_linear function cannot be used by itself. At least one PMU channel in a test must be a valid pulse_sweep_linear function call. The last three sweep types are for pulse dual sweeps (see "Dual Sweep Option" on page 16-38).

Use the start, stop, and step parameters to configure stepping/sweeping. In addition, ensure that all pulse parameters are set before calling the pulse_sweep_linear or pulse_step_linear function. For example, when performing a pulse amplitude sweep (PULSE_AMPLITUDE_SP), use pulse_vlow to set the base voltage.

Amplitude and base level:

The pulse generator can step/sweep amplitude (with base level fixed) or step/sweep base level (with amplitude fixed). SweepType examples:

PULSE_AMPLITUDE_SP (stepping or sweeping): Start = 1 V, stop = 5 V, step = 1 V

Voltage amplitudes for pulse output sequence: 1 V, 2 V, 3 V, 4 V, and 5 V.

Note: Use the pulse_vlow function to set the base level voltage.

PULSE_BASE_SP (stepping or sweeping): Start = 5 V, stop = 1 V, step = -1 V

Voltage base levels for pulse output sequence: 5 V, 4 V, 3 V, 2 V, and 1 V.

Note: Use the pulse_vhigh function to set the amplitude voltage.

DC voltage level:

The pulse generator can step/sweep a DC level. SweepType example:

PULSE_DC_SP (stepping or sweeping): Start = 1 V, stop = 5 V, step = 1 V

DC voltage output sequence: 1 V, 2 V, 3 V, 4 V, and 5 V.

Pulse period:

The pulse period is the time interval between the start of the rising transition edge of consecutive output pulses (see Figure 11-27). SweepType example:

`PULSE_PERIOD_SP` (stepping or sweeping): Start = 0.01 s, stop = 0.05 s, step = 0.01 s

Pulse periods for output sequence: 0.01 s, 0.02 s, 0.03 s, 0.04 s, and 0.05 s.

Pulse rise time and fall time:

Pulse rise time is the transition time (in seconds) from pulse low to pulse high. Pulse fall time is the transition time from pulse high to pulse low (see Figure 11-32). `SweepType` examples:

`PULSE_RISE_SP` (stepping or sweeping): Start = 0.001 s, stop = 0.005 s, step = 0.001 s

Rise times for pulse output sequence: 0.001 s, 0.002 s, 0.003 s, 0.004 s, and 0.005 s.

`PULSE_FALL_SP` (stepping or sweeping): Start = 0.001 s, stop = 0.005 s, step = 0.001 s

Fall times for pulse output sequence: 0.001 s, 0.002 s, 0.003 s, 0.004 s, and 0.005 s.

Pulse width:

The width of a pulse (in seconds) is measured at full-width half-maximum (FWHM) as shown in Figure 11-28. `SweepType` example:

`PULSE_WIDTH_SP` (stepping or sweeping): Start = 0.01 s, stop = 0.05 s, step = 0.01 s

Pulse widths for pulse output sequence: 0.01 s, 0.02 s, 0.03 s, 0.04 s, and 0.05 s.

Dual Sweep:

The dual sweep allows for a voltage level sweep that goes up and down based on the voltage start stop and step. For example, a voltage amplitude sweep from 0 V to 4 V in 1 V steps. A single sweep (`PULSE_AMPLITUDE_SP`) would output 5 points: 0 V, 1 V, 2 V, 3 V, 4 V. A dual sweep version (`PULSE_DUAL_AMPLITUDE_SP`) outputs 10 points: 0 V, 1 V, 2 V, 3 V, 4 V, 4 V, 3 V, 2 V, 1 V, 0 V. See Figure 16-37 for a diagram of this example.

Example

The following function configures channel 1 of the PMU to perform an amplitude sweep from 1 V to 5 V in 1 V steps:

```
pulse_sweep_linear(PMU1, 1, PULSE_AMPLITUDE_SP, 1, 5, 1);
```

pulse_train Configures a pulse train

Purpose

This function configures the pulse generator to output a pulse train using fixed voltage values.

Pulsers

Models 4220-PGU (VPU) and 4225-PMU

Pulse mode

Standard

Format

```
int pulse_train(INSTR_ID instr_id, int chan, double Vbase,
double Vamplitude);
```

`instr_id`	Instrument ID: VPU1, VPU2, PMU1, PMU2, and so on.
`chan`	Pulse generator channel: 1 or 2.
`Vbase`	Voltage level for pulse base level.
`Vamplitude`	Voltage level for pulse amplitude.

Remarks The configured pulse train will not change for the selected channel, but any sweep or step timing changes will affect the timing parameters of the train. For details on timing, see pulse_step_linear and pulse_sweep_linear. A `pulse_train` function cannot be used by itself in a test. At least one PMU channel in a test must be a valid `pulse_sweep_linear` function call.

Example The following function configures channel 1 of the PMU to output a 0 to 5 V pulse train:

```
pulse_train(PMU1, 1, 0, 5);
```

rpm_config Configures the Model 4225-RPM

Purpose This function sends switching commands to the 4225-RPM.

Pulsers 4225-PMU with the 4225-RPM

Pulse mode Standard (two-level pulsing), Segment ARB, and full arb

Format

```
int rpm_config(INSTR_ID instr_id, long chan, long modifier, long value);
```

`instr_id`	Instrument ID: PMU1, PMU2, and so on.
`chan`	Pulse generator channel: 1 or 2.
`modifier`	Parameter to modify: `KI_RPM_PATHWAY`.
`value`	Value to set `modifier`: `KI_RPM_PULSE` or 0: Selects pulsing (4225-RPM) `KI_RPM_CV_2W` or 1: Selects 2-wire CVU (4210-CVU) `KI_RPM_CV_4W` or 2: Selects 4-wire CVU (4210-CVU) `KI_RPM_SMU` or 3: Selects SMU (4200-SMU)

Remarks The 4225-RPM includes input connections for the 4210-CVU and 4200-SMU. Use this function to control switching inside the RPM to connect the PMU, CVU, or SMU to the output.

When using the PMU with the RPM, `rpm_config` must be called to connect the pulse source to the RPM output. Note that if there is no RPM connected to the PMU channel, the `rpm_config` command will not cause an error. The RPM connection is cleared by the `clrcon` command.

Example The following function sets channel 1 of the RPM for pulsing:

```
rpm_config(PMU1, 1, KI_RPM_PATHWAY, KI_RPM_PULSE);
```

See also **clrcon**

seg_arb_sequence Defines a Segment ARB pulse-measure sequence

Purpose This function defines the parameters for a Segment ARB pulse-measure sequence.

Pulsers 4220-PGU (VPU) and 4225-PMU

Pulse mode Segment ARB

Format

```
int seg_arb_sequence(INSTR_ID inst_id, long chan, long SeqNum,
long NumSegments, double *StartV, double *StopV, double *Time,
long *Trig, long *SSR, long *MeasType, double *MeasStart, double
*MeasStop);
```

`instr_id`	Instrument ID: VPU1, VPU2, PMU1, PMU2, and so on.
`chan`	The pulse generator channel: 1 or 2.
`SeqNum`	Sequence ID number (1 to 512, per channel) to uniquely identify this sequence.
`NumSegments`	Total number of segments in this sequence.
`StartV`	An array of start voltage levels.
`StopV`	An array of stop voltage levels.
`Time`	An array of segment time durations (in seconds with 10 ns resolution, 20 ns minimum).
`Trig`	An array of trigger values: 0 (trigger low) or 1 (trigger high). These values are for trigger output only.
`SSR`	An array of values to control the high endurance output relay: 0 (open) or 1 (closed).
`MeasType`	PMU only: An array of measure types: 0 = No measurements for this segment 1 = Spot mean discrete (see Figure 8-116) 2 = Waveform discrete (see Figure 8-119 and Figure 8-120) 3 = Spot mean average 4 = Waveform average
`MeasStart`	PMU only: An array of start measurement times (in seconds, with 10 ns resolution). A zero (0) second setting sets measure to start at the beginning of the segment.
`MeasStop`	PMU only: An array of stop measurement times (in seconds, with 10 ns resolution). This is the elapsed time, within the segment, when the measurement stops. The 4220-PGU does not have pulse-measure capability. When this function for the PGU is called, the parameter values for `MeasType`, `MeasStart`, and `MeasStop` are ignored.

Remarks For the PMU, use this function to configure each channel to output its own unique Segment ARB® waveform and perform measurements. For the PGU, use this function to configure each channel to output its own unique Segment ARB waveform.

A Segment ARB sequence is made up of user-defined segments (up to 2048 per channel). Each sequence can have a unique start voltage, stop voltage, time interval, output trigger level (TTL high or low), output relay state (open or closed), pulse

measurement type (for PMU), measurement start time (for PMU), and measurement stop time (for PMU).

A defined sequence is uniquely identified by its specified channel number and sequence ID number. This function defines the sequences, or building blocks, that are typically used for a BTI (bias temperature instability) test.

A sequence is defined as three or more segments with seamless (no voltage differences) voltage transitions. Seamless means that the voltage level for the last point in a segment must equal the voltage level for the first point of the next segment. Note that all segment transitions must be seamless. The minimum time per sequence is 20 ns.

One or more defined sequences are then combined into a Segment ARB® waveform using the seg_arb_waveform function. All sequence transitions must also be seamless. Seamless means that the voltage level for the last point in a sequence must equal the voltage level on the first point of the next sequence. Figure 8-124 shows an example of a waveform that consists of three sequences.

NOTE *See Section 5 of the User's Manual for details on using KPulse to output Segment ARB waveforms.*

Figure 8-123 shows an example of a Segment ARB sequence defined by the `seg_arb_sequence` function. Spot mean discrete measurements are performed on segments two and four.

Figure 8-123
Segment ARB sequence example

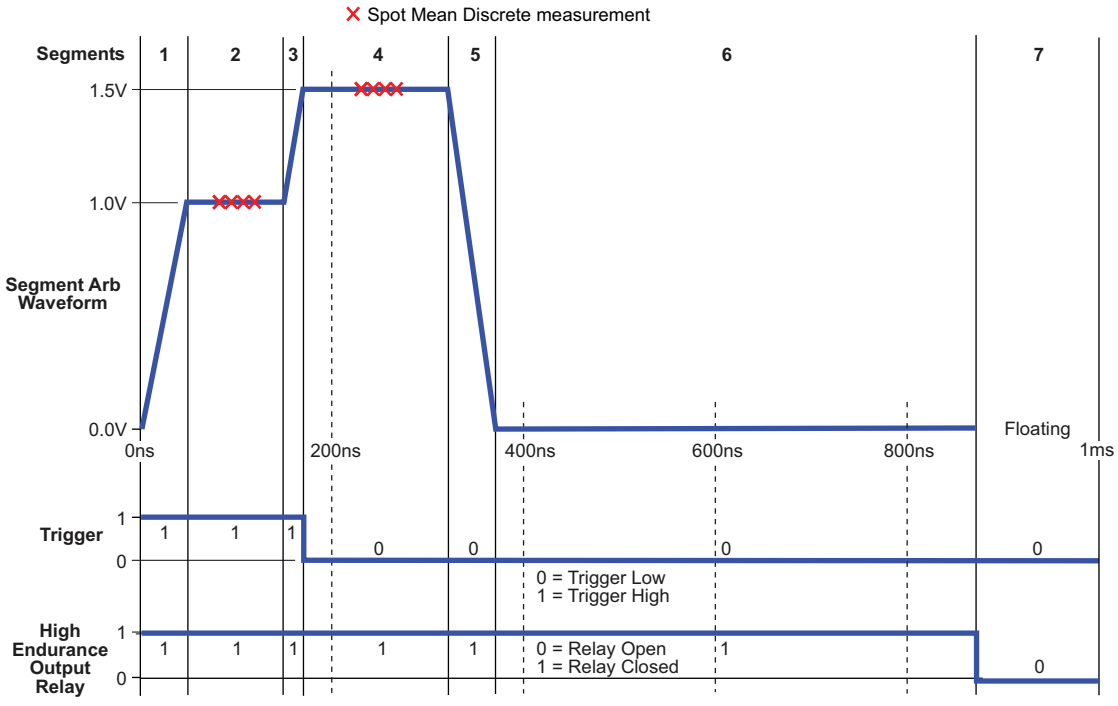


Table 8-23 lists the `seg_arb_sequence` parameter arrays for the Segment ARB sequence shown in Figure 8-123.

Table 8-23
Parameter arrays for the seg_arb_sequence example

Parameter	Array Name	Value						
SegNum	Seg_Num	1	2	3	4	5	6	7
StartV	Start_Volt	0 V	1 V	1 V	1.5 V	1.5 V	0 V	0 V
StopV	Stop_Volt	1 V	1 V	1.5 V	1.5 V	0 V	0 V	0 V
Time	Time_Interval	50e-9 s	100e-9 s	20e-9 s	150e-9 s	50e-9 s	500e-9 s	130e-9 s
Trig	Trigger_Level	1 (high)	1 (high)	1 (high)	0 (low)	0 (low)	0 (low)	0 (low)
SSR	Output_Relay	1 (closed)	1 (closed)	1 (closed)	1 (closed)	1 (closed)	1 (closed)	0 (open)
MeasType	Meas_Type	0 (none)	1 (spot mean)	0 (none)	1 (spot mean)	0 (none)	0 (none)	0 (none)
MeasStart	Meas_Start	0 s	25e-9 s	0 s	50e-9 s	0 s	0 s	0 s
MeasStop	Meas_Stop	0 s	75e-9 s	0 s	100e-9 s	0 s	0 s	0 s

Example

The following function defines the Segment ARB sequence shown in Figure 8-123:

```
seg_arb_sequence(PMU1, 1, 1, 7, Start_Volt, Stop_Volt,
Time_Interval, Trig_Level, Output_Relay, Meas_Type, Meas_Start,
Meas_Stop);
```

Arrays for the `seg_arb_sequence` function:

```
double Start_Volt[7] = {0, 1, 1, 1.5, 1.5, 0, 0};
double Stop_Volt[7] = {1, 1, 1.5, 1.5, 0, 0, 0};
double Time_Interval[7] = {50e-9, 100e-9, 20e-9, 150e-9, 50e-9,
500e-9, 130e-9};
int Trig_Level[7] = {1, 1, 1, 0, 0, 0, 0};
int Output_Relay[7] = {1, 1, 1, 1, 1, 1, 0};
int Meas_Type[7] = {0, 1, 0, 1, 0, 0, 0};
double Meas_Start[7] = {0, 25e-9, 0, 50e-9, 0, 0, 0};
double Meas_Stop[7] = {0, 75e-9, 0, 100e-9, 0, 0, 0};
```

seg_arb_waveform Creates a Segment ARB pulse-measure waveform

Purpose

This function creates a voltage segment waveform.

Pulsers

Models 4220-PGU (VPU) and 4225-PMU

Pulse mode

Segment ARB

Format

```
int seg_arb_waveform(INSTR_ID inst_id, long chan, long NumSeq,
long *Seq, double *SeqLoopCount;
```

`instr_id` Instrument ID: VPU1, VPU2, and so on.

`chan` The pulse generator channel: 1 or 2.

`NumSeq`	Total number of sequences in waveform definition (512 maximum).
`Seq`	An array of sequences using the sequence number ID (see `SeqNum` parameter for the seg_arb_sequence function).
`SeqLoopCount`	An array of loop values (number of times to output a sequence).Loop value range is 1 to 1E12.

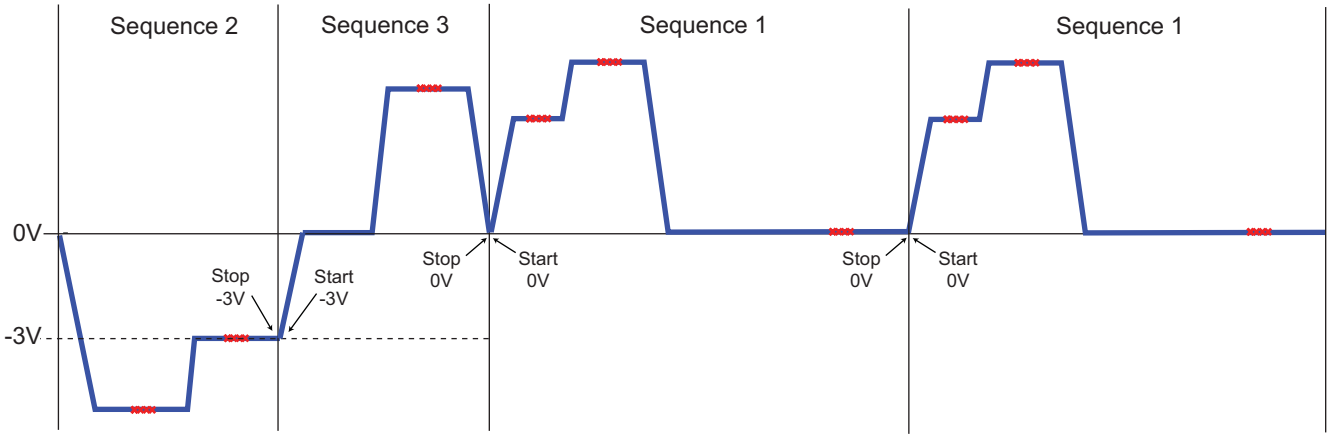
Remarks

Use this function to create a voltage segment waveform from the sequences defined by the seg_arb_sequence function. The `NumSeq` parameter defines the number of sequences that make up the waveform. The `Seq` parameter is an array that indicates the identification (ID) number for each sequence in the waveform. The sequence ID numbers are set by the seg_arb_sequence function.

You can use this function to configure a waveform that repeats one or more of its sequences with the `SeqLoopCount` parameter.

All sequence transitions must be seamless. Seamless means that the voltage level for the last point in a sequence must equal the voltage level on the first point of the next sequence. Figure 8-124 shows an example of a three-sequence waveform that uses looping (Sequence 1 is repeated). Notice that the start and stop voltage values between sequences are the same, making it seamless.

Figure 8-124
Three-sequence waveform (with looping)



Example

The following function configures channel 1 of the PMU for a single three-sequence Segment ARB® waveform (see Figure 8-124). This example assumes that the three sequences shown in Figure 8-124 have already been defined by the seg_arb_sequence function.

```
seg_arb_waveform(PMU1, 1, 3, Seq_Num, Seq_Loop_Count);
```

Arrays for the waveform:

```
int Seq_Num[3] = {2, 3, 1};
double Seq_Loop_Count[3] = {1, 1, 2};
```

setmode Set the number of iterations for load line effect compensation (LLEC)

Purpose This function sets the number of iterations for LLEC for the PMU.

Pulsers Model 4225-PMU

> ***NOTE*** *The following information pertains specifically to the Model 4225-PMU. For details on using the setmode function for other instruments, see setmode for SMU and system settings and setmode for 4210-CVU settings.*

Pulse mode Standard and Segment ARB

Format

```
int setmode(INSTR_ID inst_id, long modifier, double value);
```

`instr_id` Instrument ID of the pulse generator: PMU1, PMU2, and so on.

`modifier` Parameters to set maximum iterations tolerance windows:

`KI_PXU_LLC_MAX_ITERATIONS`	Maximum iterations
`KI_PXU_CH1_LLC_TOLERANCE`	Acceptance window (Ch 1)
`KI_PXU_CH2_LLC_TOLERANCE`	Acceptance window (Ch 2)
`KI_PXU_CH1_LLC_OFFSET`	LLEC Offset (Ch 1)
`KI_PXU_CH2_LLC_OFFSET`	LLEC Offset (Ch 2)

`value` Values to set maximum iterations and tolerance window:

Max iterations: 1 to 1000; typical setting is 20 to 30 iterations
Acceptance window: 0.0001 (0.01 percent) to 1 (100 percent); typical range is 0.001 (0.1 percent) to 0.01 (1 percent), with 0.003 (0.3 percent) the typical value. Offset value: Value is in volts

Remarks Use this function to set the number of iterations for load line effect compensation (LLEC). LLEC is an algorithm, running on each PMU in the test, that adjusts the output of the PMU to respond to the device-under-test resistance and reach the desired (programmed) output value at the DUT. This algorithm is not deterministic, meaning it cannot be guaranteed to reach the desired target value. Therefore, there are controls to fine tune the LLEC performance.

When enabled, the algorithm performs a number of iterations to determine the appropriate output voltage. The pulse_meas_sm and pulse_meas_wfm functions enable or disable LLEC. See Load line effect compensation (LLEC) for more information on LLEC.

LLEC is configured by setting the number of maximum iterations that will be performed and setting an acceptance window for one or both PMU channels. LLEC will continue until either the output voltage to the DUT falls within the acceptance window or until the maximum number of iterations are performed.

Use `KI_PXU_LLC_MAX_ITERATIONS` to set the maximum number of iterations (integer from 1 to 1000) to be performed. LLEC only performs the iterations that are needed to determine the appropriate voltage setting to provide the desired level at the DUT. The remaining iterations are not performed.

Use `KI_PXU_CHx_LLC_OFFSET` to set the offset of the tolerance window

Use `KI_PXU_CHx_LLC_TOLERANCE` to set the gain of the tolerance window (in percentage of desired signal level).

The LLEC tolerance window:

LLEC window = LLC_TOLERANCE * Desired Voltage + LLC_OFFSET

LLEC is satisfied when:

Measured voltage < Desired voltage +/- LLEC Window

For example, assume the programmed pulse output is 1 V and the acceptance window is set to 0.1 (10 percent) and offset to 10 mV. LLEC will perform iterations until the output voltage falls within the 0.9 V to 1.1 V window

Setting a smaller tolerance will result in voltage steps that are much closer to the desired voltage steps sizes, but at the expense of longer test times.

NOTE *When selecting and configuring an LLEC iteration method, remember that testing speed is affected by the maximum number of iterations as well as the tolerance window. Choosing a high maximum number of iterations and a tight tolerance will result in much longer test times.*

Example The following function sets the LLEC for channel 1 of the PMU for a one percent acceptance window:

```
setmode(PMU1, KI_PXU_CH1_LLC_TOLERANCE, 0.01);
```

LPT Library Status and Error codes

There are two types of codes reported in KITE, in the message area shown in Figure 6-1. The positive values are status or updates, and the negative values are errors and warnings. This section does not include error codes for UTMs, such as 4200-PIV-A or 4200-FLASH. See the Model 4200 Applications Manual for the UTM-based application error codes.

Each error code number is associated with a brief text explanation. However, many of the error texts are customized with specific information, such as a particular SMU or ID number. See the Key for an explanation of the type of customized data.

In addition to error codes, some conditions may prevent a valid measurement condition. In these cases, the reported measurement value itself will report a condition that is usually a large number with an exponent of 10^{22} or 10^{23}. See Table 8-26 for the conditions associated with these large numbers.

Table 8-24
LTP Library status and error codes

Key	Explanation
%d	Signed decimal number may be a parameter index or GPIB address
%g	Double value
%i	Signed decimal number
%s	String, such as “SMU1” or other test resource
%u	Unsigned integer
%04x	Hexadecimal number, 4 places
%08x	Hexadecimal number, 8 places

Table 8-25
Code status or error titles

Code	Status or error titles
2802 – 2807	RPM: Invalid Configuration Requested
2801	RPM: Returned ID Error Response
2800	RPM: Command Response Timeout
2702	PMU: Temperature Within Normal Range
2701	PMU: High Temperature Limit Exceeded
1905	PMU: Measure Program Error
1904	PMU: Source Program Error
1902	PMU: Transmission to analog from digital error
1901	PMU: Handshake from analog to digital error
1900	PMU: DA Communication Timeout
400 – 402	PMU: Invalid Attributes in SW Command
100	LPTLib is executing function %s on instrument ID %d.
55	%s is no longer in thermal shutdown.
54	%s VXIBus device busy (command ID %04x). Timed out after %g seconds.
53	%s VXIbus transaction recovered after %u timeouts.
52	%s VXIbus transaction (command ID %04x) timed out after %g seconds.
51	Interlock reset.
50	Interlock tripped.
40	%s
24	Config %d-%d complete for %s (%d).
23	Config %d-%d starting for %s (%d).
22	Binding %s (%d) to driver %s.
21	Loading driver %s.
20	Preloading model code %08x (%s).
15	Executor started.
14	%s channel closed.
13	%s channel starting.
12	TAPI services shutting down.
11	Starting TAPI services.
9	System configuration complete.
8	System configuration starting.
4	System initialization complete.
1	The call was successful (no error).
0	The call was successful (no error).
-4	Too many instruments in configuration file %s.
-5	Memory allocation failure.
-6	Memory allocation error during configuration with configuration file %s.
-20	Command not executed because a previous error was encountered.
-21	Tester is in a fatal error state.
-22	Fatal condition detected while in testing state.
-23	Execution aborted by user.
-24	Too many arguments.
-25	%s is unavailable because it is in use by another test station.
-40	%s.
-87	Can not load library %s.

Table 8-25 (continued)
Code status or error titles

Code	Status or error titles
-88	Invalid configuration file %s.
-89	Duplicate IDs
-90	Duplicate instrument addresses in configuration file %s.
-91	Duplicate instrument slots in configuration file %s.
-93	Unrecognized/missing interface for %s in configuration file %s.
-94	Unrecognized/missing PCI slot number for %s in configuration file %s.
-95	Unrecognized/missing GPIB address for %s in configuration file %s.
-96	GPIB Address out of range for %s was %i in configuration file %s.
-97	PCI slot number out of range for %s was %i in configuration file %s.
-98	Error attempting to load driver for model %s in configuration file %s.
-99	Unrecognized/missing instrument ID in configuration file %s.
-100	Invalid connection count, number of connections passed was %d.
-100	Invalid connection count, number of connections passed was %d.
-101	Argument #%d is not a pin in the current configuration.
-102	Multiple connections on %s.
-103	Dangerous connection using %s.
-104	Unrecognized instrument or terminal not connected to matrix, argument #%d.
-105	No pathway assigned to argument #%d.
-106	Path %d previously allocated.
-107	Not enough pathways to complete connection.
-108	Argument #%d is not defined by configuration.
-109	Illegal test station: %d.
-110	A ground connection MUST be made.
-111	Instrument low connection MUST be made.
-113	There are no switching instruments in the system configuration.
-114	Illegal connection.
-115	Operation not allowed on a connected pin: %d.
-116	No physical bias path from %s to %s.
-117	Connection cannot be made because a required bus is in use.
-118	Cannot switch to high current mode while sources are active.
-119	Pin %d in use.
-120	Illegal connection between %s and GNDU.
-121	Too many calls were made to trigXX.
-122	Illegal value for parameter #%d.
-124	Sweep/Scan measure table overflow.
-126	Insufficient user RAM for dynamic allocation.
-129	Timer not enabled.
-137	Invalid value for modifier.
-138	Too many points specified in array.
-139	An error was encountered while accessing the file %s.
-140	%s unavailable while slaved to %s.
-141	Timestamp not available because no measurement was made.
-142	Cannot bind, instruments are incompatible.
-143	Cannot bind, services unavailable or in use.
-152	Function not supported by %s (%d).
-153	Instrument with ID %d is not in the current configuration.
-154	Unknown instrument name %s.

Table 8-25 (continued)
Code status or error titles

Code	Status or error titles
-155	Unknown instrument ID %i.
-158	VXI device in slot %d failed selftest (mfr ID: %04x, model number: %04x).
-159	VME device with logical address %d is either non-VXI or non-functional.
-160	Measurement cannot be performed because the source is not operational.
-161	Instrument in slot %d has non-functional dual-port RAM.
-164	VXI device in slot %d statically addressed at reserved address %d.
-165	Service not supported by %s (%d).
-166	Instrument with model code %08x is not recognized.
-167	Invalid instrument attribute %s.
-169	Instrument %s is not in the current configuration.
-190	Ill-formed connection.
-191	Mode conflict.
-192	Instrument sense connection MUST be made.
-200	Force value too big for highest range %g.
-202	I-limit value %g too small for specified range.
-203	I-limit value %g too large for specified range.
-204	I-range value %g too large for specified range.
-206	V-limit value %g too large for specified range.
-207	V-range value %g too large for specified range.
-213	Value too big for range selection, %g.
-218	Safe operating area for device exceeded.
-221	Thermal shutdown has occurred on device %s.
-224	Limit value %g too large for specified range.
-230	V-limit value %g too small for specified range.
-231	Range too small for force value.
-233	Cannot force when not connected.
-235	C-range value %g too large for specified range.
-236	G-range value %g too large for specified range.
-237	No bias source.
-238	VMTR not allocated to make the measurement.
-239	Timeout occurred attempting measurement.
-240	Power Limited to 20 W. Check voltage and current range settings.
-250	IEEE-488 time out during data transfer for addr %d.
-252	No IEEE-488 interface in configuration.
-253	IEEE-488 secondary address %d invalid for device.
-254	IEEE-488 invalid primary address: %d.
-255	IEEE-488 receive buffer overflow for address %d.
-261	No SMU found, kelvin connection test not performed.
-262	SRU not responding.
-263	DMM not connected to SRU.
-264	GPIB communication problem.
-265	SRU not mechanically calibrated.
-266	Invalid SRU command.
-267	SRU hardware problem.
-268	SRU kelvin connection problem.
-269	SRU general error.
-270	Floating point divide by zero.

Table 8-25 (continued)
Code status or error titles

Code	Status or error titles
-271	Floating point log of zero or negative number.
-272	Floating point square root of negative number.
-273	Floating point pwr of negative number.
-280	Label #%d not defined.
-281	Label #%d redefined.
-282	Invalid label ID #%d.
-301	PCI ID read back on send error, slot.
-455	Protocol version mismatch.
-510	No command byte available (read) or SRQ not asserted.
-511	CAC conflict.
-512	Not CAC.
-513	Not SAC.
-514	IFC abort.
-515	GPIB timed out.
-516	Invalid function number.
-517	TCT timeout.
-518	No listeners on bus.
-519	Driver problem.
-520	Bad slot number.
-521	No listen address.
-522	No talk address.
-523	IBUP Software configuration error.
-524	No utility function.
-550	EEPROM checksum error in %s: %s.
-551	EEPROM read error in %s: %s.
-552	EEPROM write error in %s: %s.
-553	%s returned unexpected error code %d.
-601	System software internal error; contact the factory.
-602	Module load error: %s.
-603	Module format error: %s.
-604	Module not found: %s.
-610	Could not start %s.
-611	Network error.
-612	Protocol error.
-620	Driver load error. Could not load %s.
-621	Driver configuration function not found. Driver is %s.
-640	%s serial number %s failed diagnostic test %d.
-641	%s serial number %s failed diagnostic test %d with a fatal fault.
-650	Request to open unknown channel type %08x.
-660	Invalid group ID %d.
-661	Invalid test ID %d.
-662	Ill-formed list.
-663	Executor is busy.
-664	Invalid unit ID %d.
-701	Error configuring serial port %s.
-702	Error opening serial port %s.
-703	Call kspcfg before using kspsnd or ksprcv.

Table 8-25 (continued)
Code status or error titles

Code	Status or error titles
-704	Error reading serial port.
-705	Timeout reading serial port.
-706	Terminator not received before read buffer filled.
-707	Error closing serial port %s.
-801	Exception code %d reported from VPU in slot %d, channel %d.
-802	VPU in slot %d has reached thermal limit.
-803	Start and stop values for defined segmented arb violate minimum slew rate.
-804	Function not valid in the present pulse mode.
-805	Too many points specified in array.
-806	Not enough points specified in array.
-807	Function not supported by 4200-VPU.
-808	Solid state relay control values ignored for 4200-VPU.
-809	Time Per Point must be between %g and %g.
810	Attempts to control VPU trigger output are ignored by the 4200-VPU.
-811	Measure range not valid for %s.
-812	WARNING: Sequence %d, segment %d. Cannot measure with PGUs/VPUs.
-820	PMU segment start value %gV at index %d does not match previous segment stop value of %gV.
-821	PMU segment stop time (%g) greater than segment duration (%g)
-822	PMU sequence error for entry %d. Start value %gV does not match previous stop value of %gV.
-823	Start and stop window was specified for PMU segment %d, but no measurement type was set.
-824	Measurement type was specified for PMU segment %d, but start and stop window is invalid.
-825	%s set to post to column %s. Cannot fetch data that was registered as real-time.
-826	Cannot execute PMU test. No channels defined.
-827	Invalid pulse timing parameters in PMU Pulse IV test.
-828	Maximum number of segments per PMU channel exceeded (%d).
-829	The sum of base and amplitude voltages (%gV) exceeds maximum (%gV) for present range.
-830	Pulse waveform configuration exceeded output limits. Increase pulse period or reduce amplitude or total time of pulsing.
831	Maximum number of samples per channel (%d) exceeded for PMU%d-CH%d.
-832	Pulse slew rate is too low. Increase pulse amplitude or reduce pulse rise and fall time.
-833	Invalid trigger source for PIV test.
-834	Invalid pulse timing parameters.
-835	Using the specified sample rate of %g samples/s, the time (%g) for sequence %d is too short for a measurement.
-836	WARNING: Sequence %d, segment %d is attempting to measure while solid state relay is open. Disabling measurement.
-837	No RPM connected to channel %d of PMU in slot %d.
-838	Timing parameters specify a pulse that is too short for a measurement using %g samples/s.
-839	Timing parameters contain measurement segment(s) that are too short to measure using %g samples/s.
-840	SSR cannot be opened when using RPM ranges. Please change SSR array to enable relay or select PMU measure range.
-841	WARNING: SSR is open on segment immediately preceding sequence %d. Measurement will be invalid for 25 us while relay settles.
-842	This test has exceeded the system power limit by %g watts.
-843	Step size of %g is not evenly divisible by 10 ns.

Table 8-25 (continued)
Code status or error titles

Code	Status or error titles
-844	Invalid combination of start %g1, stop %g2 and step %g3.
-845	No pulse sweeper was configured - Test will not run.
-846	Maximum Source Voltage Reached: Requested voltage across DUT resistance exceeds maximum voltage available.
-847	Output was not configured - Test will not run.
-848	Sweep step count mismatch for the sweeping channels. All sweeping channels must have same # of steps.
-849	ILimit command is not supported for RPM in slot %d, channel %d.
-850	Sample Rate mismatch. All channels in test must have the sample rate.
-851	Invalid PxU stepper/sweeper configuration.
-900	Environment variable KI_PRB_CONFIG is not set. The prober drivers will be inaccessible.
-901	Environment variable KI_PRB_CONFIG contains an invalid path. The prober drivers will be inaccessible.
-902	Prober configuration file not found. File was %s. The prober drivers will be inaccessible.
-903	Unable to copy the prober configuration %s to %s. The prober driver many not be unable.
-10000 – -20000	User Module (UTM) error codes. Refer to user module description (help) for details.

Table 8-26
Large number reported readings and explanations

Measurement Value	Condition
1.0000E+22 or 10.0000E+21	SMU is in range compliance (see Types of compliance in Section 3), where the reading is at the maximum of a fixed range.
5.0000E+22	SMU measurement at the maximum SMU voltage or current.
7.0000E+22	SMU in real compliance (see Types of compliance in Section 3).
1.0000E+23	Measurement aborted or not able to be performed. For example, if using LPT command to make a measurement, but the SMU output was not enabled, then this measurement value will be reported.

LPTLib and KITE interaction via UTMs

ITMs and UTMs are typically independent. However, an ITM and a UTM are not independent if 1) the UTM occurs before an ITM in the project plan, and 2) the UTM configures a switch matrix. Under these conditions, the following occur:

- KITE assumes that the ITM depends on the UTM-created switch configuration.
- KITE maintains the UTM-created switch configuration during execution of the ITM.

Refer to Table 8-27.

Table 8-27
KITE actions affected by ITM and UTM sequence

Test sequence in the project plan	KITE action
A UTM precedes an ITM	Before the ITM executes, the `devint` function initializes all devices, *except* for the switch matrix (the switch configuration is preserved to run the subsequent ITM).
A UTM precedes a UTM	No initialization operations occur.
An ITM precedes an ITM	No LPTLib calls occur.
An ITM precedes a UTM	Before the UTM executes, the `devint` function initializes all devices, *including* the switch matrix.

Cross-platform LPTLib compatibility

The LPT Library (LPTLib) is included with the 4200-SCS to provide an application programming interface (API) for controlling instrumentation and accessing I/O. LPTLib is available on the Keithley Instruments Models S400 and S600 series parametric test systems for the same purpose. Therefore, user libraries can be moved between the Keithley Instruments Model 4200-SCS, S400, and S600 test systems. In most cases, user libraries can be moved from one system to another with little or no modification to the user library source code. Simply recompile and build the library on the target platform, and the library is ready for use. However, in some cases it is necessary to modify the user library to address platform-specific differences in LPTLib functions.

Table 8-28 indicates the compatibility of each LPTLib function across all LPTLib-based Keithley Instruments test systems, using the following symbols:

- A dash (**-**) indicates that the corresponding LPTLib function is not supported on the indicated Keithley Instruments test system.
- An **X** indicates that the function is supported on the indicated Keithley Instruments test system.
- A superscripted numeral (**1**, **2**, or **3**) next to a dash (**-**) or **X** indicates that the corresponding LPTLib function behaves differently on each platform. This number refers to a footnote that describes the significant platform differences.

Unless otherwise indicated, unsupported LPTLib functions in Table 8-28 cannot be used in a KULT user module. Use of an unsupported function causes a build error of the form:

```
<module_name>.obj: error LNK2001: unresolved external symbol _<function_name>
```

Use of an unsupported function may also cause a compilation error of the form:

```
<path\module_name>.c: warning C4013: _<function_name> undefined;...
```

For more detailed information regarding moving 4200-SCS user libraries to and from the Models S400 and S600, refer to Moving user libraries: 4200-SCS to S400 and Moving user libraries: Model 4200-SCS to a Model S600/S630 later in this section.

Table 8-28
LPTLib function compatibility

Group	Function / module	4200-SCS	S400	S600
Instrument	devclr	X	X	X
	devint	X	X	X
	setvims	-	X	-
	setvmtr	-	X	-
	setimtr	-	X	-
Matrix	addcon	X	X	X
	conpin	X	X	X
	conpth	X	X	X
	clrcon	X	X	X
	delcon	X	X	X
	floatpin	-	-	X
Ranging	atten	-	X	-
	lorangei	X	X	X
	lorangev	X	X	X
	lorangec	-	X	X
	lorangeg	-	X	X
	rangei	X	X	X
	rangev	X	X	X
	rangec	- [1]	X	X
	rangeg	- [1]	X	X
	setauto	X	X	X
Sourcing	forcei	X	X	X
	forcev	X	X	X
	limiti	X	X	X
	limitv	X	X	X
	outebl	-	X	-
	pulsev	X	X	-
	pulsei	X	X	-
Measuring	avgi	X	X	X
	avgv	X	X	X
	avgc	- [1]	X	X
	avgg	- [1]	X	X
	fltoff	-	X	-
	flton	-	X	-
	intgi	X	X	X
	intgv	X	X	X
	intgc	-	X	X
	intgg	-	X	X
	measi	X	X	X
	measv	X	X	X
	measc	- [1]	X	X

Table 8-28 (continued)
LPTLib function compatibility

Group	Function / module	4200-SCS	S400	S600
Measuring (continued)	measg	-[1]	X	X
	measf	-	X	-
	meast	X	X	X
	setac	-	X	-
	setdc	-	X	-
	setfilter	-	X	X
	setgate	-	X	-
	settrig	-	X	-
	ssmeasi	-	X	X
	ssmeasv	-	X	X
	ssmeasc	-	X	-
	ssmeasg	-	X	-
	nslope	-	X	-
	pslope	-	X	-
	zchoff	-	X	-
	zchon	-	X	-
Combination	asweepi	X	X	X
	asweepv	X	X	X
	bmeasi	X	X	-
	bmeasv	X	X	-
	bmeasc	-	X	-
	bmeasg	-	X	-
	bsweepv	X	X	X
	bsweepi	X	X	X
	clrtrg	X	X	X
	clrscn	X	X	X
	mpulse	X	X	-
	rtfary	X	X	X
	savgi	X	X	X
	savgv	X	X	X
	savgc	-	X	X
	savgg	-	X	X
	scnmeas	X	-	X
	searchi	X	X	X
	searchv	X	X	X
	sintgi	X	X	X
	sintgv	X	X	X
	sintgc	-	-	X

Table 8-28 (continued)
LPTLib function compatibility

Group	Function / module	4200-SCS	S400	S600
Combination (continued)	sintgg	-	-	X
	smeasi	X	X	X
	smeasv	X	X	X
	smeasc	- [1]	X	X
	smeasg	- [1]	X	X
	smeast	X	X	X
	sweepi	X	X	X
	sweepv	X	X	X
	trigcomp	X	-	X
	trigig	X	X	X
	trigvg	X	X	X
	trigcg	-	X	X
	triggg	-	X	X
	trigrg	-	X	-
	trigfg	-	X	-
	trigtg	X	X	X
	trigil	X	X	X
	trigvl	X	X	X
	trigcl	-	X	X
	triggl	-	X	X
	trigrl	-	X	-
	trigfl	-	X	-
	trigtl	X	X	X
Timing	adelay	X	X	X
	delay	X	X	X
	rdelay	X	X	X
	enable	X	X	X
	imeast	X	X	X
	disable	X	X	X
	retmrstats	-	X	X
Pulse	arb_array	X	-	-
	arb_file	X	-	-
	pg2_init	X	-	-
	pulse_burst_count	X	-	-
	pulse_current_limit	X	-	-
	pulse_dc_output	X	-	-
	pulse_delay	X	-	-
	pulse_fall	X	-	-
	pulse_halt	X	-	-

Table 8-28 (continued)
LPTLib function compatibility

Group	Function / module	4200-SCS	S400	S600
Pulse (continued)	pulse_init	X	-	-
	pulse_load	X	-	-
	pulse_output	X	-	-
	pulse_output_mode	X	-	-
	pulse_period	X	-	-
	pulse_range	X	-	-
	pulse_rise	X	-	-
	pulse_ssrc	X	-	-
	pulse_trig	X	-	-
	pulse_trig_output	X	-	-
	pulse_trig_polarity	X	-	-
	pulse_trig_source	X	-	-
	pulse_vhigh	X	-	-
	pulse_vlow	X	-	-
	pulse_width	X	-	-
	seg_arb_define	X	-	-
	seg_arb_file	X	-	-
Execution & Synchronization	execut	X[2]	X	X
	inshld	X	X	X
	kthtimo	-	X	-
	rexcut	-	X	-
	syncmode	-	-	X
	xrf_buffer_size	-	X	-
Arithmetic	kfpabs	X[3]	X	X[3]
	kfpadd	X[3]	X	X[3]
	kfpdiv	X[3]	X	X[3]
	kfpexp	X[3]	X	X[3]
	kfplog	X[3]	X	X[3]
	kfpmul	X[3]	X	X[3]
	kfpneg	X[3]	X	X[3]
	kfppwr	X[3]	X	X[3]
	kfpsqrt	X[3]	X	X[3]
	kfpsub	X[3]	X	X[3]
Parallel I/O	pior	-	X	X
	piorb	-	X	X
	piow	-	X	X
	piowait	-	X	X
	piowb	-	X	X

Table 8-28 (continued)
LPTLib function compatibility

Group	Function / module	4200-SCS	S400	S600
GPIB	ibup	-	X	X
	kibcmd	X	X	X
	kibdefclr	X	X	X
	kibdefint	X	X	X
	kibdefdelete	X	-	-
	kibrcv	X	X	X
	kibsnd	X	X	X
	kibspl	X	X	X
	kibsplw	X	X	X
RS-232	kspcfg	X	-	-
	kspsnd	X	-	-
	ksprcv	X	-	-
	kspdefclr	X	-	-
	kspdefdelete	X	-	-
	kspdefint	X	-	-
Branch	klpbeq	-	X	-
	klpbge	-	X	-
	klpbgt	-	X	-
	klpble	-	X	-
	klpblt	-	X	-
	klpbne	-	X	-
	klpbra	-	X	-
	klplbl	-	X	-
General	beep	-	-	X
	compclr	-	-	X
	getinstid	X	-	-
	getinstname	X	-	-
	getinstattr	X	-	-
	getstatus	X	X	X
	insbind	-	-	X
	insinfo	-	-	X
	setmode	X	X	X
	refctrl	-	-	X
	prbsel	-	X	-
	tstdsl	X	X	X
	tstsel	X	X	X

Table 8-28 (continued)
LPTLib function compatibility

Group	Function / module	4200-SCS	S400	S600
Error handling	`display_lpt_error`	-	X	-
	`extract_lpt_error`	-	X	-
	`getlpterr`	X	X	X
	`log_lpt_error`	-	X	-
	`kthvmerror`	-	X	-

Notes:
1. LPTLib functions to facilitate capacitance measurements are not directly supported on the Model 4200-SCS. However, user libraries for controlling the Keithley Instruments Model 590 CV Analyzer and the Hewlett Packard Model 4980 LCR Meter are provided with the 4200-SCS. Refer to Capacitance-meter support differences later in this section for more information.
2. `execut()` simply calls `devint()` on the 4200-SCS. It does not execute the program.
3. Provided for legacy user library compatibility purposes only. Usage of this LPTLib command is not recommended. Use the ANSI C-language equivalent.

S400/S600 functions not supported by the Model 4200-SCS

The following list summarizes the functions not supported by the 4200-SCS:

- **Database calls**: `PutLot, PutWafer, PutSite, PutParam, PutParamList, EndLot, EndWafer, EndSite, GetLot, GetWafer, GetSite, GetParam, GetParamList, GetLotData, LogLot, LotWaf, LogSit, LogPtr, LogPta, MatchParam2Limit, FileExist, LotExist, GetStartTime, DeleteLot, DeleteWafer, DeleteSite, DeleteParam, DeleteLimitCode, DeleteLimit, GetComment, PutComment, GetLimitCode, GetLimit, PutLimit, GetLotStats, GetWaferStats, FindLot, FindData, AddNewLimit, CreateNewLimit, FindFirstLimit, FindLastLimit, FindNextLimit, FindPrevLimit, InsertNewLimit, RemoveLimit`
- **KUI (Keithley user interface) calls:** `GetProgramArgs, InitUI, InputMsgDlg, LotDlg, OkCancelAbortMsgDlg, OkCancelDlg, OkMsgDlg, QuitUI, ScrollMsgDlg, ScrollMsgDlgMsg, StatusDlg, UpdateModelessDlgs, UpdateStatusDlg, VerifyAbort, WfrIdsDlg, WfrIdDlg, YesNoAbortMsgDlg, YesNoCancelMsgDlg`
- **KWF (Keithley wafer file) calls:** `AddNewSubsite, CreateNewSubsite, FindFirstSubsite, FindNextSubsite, FindSubsiteId, readWDF`
- **PARLIB (PARameter LIBrary) calls:** `Bchk, Beta1, Beta2, Beta2a, Beta3a, Bice, Bice1, Bice2, Bvcbo, Bvcbo1, Bvceo, Bvceo1, Bvceo2, Bvces, Bvces1, Bvebo, Ev, Ibic1, Ibic2, Ibic3, Icbo, Iceo, Ices, Iebo, Is1, Is2, Pbice, Pbice1, Pbice2, Picib, Prb, Pvcic, Pvcicr, Rb, Rcsat, Re, Rev, Vbes, Vbibic, Vcesat, Vcic, Vcicr, Bkdn, Cap, Capg, Con, Evalcj, Fimv, Fndcj, Fvmi, Leak, Meascp, Measc s, Pcp, Pcs, Psimv, Psvmi, Rcont, Res, Res2, Res4, Resv, Rsq, Rvdp, Simv, Svmi, Tox, Vf, Bvdss, Bvdss1, Deltl1, Deltw1, Gamma1, Gd, Id1, Idsat, Idvsvd, Idvsvg, Isubmx, Pidvd, Pidvg, Pimax, Pvtvbs, Vg2, Vg2a, Vgsat, Vt14, Vt14s, Vtati, Vtext, Vtext2, Vtext3, Vtvbs, Gm, Idss, Imax, Rsd, Rsg, Vp, Vp1`

Moving user libraries: 4200-SCS to S400

This section describes the issues involved with moving an S400UX C-language function to the 4200-SCS and moving a 4200-SCS function to the S400UX. It is important to note that this section does not cover the porting of code from the S400 VAX to the 4200-SCS, because FORTRAN-to-C code conversions are beyond the scope of this document.

Header files

When you move functions, generally you need not move header files if they are covered by the ANSI C standard (for example, `stdio.h`, `time.h`, and so on). However, three important exceptions follow:

- The first exception is in the case of *absolute path names*. If the S400 UNIX path name is hard-coded in the source code as follows:

  ```
  /pathname1/pathname2/HeaderFileName.h
  ```

  then in the 4200-SCS, this path name must be corrected to reflect the header file's location on the embedded computer hard drive. For example:

  ```
  c:\pathname1\pathname2\HeaderFileName.h
  ```
- The second exception is in the case of non-standard or UNIX-specific header files. For example, the Model S400 header file `/usr/include/sys/asynch.h` has no Windows® equivalent. You must locate a suitable replacement for such a header file when using an associated function in the 4200-SCS.
- The third exception is in the case of S400 header files related to the Keithley Instruments KDF (Keithley data files) database. There is no equivalent to the KDF database on the 4200-SCS. Therefore, when using S400 code in a 4200-SCS, remove any reference to the KDF database.

Instrument hardware differences

On the S400, the source/measure units were referred to as VIMS. Therefore, when using S400 code in the 4200-SCS, you must replace any instance of the string `VIMS` with the string `SMU`.

The terminals that are referred to as GPTs (general-purpose terminals) on the S400 are referred to as GPIs (general-purpose instruments) on the 4200-SCS. Therefore, you must replace the string `GPT` in the S400 code with the string `GPI` in the 4200-SCS code.

The following instruments that were supported on the S400 are either not supported on the 4200-SCS or are used in a different way on the 4200-SCS:

- **FMTR**: The 4200-SCS does not support a frequency counter.
- **PSRC**: The 4200-SCS does not support the power source. In many instances, if the current needed is 1 A or less, you may be able to use an SMU as a replacement.
- **VMTR**: There is no separate voltmeter for the 4200-SCS. The S400 used separate voltmeters, such as the KI2001 and Model 192/196, to provide low-voltage measurement capabilities. On the 4200-SCS, an SMU provides equivalent low-voltage measurement capabilities.
- **IMTR**: There is no separate current meter for the 4200-SCS. The S400 uses separate current meters, such as the Model 617 and 9162-PAU (VME), to provide picoampere-level measurement capabilities. The 4200-SCS SMUs can measure at picoampere and sub-picoampere levels.
- **VSRC**: The 4200-SCS does not currently support high-voltage instruments.
- **CMTR**: The 4200-SCS supports the KI590 and HP4980 capacitance meters. However, the 4200-SCS supports these capacitance meters through user libraries that are not call-compatible with S400 user libraries or S400 LPT functions (on the S400, the CMTR was supported through VME-level software drivers). For more information about the differences, refer to Capacitance-meter support differences later in this section.

As a rule of thumb, VMTRs, PSRCs, and IMTRs of the S400 can be replaced with SMUs on the 4200-SCS. The functionality of the ki590ulib and hp4980ulib user libraries is equivalent to the functionality of the S400 LPT capacitance-meter functions.

Instrument range differences

Table 8-29 shows the range differences between the 4200-SCS SMUs and the S400 VIMS.

Table 8-29
Range differences: 4200-SCS SMUs and S400 VIMS

System	Instrument	Voltage ranges	Current ranges
S400	VIMS	200 V, 40 V, 4 V, 0.4 V	200 mA, 20 mA, 2 mA, 200 µA, 20 µA, 2 µA, 200 nA, 20 nA
	IMTR (PAU)	Not applicable	20 mA, 2 mA, 200 µA, 20 µA, 2 µA, 200 nA, 20 nA, 2 nA, 200 pA
	Microvoltmeter	200 mV, 2 V, 20 V, 200 V	Not applicable
	MIVS (medium current voltage source, PSRC)	20 V	1 A, 10 A
4200-SCS	SMU	200 mV, 2 V, 20 V, 200 V	1 pA, 10 pA, 100 pA, 1 nA, 10 nA, 100 nA, 1 µA, 10 µA, 100 µA, 1 mA, 10 mA, 100 mA, 1 A

Capacitance-meter support differences

The 4200-SCS and S400 systems support capacitance meters in very different ways. When measuring capacitance and conductance on the S400, you generally used standard capacitance/conductance-measurement functions such as `measc`, `measg`, `intgc`, `intgg`, `rangec`, `rangeg`, `avgc`, `avgg`, `savgc`, `savgg`, `sintgc`, and `sintgg`. The 4200-SCS uses no equivalent commands. Instead, the 4200-SCS supports capacitance meters through Keithley Instruments-supplied user-library user modules (or, more specifically, through KITE UTMs that are connected to Keithley Instruments-supplied user modules). Table 8-30 provides guidance in substituting 4200-SCS user modules for S400 functions.

Table 8-30
Capacitance-meter support differences: 4200-SCS SMUs and S400 VIMS

S400 CMTR function PostDataDouble (PMU)	**Equivalent Model 4200-SCS user-library user module**
`measc, measg`	Cmeas590 or Cmeas4284
`bmeasc, bmeasg, intgc, intgc, sintgc, sintgg, ssmease, ssmeasg, trigcg, triggg, trigcl, triggl`	n/a
`smeasc, smeasg`	Cvsweep590 or Cvsweep4284
`rangec, rangeg`	Cmeas590 or Cmeas4284
`forcev(CMTRx…)`	Cmeas590 or Cmeas4284
`avgc, avgg`	Use Cmeas590 or Cmeas4284 and mathematically average

Absence of the KDF database on the 4200-SCS

There is no equivalent to the Keithley Instruments KDF database on the 4200-SCS. You must eliminate the database calls when porting to the 4200-SCS.

Parameter differences

Many Keithley Instruments LPT functions on the S400 required `float` input arguments. On the 4200-SCS, these same calls require `double` input arguments. For example, if your S400 function contained the following:

```
float range = 1e-6;
       *
       *
       *
rangei(VIMS1, range);
```

this code must be changed to the following to work on the 4200-SCS:

```
double range = 1e-6;
      *
      *
      *
rangei(SMU1, range);
```

LPT execution differences

There are fundamental differences between the ways the 4200-SCS and S400 execute LPT commands. These differences are listed below:

- **Command synchronization**:
  On the S400, LPT commands were first downloaded to the VME instruments in packets. These packets were executed completely and independently of the applications processor (for example: the Sun workstation) and only returned results when an `execut`, `inshld`, or `rexcut` command was encountered. This is called results synchronization. By contrast, on the 4200-SCS, all commands are executed sequentially by the internal CPU and results are returned as soon as they are available.

  When moving code from the S400 to the 4200-SCS, no changes to your code are necessary to accommodate this fundamental difference. However, when moving code from the 4200-SCS to the S400, you must add the `execut`, `inshld`, or `rexcut` commands at the appropriate location in your source code. For example, if your 4200-SCS code performs a mathematical operation on a measured result, such as the following:

  ```
  measv(SMU1, &voltage);
  resistance = voltage/current;
  ```

  you would need to add a call, after the `measv` call, either to `inshld` (which holds all instruments in their current state) or to `execut` (which executes all previous LPT commands and returns all instruments to their default states). In the latter case, your code would be modified to read as follows:

  ```
  measv(SMU1, &voltage);
  execut();
  resistance = voltage/current;
  ```

- **Instrument clearing**:
  On the S400, the `execut` command caused all previously issued LPT commands to execute. Once all LPT commands executed, all instruments were cleared and set to their default states, and all matrix connections were cleared. However, on the 4200-SCS, you may call `execut()`, which initiates a `devint()` call.

Moving user libraries: Model 4200-SCS to a Model S600/S630

The issues described above under Moving user libraries: 4200-SCS to S400 also apply to moving KULT libraries between the S600 and 4200-SCS, with the following exceptions:

- **Instrument names**:
  The source-measure units on the S600 and 4200-SCS have the same instrument IDs (for example, SMUx, where x = 1 to 8). You do not need to make any changes to your code to change the SMU instrument IDs.
- **Parameter differences**:
  The input parameter types for Keithley measurement functions are exactly the same on the S600 and the 4200-SCS.
- **Instrument differences**:
  On the S600, the general-purpose instrument terminals have instrument IDs called FOHMx (where x is 1 to 8). On the 4200-SCS, the equivalent terminals have instrument IDs called GPIx (where x is an integer). When moving C-language code between the platforms, you must make the appropriate substitutions or conditionalize your code with the `#ifdef` preprocessor statement.
- **SMU current range differences**:
  When equipped with a preamplifier, a 4200-SCS SMU has two additional ranges that do not exist on the S600: 10 pA and 1 pA. When moving source code to the S600, make sure that you make the appropriate changes to your source code.
- **SMU voltage ranges**:
  The 4200-SCS and S600 have identical voltage ranges.
- **LPT function differences**:
  Refer to Table 8-28 for the list of LPT functions supported by the 4200-SCS and S600. In cases where functions on one system are not supported on the other system, you must make appropriate changes to the source code when moving between the systems.
- **Command synchronization**:
  When moving code from the 4200-SCS to the S600, you generally do not need to make changes to the code to account for command synchronization. When moving code from the S600 to the 4200-SCS, be aware that the S600 has several synchronization modes that are not supported by the 4200-SCS. However, it is generally safe to comment out any code that changes the S600 synchronization method.